1-1-2021

Extensible Embedded Processor for Convolutional Neural Networks

Joshua Misko
San Jose State University

Shrikant S. Jadhav
San Jose State University, shrikant.jadhav@sjsu.edu

Youngsoo Kim
Bradley University

Recommended Citation

Joshua Misko, Shrikant S. Jadhav, and Youngsoo Kim. "Extensible Embedded Processor for Convolutional Neural Networks" *Scientific Programming* (2021). https://doi.org/10.1155/2021/6630552

Hindawi
Scientific Programming
Volume 2021, Article ID 6630552, 12 pages
https://doi.org/10.1155/2021/6630552

Research Article

Extensible Embedded Processor for Convolutional Neural Networks

Joshua Misko,[1] **Shrikant S. Jadhav**,[2] **and Youngsoo Kim**[3]

[1]*San Jose State University, San Jose, USA*
[2]*Fort Lewis College, Durango, USA*
[3]*Bradley University, Peoria, USA*

Correspondence should be addressed to Youngsoo Kim; ykim@fsmail.bradley.edu

Received 23 November 2020; Revised 11 March 2021; Accepted 3 April 2021; Published 21 April 2021

Academic Editor: Manuel E. Acacio Sanchez

Convolutional neural networks (CNNs) require significant computing power during inference. Smart phones, for example, may not run a facial recognition system or search algorithm smoothly due to the lack of resources and supporting hardware. Methods for reducing memory size and increasing execution speed have been explored, but choosing effective techniques for an application requires extensive knowledge of the network architecture. This paper proposes a general approach to preparing a compressed deep neural network processor for inference with minimal additions to existing microprocessor hardware. To show the benefits to the proposed approach, an example CNN for synthetic aperture radar target classification is modified and complimentary custom processor instructions are designed. The modified CNN is examined to show the effects of the modifications and the custom processor instructions are profiled to illustrate the potential performance increase from the new extended instructions.

1. Introduction

Convolutional neural networks (CNNs) have become increasingly popular for image classification and a variety of other machine learning tasks. Existing methods either required massive computational power or frequently performed poorly when new cases were presented for classification. CNNs have become popular because they are increasingly accurate classifiers as the networks are trained on more data without incurring the increase in model size for new learning. AlexNet, trained with convolutional neural network, won the ILSVRC in 2012, and its victory marked the beginning of CNNs as the premier method for image classification.

Fixed classifier model size makes the CNN an attractive platform for mobile and embedded applications with memory and speed constraints. A CNN can be trained in an environment with massive computational capability and large datasets and then deployed on platforms with limited computational power. Although they are more efficient than other classifier types that can be trained with large datasets, CNNs are still computationally intensive applications. Large CNNs require billions of operations to classify a single image [1]. In CNNs, most of the execution time is consumed by convolution operations.

In addition to computation requirements, memory access penalties significantly impact overall execution time and power consumption. The weights and biases used for inference can approach a gigabyte for large CNN models when stored in single-precision floating point format [1]. In addition, data inputs and outputs for each layer also consume significant amounts of memory. Running a CNN purely from fast, on-chip memory is not feasible for large CNNs. Once CNNs need to access external memory such as DRAM, hundreds of cycles per access are typically added to the overall execution due to DRAM latency. Also, external memory accesses incur energy consumption penalties up to 128 times greater than on-chip memory [2]. Minimizing access to external DRAM can drastically improve efficiency.

The weights, biases, and activations of the hidden layers of CNNs can be converted to fixed-point formats with low bit-width representations without hurting overall classification accuracy [3]. The quantization noise has a regularizing effect on the CNN and reduces negative effects of lower precision on classification performance. In the extreme case, binary CNN models can be used if the loss in classification accuracy is tolerated [4]. Converting to smaller bit-width representations of weights and data in the middle layers of a CNN drastically reduces the number of memory accesses and increases execution speedup in real systems. Multiple data items can be packed into a single register enabling SIMD operations running on all lanes in the register in parallel. In addition, fixed-point multipliers have simpler hardware implementations compared to floating point. For the same chip area, multiple small fixed-point multipliers increase the computational throughput for convolution and fully connected layers.

GPUs are the preferred method of training and running CNNs in research because they hide memory access penalties by compensating for image throughput. Gigabytes of training images are loaded onto the local RAM of the GPU, and the operations are distributed among hundreds of small cores optimized for general matrix multiplication. The training and inference operations of a CNN are batched over tens to hundreds of images at a time to minimize the data transfer overhead between the RAM and GPU core on the GPU card. However, GPUs can draw hundreds of watts of power and are inefficient for applications which require low latency.

CPUs, on the other hand, lack application specific instructions. Chip area is saved only for the most useful instructions. Single Instruction Multiple Data (SIMD) units and Multiply and Accumulate (MAC) instructions are common among processors for media processing, but the concepts can be taken further for CNNs as the CNNs are deployed in various applications. Low-power CPUs paired with custom CNN accelerating instructions can fill the void between power hungry GPUs and basic microprocessors at the expense of some chip area.

Impact on chip area can then be minimized by selecting the most useful operations corresponding to useful CNN layer types. State-of-the-art image classifying CNNs favor stacks of small 3×3 convolutions with Rectified Linear Unit (ReLU) activations and 2×2 max pooling operations [5]. New processor instructions to calculate these layers efficiently alongside a SIMD MAC instruction for fully connected layers and 1×1 convolution can cover all basic layers of a modern CNN and enable fast, low-power inference.

Significant contributions to CNN research and commercial support have been made in recent years as discussed in our Section 4 Related Work. However, there are gaps between the realms of software and hardware. Powerful image classifier architectures are beginning to be reduced in size while maintaining classifying accuracy. Moreover, compressed models are not yet well suited to widely available hardware or do not have well developed software. Until CNN research and infrastructure stabilizes, application-specific integrated circuits (ASICs) designed for CNN inference may require too much design effort and capital investment to become viable for widespread adoption. Extended instruction-based approaches in this paper have the potential to fill the gap between GPUs and fully custom ASICs with a mature hardware software codesign tools.

This paper focuses on developing a method to convert CNN architectures to be ready for custom processor instructions. The proposed framework achieves this goal by reducing the set of layer types that can be used with a CNN and then creating custom instructions to most efficiently process the reduced set of layer types. In order to prove the method of converting the CNN architecture is sound, a CNN used for SAR target classification is compared to an equivalent CNN with a reduced set of layers. The secondary objective of this work is to measure the factor of acceleration associated with the new custom instructions alongside the gate count. Computational speed increase and gate counts of the custom instructions provide a basis for the viability of the proposed study in a real application-specific instruction set processor (ASIP) application.

2. Convolutional Neural Network Architecture

2.1. Design Flow. A general flow for using the proposed framework can be formed from the concepts of reduction in layer types and custom instructions as seen in Figure 1. The flowchart for applying the acceleration framework is straightforward but requires manual adjustment at two stages. First, the reference CNN architecture is converted to use 3×3 convolutions, Rectified Linear Unit (ReLU) activations, 2×2 max pooling, and fully connected hidden layers whenever possible. Deviations from the allowed layer types are acceptable, but the deviations will incur some execution speed penalty. Then, the CNN is trained with full precision floating point numbers and CNN hyperparameters are adjusted to maximize performance. Minimal classification error, fastest execution speed, minimal memory use, or a combination of performance metrics is considered depending on the application. Ideally, this step of the framework would be performed using a high-level CNN software library with the final model description, weights, and biases as the output. Next, the CNN weights and biases are converted to fixed point for all layers. If the performance drop from fixed-point conversion is too great, the number precision can be adjusted and the CNN can be reevaluated. Finally, the fixed-point CNN can be profiled with the new custom instructions and put in use after validation.

Two stages of the proposed compression and acceleration framework are examined in this paper. In the first stage, layers of a sample CNN are converted to fit the rules of the proposed framework. The tradeoff between classifier accuracy and trainable parameters is examined after layer conversion. The layer conversion stage focuses on 3×3 convolution and 2×2 max pooling layers to demonstrate how layer conversion fits in the overall acceleration framework. In the second stage, custom CNN instructions are explained and profiled in an application-specific instruction set processor (ASIP) environment. Single-cycle and multicycle approaches to 3×3 convolution are explored with additional instructions for maximizing convolution speedup and supporting the other middle layers. Additional

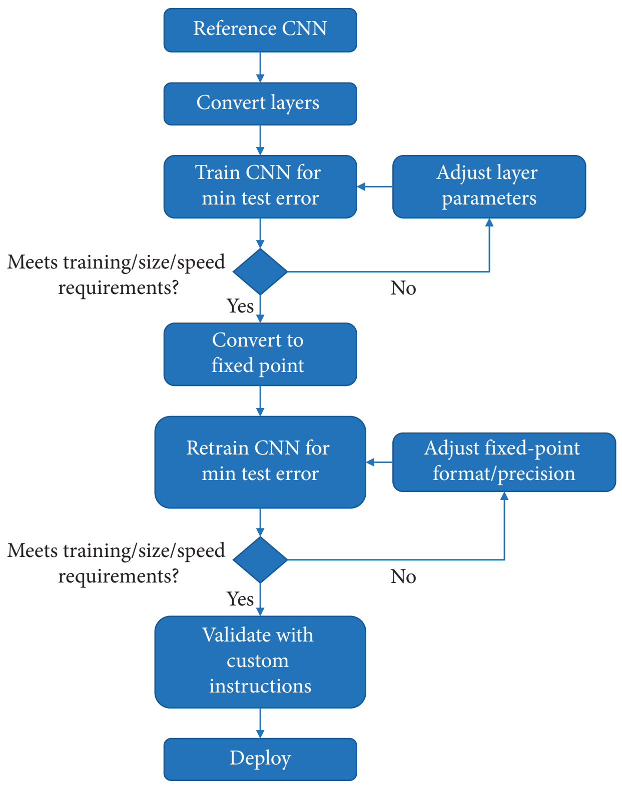
Reference CNN
Convert layers
Train CNN for min test error
Adjust layer parameters
Meets training/size/speed requirements?
No
Yes
Convert to fixed point
Retrain CNN for min test error
Adjust fixed-point format/precision
Meets training/size/speed requirements?
No
Yes
Validate with custom instructions
Deploy

Figure 1: CNN acceleration framework.

instructions are also explored for the other layers of the CNN. The additional instructions include 2×2 max pooling, ReLU activation, data reordering, and SIMD MAC instructions.

2.2. CNN Layer Conversion. Ideally, the candidate CNN already meets the format of the proposed acceleration framework. For CNN architectures that have convolution kernels larger than 3×3 or max pool layer kernels larger than 2×2, the layers can be converted for hardware acceleration. However, the conversion of convolution and max pool layers affects the overall size and performance of the CNN. For demonstration of the effects of layer conversion, SAR automatic target recognition data set is used for this study from authors' previous work and CNN model parameters are presented in Table 1 [6].

Two-dimensional convolution layers make up most layers of modern CNN architectures. The convolution kernels have odd number width and height dimensions, with 3×3 being the most common. 3×3 is the smallest kernel size that can express adjacent pixel relationships while maintaining a center pixel for output. 1×1 convolution, although technically smaller, has a different meaning in CNN frameworks and literature. With single channel data, e.g., gray-scale image data, 1×1 convolution refers to flattening a convolutional layer output to a vector for use with a fully connected layer for classification at the end layer of a network. In multichannel classifiers, e.g., RGB image data, 1×1 convolution refers to 1×1 x 3 convolution. 1×1 convolution collapses the three channels of the input data into a single channel or expands a single channel back into 3 channels. Collapsing the channels has the benefit of reducing the number of parameters in the neural network. State-of-the-art CNN architectures such as Inceptionv3 in [5] collapse the data channels with 1×1 convolution while performing 3×3 convolution, 5×5 convolution, and max pooling in parallel and then expand the outputs back out with 1×1 convolution again.

Odd dimension convolution kernels also ensure that the output of the layer output maintains even dimensions at the output. The output dimensions of a convolution layer are reduced by the width of the kernel minus one. When cascading convolution layers, the input data width and height dimensions are reduced by 2 for each 3×3 convolution layer until the desired output data dimensions are achieved. For example, 7×7 convolution layers become three 3×3 convolution layers in series.

Cascading convolutional kernels allows a larger receptive field to be covered by smaller 3×3 convolutional kernels. However, the conversion impacts the size of the neural network in terms of memory and learnable parameters. Increased memory requirements for a CNN increase the time and power penalties due to memory access. Although estimating the changes in performance of a CNN is difficult, the general effects of changing layers can be quantified for comparing CNN models. Let n be the kernel width, f the number of input feature maps, and w the number of output

Table 1: Configuration 1: CNN model summary.

Layer (type)	Original MSTAR model Kernel size	Output shape	Parameters
conv2d_1	9×9	(120, 120, 18)	1476
max_pooling2d_1	6×6	(20, 20, 18)	0
conv2d_2	5×5	(16, 16, 36)	16236
max_pooling2d_2	4×4	(4, 4, 36)	0
conv2d_3	4×4	(1, 1, 120)	69240
flatten_1 (flatten)		(1, 120)	0
dense_1 (dense)		(1, 120)	14520
dense_2 (dense)		(1, 10)	1210
		Total parameters:	**102682**

feature maps. Assuming a square convolution kernel, the convolution layer requires three words of memory to store the weights and bias term:

$$(n \times n \times f + f) \times w. \quad (1)$$

In addition to the weights, the convolutional layer requires intermediate memory for the calculation. The intermediate memory size is equal to the input data width times the height times the number of output feature maps. As the kernel size increases, the memory for weights increases roughly as the square of the width of the convolution kernel. The intermediate memory is the same size for a single layer but is repeated for the cascaded layers which increases the number of memory accesses. When the input data size is larger than that of the weights, cascading convolution layers will increase memory usage over the CNN.

The number of parameters in a CNN is associated with the complexity of the classifier. A CNN with more parameters can better fit complex classification problems. However, a complex classifier will not always generalize better than a simpler model. Increased parameter counts also require more time to train the CNN. The effects of parameter count are difficult to directly correlate to classifier performance, but significant changes to parameter in a CNN can indicate where changes to the CNN architecture may be necessary. Let i be the number of input feature maps. The number of learnable parameters for a single layer is 4. Like memory for weights, cascaded 3×3 layers will grow linearly instead of growing with the square of the kernel size as follows:

$$i \times \left(n^2 \times f + f\right). \quad (2)$$

Max pool layers can also be cascaded with some limitations. 2×2 max pool operations with stride 2 exactly cover 2^n x 2^n max pool kernel. However, kernels outside this range cannot be converted directly. Rounding to the next valid kernel size changes the dimensions of the output layer after the max pool layer and can have significant effects on the overall performance of the neural network. Rounding down will increase the number of parameters in the subsequent layers possibly degrading both speed and classifier accuracy. Rounding to larger size kernels may force a subsequent layer to vanish and force the rest of the CNN to be redesigned.

To understand the effects of converting convolutional and max pool layers of a CNN, the classifier accuracy and parameter numbers of three CNN configurations are compared. Configuration 1 is the SAR ATR CNN in its original form. It serves as a base line, and the other configurations will show how the number of parameters changes with the layer parameters. Table 1 illustrates the size and shape of Configuration 1.

Configuration 2 presented in Table 2 converts the convolutional layers to 3×3 layers but does not convert the 6×6 max pooling layer. The 6×6 max pool layer is preserved to maintain the input and output shape of each layer and roughly maintain parameter count from Configuration 1. The last convolutional layer from Configuration 1 is converted from 4×4 to 3×3. Although the 3×3 convolution does not cover the 4×4 receptive field, the next layer flattens the output and is densely connected. The result of this conversion reduces the parameter count for the 4×4 convolution layer but increases the number of parameters in the dense layer by a factor of 4. Table 2 describes the shape of Configuration 2.

Configuration 3 converts all convolutional layers to 3×3 and all max pool layers to 2×2. Configuration 3 exclusively uses layers allowed by the framework at the expense of having nearly three times more parameters than Configuration 1. Increase in the number of parameters may degrade the test classification accuracy much. Table 3 summarizes the shape of Configuration 3.

The first two configurations have a similar number of parameters. Configuration 1 uses less parameters for the first convolution layer than the first 4 convolution layers. However, Configuration 2 has less parameters in the last convolution layers than the last convolution layer in Configuration 1. Memory and classification accuracy should be similar, but Configuration 2 can now use custom 3×3 convolution instructions as part of the framework. Configuration 3 can use custom instructions for all layers but the overall test accuracy is likely to change. The acceleration factor from custom convolution instructions will be reduced by the increased number of parameters in all layers after the first max pooling layer.

3. Custom Instruction Implementation Details

Custom instructions for CNNs allow the processor to directly use the compressed fixed-point neural networks and realize inference acceleration. The downsides of power consumption and latency associated with general purpose

Table 2: Configuration 2: CNN model summary.

Keep 6×6 max pooling layer			
Layer (type)	Kernel size	Output shape	Parameters
conv2d_1	3×3	(126, 126, 18)	180
conv2d_2	3×3	(124, 124, 18)	2934
conv2d_3	3×3	(122, 122, 18)	2934
conv2d_4	3×3	(120, 120, 18)	2934
max_pooling2d_1	6×6	(20, 20, 18)	0
conv2d_5	3×3	(18, 18, 36)	5868
conv2d_6	3×3	(16, 16, 36)	11700
max_pooling2d_2	2×2	(8, 8, 36)	0
max_pooling2d_2	2×2	(4, 4, 36)	0
conv2d_7	3×3	(2, 2, 120)	39000
flatten_1 (flatten)		(1, 480)	0
dense_1 (dense)		(1, 120)	57720
dense_2 (dense)		(1, 10)	1210
		Total parameters:	**124480**

Table 3: Configuration 3: CNN model summary.

Convert all layers			
Layer (type)	Kernel size	Output shape	Parameters
conv2d_1	3×3	(126, 126, 18)	180
conv2d_2	3×3	(124, 124, 18)	2934
conv2d_3	3×3	(122, 122, 18)	2934
conv2d_4	3×3	(120, 120, 18)	2934
max_pooling2d_1	2×2	(60, 60, 18)	0
max_pooling2d_2	2×2	(30, 30, 18)	0
conv2d_5	3×3	(28, 28, 36)	5868
conv2d_6	3×3	(26, 26, 36)	11700
max_pooling2d_3	2×2	(13, 13, 36)	0
max_pooling2d_4	2×2	(6, 6, 36)	0
conv2d_7	3×3	(4, 4, 120)	39000
flatten_1 (flatten)		(1, 1920)	0
dense_1 (dense)		(1, 120)	230520
dense_2 (dense)		(1, 10)	1210
		Total parameters:	**297280**

CPUs and GPUs can be efficiently mitigated with minimal impact on overall processor design. Custom SIMD instructions in an ASIP have the potential to parallelize the most utilized functions in CNN applications. In addition, hardware software codesign is easily done due to the maturity of the tools.

3.1. Configurable Processor Limit Study. Many researchers have investigated using configurable architectures and extending the basic instruction set to speed up the execution of specific applications. In these designs, we typically start out with a parameterized processor and its basic instruction set. We then use an architectural description language to generate the datapaths or the functional units needed for the extension instructions. To complement the new hardware, we need to create the necessary software tools to complete the design cycle. Even though these tools can significantly shorten one iteration of the design cycle, many iterations are needed to find the extension candidates that result in sufficient acceleration of application. In this section, we present a limit study on the performance potential of a typical configurable processor in a battery powered environment.

We configured our reference code to process MSTAR data. We then profiled the application with the GNU gprof tool. Next, we analyzed the longest executing functions in the profile to determine which instructions to accelerate. Based on the profiling results, new instructions are created for reducing the execution time spent. In general, the new instructions can be proposed to (1) perform the same operation on multiple data items in parallel; these operations are prevalent in CNN applications and thus can provide significant performance improvements with specialized SIMD instructions, and (2) to combine instructions: when multiple operations are applied to single data item sequentially, the new instruction can combine these into one instruction.

Additionally, based on analytical performance models derived from these papers [7, 8], we decided 3×3 kernel size is the representative kernel for convolution kernels for AlexNet and VGG16 [9–11]. Custom instructions for 3×3 convolution, MAC, and 2×2 max pooling are examined, and high-level descriptions for the proposed experiments are described in the following sections. Small helper instructions for byte swapping are not included in this section but are evaluated in the results section as a potential performance enhancement.

3.2. 3 × 3 Convolution. The majority of neural network inference acceleration will come mostly from a custom SIMD instruction for 3 × 3 convolution with small bit-width weights. Convolutional layers consume the most computation time of common CNN layers, so performance gains in the convolution layers will provide the most overall benefit. The algorithm for a single 3 × 3 convolution requires 9 multiplications and 9 additions for each pixel of the input data. Then, an activation function is applied to the output of the convolution. ReLU activation is the most common in middle layers of a CNN because of the simplicity of implementation and overall classification accuracy performance. A custom instruction could perform all the multiplications, additions, and activation in a single cycle, but for flexibility, the ReLU activation will be a separate instruction. To further increase the performance of the custom instruction, we can perform this operation on multiple input fields which is executed in parallel. A 36-fold performance increase for convolutional layers is possible assuming the base processor has single data MAC operations and the effects of memory access are ignored.

4-bit weights and activation outputs in will be used in the experiments. 4-bit numbers allow a 64-bit word to hold an entire 4 × 4 input data field. Weights and biases can fill 40 bits of the same size register. A single cycle 3 × 3 convolution instruction is to be examined for 4 × 4 input data tiles where the input and output perfectly fit the instruction. The four data outputs correspond to all valid convolutions on a single 4 × 4 tile. To cover larger convolutions, the instruction on four adjacent 4 × 4 tiles will be examined. The large 8 × 8 input to 3 × 3 convolution represents how the 3 × 3 convolution instruction would be used in the more general case. The 8 × 8 input still reuses the 3 × 3 convolution at its core to achieve a speedup, but the effects of reusing output data and selecting submatrices from the original four input tiles will reduce the overall speedup. Nine 3 × 3 convolutions are required to achieve the 6 × 6 output product from an 8 × 8 input tile.

3.3. Multiply and Accumulate (MAC). The second most computationally expensive layer is the fully connected layer. The fully connected layer is essentially a vector matrix product. MAC instructions typically compute these layers for single data operations and are common in DSP processors. A 16-lane SIMD MAC instruction is to be examined to match the output of the proposed convolution and max pooling output data types. This operation has the potential for 16x cycle acceleration over single data MAC assuming one cycle for each MAC and disregarding the effects of memory access.

3.4. 2 × 2 Max Pooling. The max pooling layer is the least performance restrictive layer of CNNs, but max pooling could become a performance bottleneck once the other layers are tuned for acceleration. A 2 × 2 max pooling operation requires four memory reads, three compare operations, and one memory write for single input data. However, the data format change to 4 × 4 tiles to accommodate 3 × 3 convolution impacts the standard max pooling operation. Bit shifting and masking to extract each 4-bit input data word is inefficient. A custom SIMD max pooling instruction can perform four 2 × 2 max pooling comparisons in a single operation. A 12x performance increase is possible if only compare operations are accounted for. However, memory read and write operations account for most of operations in a max pooling routine. The number of cycles for memory operations varies dramatically between different hardware architectures, and the effects of compiler optimizations are difficult to predict.

4. Experimental Results

4.1. CNN Layer Conversion. The model training follows the same guidelines from the authors' paper with some deviations to improve performance [6]. The training data samples are minimally preprocessed. Only the names of each class label are converted from strings to one-hot encoded vectors. The weights and biases of the CNN are initialized according to what is called a Glorot or Xavier distribution in machine learning frameworks [12]. The Glorot uniform distribution is a uniform random distribution and centered around zero. The limits of the distribution are determined by the number of connections into and out of the connection as seen in

$$\text{limit} = \sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}. \tag{3}$$

The training phase uses a categorical cross-entropy loss function to determine how well the weights used during for a specific batch fit the input training data. The Adam optimizer [13] is used over standard stochastic gradient descent because Adam does not require manual tuning of the decay and momentum parameters after initial values are chosen. Training is performed for 30 epochs on 3621 images in batches of 32. The test data set of 3203 samples is used for validation directly instead of setting aside a part of the training set for validation. The CNN is trained in 10 separate sessions to vary the weight initializations and display the range of classification performance for a given model. The model with the best overall validation accuracy after the 10 training sessions is used for final accuracy evaluation. Table 4 shows the training results of the three CNN configurations.

The converted CNN configurations perform as well or better than the original CNN in this case. However, the configuration with the most parameters does not increase classifier performance likely due to overfitting and would not be ideal from the perspective of memory and execution speed. Configuration 3 demonstrates the ill effects of trying to force layers to use the custom instructions for max pooling when they do not fit the rules. Nearly 3 times as many parameters are required for the same overall

Table 4: CNN configuration accuracy summary.

Model	Min val accuracy	Avg val accuracy	Min val accuracy	Parameters
Original	0.951	0.960	0.974	102682
Keep 6×6 max pooling	0.923	0.965	0.982	124480
Convert all layers	0.932	0.955	0.973	297280

performance. Configuration 2 also has more parameters than the original but has the key benefits of the highest accuracy and ability to use the custom instructions in the following sections. The benefits of conversion to small bit-width numbers and fast SIMD instructions make up for the minor growth of the CNN model.

4.2. Custom Instructions. All custom processor instructions are designed and profiled in Cadence Xtensa Xplorer SDK. New instructions are defined using the TIE language. TIE is a Verilog-like language used to define hardware with special bindings for the Xtensa environment. TIE files are compiled and C language bindings are automatically created from the description. The SDK also provides area and gate estimates for the compiled TIE files. The processor is a Cadence LX.7 in the base configuration with a 128-bit wide data bus to accommodate wide vectorized data. Four additional 128-bit registers and ten additional 64-bit registers are included for fast register-to-register operations used with the new CNN instructions.

4.2.1. 3×3 Convolution. Two implementations of convolution are tested and profiled in this work, parallel and shared. In the fully parallel instruction, four 3×3 convolutions are performed in a single instruction. Twenty-eight separate 4-bit multipliers and 4 separate 10 input adders are required for this implementation. Figure 2 demonstrates the arrangement of the data and output products for a tiled 3×3 convolution instruction.

The fully parallel implementation requires only one cycle to obtain the four output products. The shared implementation splits the single instruction into 5 cycles to match the pipeline depth of the processor. The shared implementation uses 9 multipliers and a single 10 input adder and then multiplexes the hardware resources automatically across 5 cycles. One cycle is used for each output product, and the last cycle is used to evaluate the result. Sharing the hardware multipliers reduces the overall area for the multiplier at the expense of more time for execution. The extra circuitry required for sharing the multipliers and adders adds some extra gates so the overall reduction in gates and area is less than four times.

A general 8-bit convolution function is used for baseline comparison. 8-bit is used in the baseline function instead of 4-bit like in the custom instructions because standard C-types like int8_t or char can be used. This allows the C compiler to effectively fill the processor pipeline and provide a fair comparison. However, the 8-bit integers will be limited to 4-bits so the outputs of the custom convolution can be verified against the C model.

The perfect alignment situation for this instruction is a 4×4 input tile. Memory access and data reordering is not necessary. The parallel implementation obtained a 14.1x speedup while the shared implementation obtained an 18.5x speedup. Although the 3×3 convolution is performed in one or five cycles, memory accesses to load and store the registers for the data and kernel hide the performance decrease from sharing the multipliers.

The more complex 8×8 input tile situation is more representative of how 3×3 convolution instructions would be used in a DNN framework. The 3×3 convolution operations are performed 9 times and additional operations are required to extract data from adjacent 4×4 tiles. The convolution is accelerated 26.2 times for both the parallel and the shared instruction. The compiler optimization hides the extra cycles from the shared instruction and achieves the same speedup as the parallel instruction.

The bit shifting and masking instructions used to obtain the intermediate 4×4 tiles compose about 40% of the total cycles in the 8×8 convolution test. Adding custom reordering instructions requires minimal gates and removes the cycle penalty associated with obtaining 4×4 data tiles in-between the 4 original tiles. Additional left right tile split and top bottom tile split instructions remove the need for bit shifting and masking. Figure 3 demonstrates how the 8×8 tile is separated to achieve all the required output tiles.

Split instructions boost the speed up to 37.4 times for shared convolution and 38.3 times for parallel convolution over the pure C implementation. Interestingly, the speedup is greater than the estimated 36x speedup. This is likely because the data are reused between all operations and no intermediate memory reads or writes are performed.

4.2.2. Max Pooling. The SIMD max pooling instruction performs four separate 2×2 max pooling operations on a 4×4 input data tile stored as a 64-bit vector. The new instruction is compared to a baseline C-function that loads a data item and compares the next three data items. This sequence is performed four times in a loop to replicate the max pooling layer on a 4×4 tile. The custom instruction version reads in a 4×4 input tile and performs all 4 2×2 max pooling operations in a single cycle. It resulted in the speed increase by more than 3 times.

4.2.3. SIMD MAC. Fully connected layer evaluation can be performed with a SIMD MAC instruction. These instructions exist in various commercial processors, but the smallest data size for each SIMD lane is 8-bits. A 4-bit MAC instruction was profiled to match the bit-width of the other new instructions. Building a MAC instruction in the same processor also serves as a comparison between standard

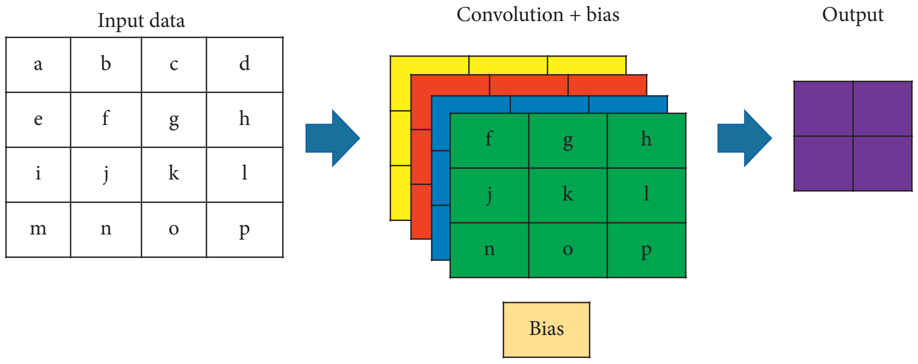

FIGURE 2: 3×3 convolution on 4×4 data tile.

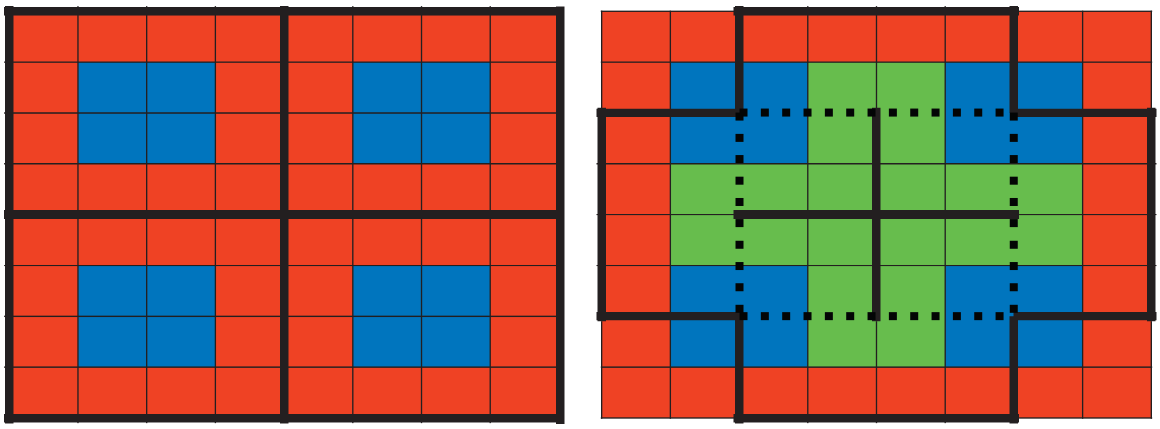

FIGURE 3: Demonstration of the split instruction. (Left) blue squares are outputs from adjacent input tiles with thick black borders. (Right) green outputs after obtaining intermediate tiles from 4 original tiles.

instructions and the proposed instructions specialized for CNN in terms of area and power.

Evaluation of the new 4-bit MAC instruction was performed using a 16×16 matrix and 16 element vectors to simulate a fully connected layer with 16 inputs and 16 outputs. The baseline C-function loads the accumulator with the bias then performs multiplication of each element of the vector with each element of a row of the input data matrix. Loading the bias and element-by-element multiplication is repeated for each row of the matrix. This baseline requires 4096 MAC instructions in total. The overall speedup, 27.5x for shared and 29.8 for full parallel, is significantly larger than the expected 16x because of the reduced cycle penalty on the memory accesses. The speedup in this situation can be extrapolated to larger matrix vector multiplications for cycle time estimates. This instruction is least efficient for vector and matrix sizes with dimensions of $16n + 1$ so using multiples of 16 for hidden layers is recommended in practice.

4.2.4. Custom Instruction Summary and Discussion. Table 5 summarizes the speedup and gate count impact of the custom CNN instructions. In general, the shared instructions use roughly half the gates compared to the fully parallel implementation with little impact on the overall speedup. The exception is the shared version of the 16 lane MAC instruction because all the instructions were implemented on the same design. The automatic placement and routing tools use more gates to meet timing requirements of the overall design. Creating separate designs for each instruction would reduce the number of gates for each operation.

The overall overhead increase to the chip size or gate count when all layers are considered is relatively small. Whether we use shared or full connected 4 by 4, or 8 by 8 depending on the custom instructions for convolution and dense layer, less than 4% more gates would be added to the Tensilica core. Tensilica core is 156k gate count small RISC core which can be fabricated in 0.13 micron standard cell technology process. The silicon density of 100k logic gates per mm^2 is achieved. A low-cost chip can carry 5 million logic gates in a single chip using the same process technology. In our analysis, we assume that the processing of 15 to 30 frames per second achieves real-time CNN inference realization because IoT edge device is used in the low bit rate mobile environment. Therefore, our approach using a processor-based CNN realization will help gain the amount of speedup crucial determining feasibility of this type of CNN system.

We assumed that the power consumption of the CNN custom block roughly depends on the area or chip size which

Table 5: Speedup and gate summary for custom SIMD instructions.

4 x 4 tile, 3 x 3 conv	**Cycles**	**Speedup**	**Gates**
Baseline	408	—	156466
Shared	29	14.1	2389
Full	22	18.5	4456
8 x 8 tile, 3 x 3 conv	**Cycles**	**Speedup**	**Gates**
Baseline	3404	—	156466
Shared	130	26.2	2389
Full	130	26.2	4456
Shared + splits	91	37.4	2405
Full + splits	88	38.7	4472
Max pooling	**Cycles**	**Speedup**	**Gates**
Baseline	44	—	156466
Tie	13	3.4	262
FC16	**Cycles**	**Speedup**	**Gates**
Baseline	2114	—	156466
Shared	77	27.5	4124
Full	71	29.8	2970

is connected, based on the high-level power estimation literature [14]. We assume that the speedup out of our custom instructions offer more design choices for HW designers. HW designers and architects can explore and decide on their options with confidence based on their insight, with the help of our methodology's specific CNN implementation choices.

It would be preferable to execute the proposed custom instructions on different processors other than Tensilica SDK with those custom instructions implemented. Due to its limited availability on processor core internals, we did not perform fixed-point analysis on finite word effects for our 4-bit SIMD instructions. However, we think that this has minimal impact on our classification performance, based on the literature [15, 16].

Tensilica is based on a 32-bit RISC processor architecture, so we believe those candidate instructions are applicable to other architecture as well. We plan to prototype on a hardware emulation board with multiple workloads so that we can validate our proposed instructions.

We concluded that our methodology requires less development time and less design effort than conventional methodologies. We assessed our design time including model building, simulation, and verification time. In most cases, verification takes a significant portion of development time if it is a conventional simulation-based verification. The ASIP-based CNN method is easily adapted to the existing processor-based design flow, leading to a shorter development time for generating performance parameters.

5. Related Work

Four general classes of hardware are used for accelerating neural network inference: high-power CPUs, low-power embedded microprocessors, FPGAs, and GPUs. CPUs are the most general processors and can run all neural network models. However, general purpose comes at the cost of being fast or efficient for a single kind of computation. Most of the computation time required to run a neural network is spent executing multiplication and addition operations. Fortunately, these simple operations are useful enough in general programs that most modern CPUs have special hardware to execute multiple multiply and add instructions simultaneously. Intel's latest CPUs utilize a wide 512-bit register that equates 16 single-precision floating point numbers to execute in a single instruction in parallel [17]. Embedded processors are usually SoCs. In the industry, ARM processors are the core of choice for many embedded SoCs because the ARM architecture is designed for low-power applications. ARM-based embedded SoCs are becoming increasingly powerful with multiple cores and small SIMD units designed to accelerate the most common tasks. Although embedded SoCs are not as powerful as large CPUs or GPUs, they are much more efficient and can operate with low latency at a fraction of the cost of other platforms. The benefits of small efficient embedded processors drive the need for neural network compression and acceleration.

GPUs operate using a completely different paradigm compared to CPUs favoring parallelism over general compute ability. It contains hundreds of cores running in parallel with a grid such as interconnect architecture. This specialized, massively parallel architecture perfectly suits neural network computation with some caveats. State-of-the-art consumer graphics cards such as the Nvidia V100 are capable of 14 teraflops (trillions of floating point operations per second) with single-precision floating point [18]. However, the GPU platform has some major downsides. The large, fast GPUs used for neural network research require hundreds of watts of electrical power to fully utilize the hardware. For example, the V100 draws up to 250 watts at peak consumption [18]. In addition, powerful GPUs run as coprocessors and need to be fed data from a general computing system. PCIe is the most popular interface for GPUs and although PCIe is high bandwidth, it is also high latency and difficult to integrate into embedded systems. Latency is unacceptable in many embedded applications ruling GPUs out as a processing platform. In cases where the CPU load is high and the GPU needs to be sent data for processing, data transfer time can approach hundreds of milliseconds depending on the system [19].

Image classification networks such as AlexNet are proven to classify the 1000 class dataset called ImageNet with high accuracy. The disadvantage of the AlexNet and other popular architectures are that they require hundreds of megabytes to gigabytes of RAM to store the parameters [20]. In many cases, the best option is to start with a small neural network to minimize the memory footprint in the system. However, large complex models are unavoidable for more complicated tasks. As a result, the large footprint of complex neural networks will not fit into memory in small embedded systems without dynamic memory. If dynamic memory is available, memory accesses are orders of magnitude slower and higher power consumption than access to on-chip ram.

Architectures such as SqueezeNet are a response to large CNNs such as AlexNet. SqueezeNet reduces the memory with three strategies, reducing convolutional filter size, decreasing the number of input channels into filters, and downsampling later in the network [20]. SqueezeNet is an interesting design space exploration of architectures for image classification but is specialized for three channel data such as RGB images. An alternative to SqueezeNet is MobileNets. MobileNets reduces the number of computations for a neural network by using separable convolutions [21].

Converting the matrices of neural network layers into the frequency domain with FFT and multiplying reduces the number of mathematical operations for the same convolution operation [22]. Reducing the number of mathematical operations for convolution will improve performance across all convolutional layers of a neural network and improve overall network performance. The performance increase does come at the cost of memory footprint. Convolution kernel storage is dictated by the largest kernel in the network since the filters can be shared across all layers, but these kernels still use more memory than traditional weights. Due to the lack of memory in many embedded systems, this method will likely not be feasible and will not be focused on in the framework for DNN compression.

Neural networks are typically trained using a 32-bit floating point model to effectively backpropagate errors to the next layer. Since the total network size can be estimated by the number of parameters times the number of bits per parameter, reducing the model to an 8-bit floating point model immediately reduces the memory used for storing the parameters to one-quarter of the original size [23]. CNN classifier performance is slightly reduced by reductions in bit-width representation during inference and in some cases smaller parameter representation can even act as regularization technique to increase test accuracy performance.

In addition to reducing bit-widths of numbers, the number format can also be dynamically adjusted to further mitigate loss from quantization. The Ristretto method explores a simple method to optimally adjust to the dynamic range of each layer in a network using a simple rule [24]. The length of the integer part of a fixed-point number is made large enough to avoid being saturated, and the remaining bits are devoted to the fractional part of the number. This allows for further bit-width reduction to as low as 4-bits per parameter with 1% loss in classification accuracy.

In the most extreme case of quantization, weights can be represented as 1-bit numbers. 1-bit weights in neural networks maximize the benefits from simpler hardware and small memory footprint. In addition, multiplication or bit shifting can be replaced by binary operations. Remarkably, the classification error rates do not drop significantly for some applications. In [4], the authors experiment with the binary weight and conclude binary CNNs perform nearly as well as full precision CNNs but accelerate convolution by a factor of 58. Binary neural network benefits are a promising for applications where some classification accuracy loss is acceptable for fast and low-power inference.

Compression algorithms exist across a wide variety of file types and utilize the unique characteristics of the target data to achieve the smallest file size. The deep compression method follows suit by exploiting the shape and distribution of the stored parameters. Deep compression achieves between 35x and 49x reduction in memory footprint for popular image classification architectures [25]. Converting to sparse matrix formats provides most of the compression benefit and can be further tuned depending on the specific sparse matrix format [26]. Sparse matrix formats are only effective in the case of neural networks because the sparsity of the matrices representing the weights in each layer is greater than the additional storage space from tuple representation.

There are several custom frameworks for special type of CNNs in the literature. The authors in [10] present an efficient HW/SW implementation of sparse convolutional neural networks. Using their approach, the authors designed an accelerator attached to AXI processor bus. Processing units (PUs) are designed by inspecting CNN kernels. The paper shows 2.93x better performance over previous FPGA-based accelerators.

In addition to those approaches, there are several CNNs implemented using FPGAs or ASICs available presently [9, 15, 27]. OpenCL-based design methodology is presented in the paper. In this work, the authors wanted to exploit the parallelism capabilities of the OpenCL. They have used a standalone method to implement a hardware accelerator. The host and kernel programs are compiled to link together as a single binary file in the standalone method. Thus, standalone application specific processor includes host, compute device, and compute unit. In the host/device method, the host code controls the ASP's execution implemented as a kernel program. In addition, the authors have used Transport Triggered Architecture (TTA) as an ASP. TTA consists of Function Units (FUs) and Register Files (RFs). The FUs communicate with the RF via datapath. The RF stores the operands, whereas FUs contain the custom operations, i.e., load, store, the arithmetic unit, and logic unit. The TTA's instructions are implemented as separate work-items grouped in the work-groups. The instructions from the couple of work-items can be executed in parallel. The authors have used barriers to chain the work-items together in a work-group. When one work-item reaches the barrier, it has to wait for other work-items in the work-group before continuing the execution. The authors did not compare their results with existing results. They have not reported the actual hardware and software execution times. The power consumption and resource utilization is also not been reported. One of reasons we implement our SIMDy CNN on Tensilica core is to provide a realistic measure for hardware-based implementation overhead. Tensorflow cores as well as specialized FPGA-based CNNs are reported in the literature [28].

There are efficient SIMD library implementations developed for CNN such as [11, 29]. These are SIMD intrinsic instructions offered by traditional processors including Intel® AVX512 [30]. However, the baseline processor should be somewhat powerful enough to have dedicated floating point units such as Neon or Helium blocks [29]. In our work, we are interested in extending embedded processor using common instruction set so that we can significantly accelerate the overall performance, especially for convolutional neural network inference for low-power embedded IoT nodes. Therefore, we focused on custom instructions for CNN. Often, the dedicated CNN accelerators were implemented on PCI-type FPGAs with several hundreds of watts power consumption.

In this section, we looked into the relevant prior state-of-the-art tools for CNN implementations on GPU, embedded processors, and SIMD extensions.

6. Conclusions

This work describes the compression and acceleration framework for the convolutional neural network (CNN) and its processor-based implementation with instruction extensions for embedded processing, by offering design options:

(i) Overall CNN design flow for processor-based implementations
(ii) Fixed-point custom CNN instructions
(iii) CNN layer compression and conversion case studies

Specifically, features such as 3×3 convolution and 2×2 max pooling layers are used to convert CNNs to accommodate new candidate instructions. After the CNN is converted, custom CNN instructions were proposed to accelerate CNN training and inference. Introducing 3×3 convolution, ReLU, max pooling, and MAC instructions result in significant speedups with minimal impact on chip area. Tensilica is based on a 32-bit RISC processor architecture, and thus, those candidate instructions can be applied to other architectures. Additionally, 32-bit floating point CNNs were converted to fixed point for inference. The results show the advantage of our custom instruction-based CNN implementation, leading to less design time with a significant speedup for the low-power embedded system.

Abbreviation

GPU:	Graphic processor unit
ASIP:	Application specific instruction set processor
FCCM:	Field-programmable custom computing machine
FPGA:	Field-programmable gate array
HPC:	High-performance computing
FCCM:	Field-programmable custom computing machine
ALU:	Arithmetic logic unit
DDR:	Double data rate
SRAM:	Static random access memory
BRAM:	Block random access memory.

Data Availability

The datasets generated during the current research are not publicly available due to funding requirements but are available from the corresponding author on reasonable request.

Conflicts of Interest

The authors declare no conflicts of interest.

Authors' Contributions

This work was done by Joshua Misko and Young Soo Kim. Young Soo Kim developed the idea throughout this research project and wrote a convolutional neural network reference implementation. Joshua Misko implemented and tested the proposed system based on their idea while writing his paper. Shrikant Jadhav helped to write this paper and designed the paper structure during writing, simulation, and its experimental result analysis. As a corresponding author, Young Soo Kim managed this project during the project year and advised the direction of this work while reviewing the paper.

Acknowledgments

The authors would like to thank Dr. In Soo Ahn for constructive criticism of the earlier version of the manuscript. This work was supported in part by the Bradley University Provost Office through 2019-2021 Caterpillar Award and Teaching Excellence Award.

References

[1] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," 2016, http://arxiv.org/abs/1605.07678.
[2] S. Han, X. Liu, H. Mao et al., "EIE: efficient inference engine on compressed deep neural network," 2016, https://arxiv.org/abs/1602.01528.
[3] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of the 33rd International Conference on Machine Learning*, New York, NY, USA, June 2016.
[4] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-net: imagenet classification using binary convolutional neural networks," 2016, https://arxiv.org/abs/1603.05279.
[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015, https://arxiv.org/abs/1512.00567.

[6] J. Misko, Y. Kim, C. Qi, and B. Sirkeci, "Mobile high-performance computing (HPC) for synthetic aperture radar signal processing," *Proceedings of SPIE*, vol. 10647, 2018.

[7] L. Jünger, N. Zurstraßen, T. Kogel, H. Keding, and R. Leupers, "AMAIX: a generic analytical model for deep learning accelerators," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach, Eds., Springer, Cham, Switzerland, 2020.

[8] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Performance modeling for CNN inference accelerators on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 843–856, 2020.

[9] D. Manatunga, H. Kim, and S. Mukhopadhyay, "SP-CNN: a scalable and programmable CNN-based accelerator," *IEEE Micro*, vol. 35, no. 5, pp. 42–50, 2015.

[10] W. You and C. Wu, "RSNN: a software/hardware Co-optimized framework for sparse convolutional neural networks on FPGAs," *IEEE Access*, vol. 9, pp. 949–960, 2021.

[11] S.-J. Lee, S.-S. Park, and K.-S. Chung, "Efficient SIMD implementation for accelerating convolutional neural network," in *Proceedings of the 4th International Conference on Communication and Information Processing (ICCIP '18)*, pp. 174–179, New York, NY, USA, November 2018.

[12] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Chia Laguna Resort, Sardinia, Italy, May 2010.

[13] D. P. Kingma and B. J. Adam, "A method for stochastic optimization," 2014, http://arxiv.org/abs/1412.6980.

[14] G. C. Cardarilli, L. D. Nunzio, R. Fazzolari, M. Re, F. Silvestri, and S. Spanò, "Energy consumption saving in embedded microprocessors using hardware accelerators," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 16, no. 3, pp. 1019–1026, 2018.

[15] X. Chen, X. Hu, H. Zhou, and N. Xu, "FxpNet: training a deep convolutional neural network in fixed-point representation," in *Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2494–2501, Anchorage, AK, USA, May 2017.

[16] S. Anwar, K. Hwang, and W. Sung, "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)*, pp. 1131–1135, South Brisbane, QLD, Australia, April 2015.

[17] Intel AVX-512 Instructions, 2017. Available:, https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[18] NVIDIA TESLA V100, 2017. Available:, https://www.nvidia.com/en-us/data-center/tesla-v100/.

[19] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for GPU computing," in *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, Seoul, South Korea, December 2013.

[20] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, and W. J. Dally, "SqueezeNet: alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size," 2015, https://arxiv.org/abs/1602.07360.

[21] A. G. Howard, M. Zhu, B. Chen et al., "Efficient convolutional neural networks for mobile vision," 2017, https://arxiv.org/abs/1704.04861.

[22] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," 2013, https://arxiv.org/abs/1312.5851.

[23] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, June 2016.

[24] P. G. Ristretto, "Hardware-oriented approximation of convolutional neural networks," 2016, http://arxiv.org/abs/1605.06402.

[25] S. Han, H. Mao, and W. J. Dally, "Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding," 2016, http://arxiv.org/abs/1510.00149.

[26] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy, "Sparse convolutional neural networks," in *Proceedings of the 2015 Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, June 2015.

[27] M. Y. Lee, "Trends in AI processor technology," *Electronics and Telecommunications Trends*, vol. 35, pp. 66–75, 2020.

[28] N. P. Jouppi, A. Borchers, R. Boyle et al., "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, pp. 1–12, Toronto, Canada, June 2017.

[29] NXP eIQ for TensorFlow Lite, eIQTensorFlowLite, 2021, https://www.nxp.com/design/software/development-software/eiq-ml-development-environment/eiq-for-tensorflow-lite.

[30] *Intel*, AVX512 Deep Learning Boost, 29 12 2020, https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-vector-extensions-512-intel-avx-512-new-vector-neural-network-instruction.html.