Journal of
Low Power Electronics and Applications

MDPI

Article

InSight: An FPGA-Based Neuromorphic Computing System for Deep Neural Networks †

Taeyang Hong, Yongshin Kang and Jaeyong Chung *

Department of Electronic Engineering, Incheon National University, Incheon 22012, Korea; hty9494@inu.ac.kr (T.H.); kyscall5180@gmail.com (Y.K.)

* Correspondence: jychung@inu.ac.kr

† Our system, InSight, is named as a concatenation of "In" from "In"cheon National University and "Sight" representing its ability to see.

Received: 28 September 2020; Accepted: 27 October 2020; Published: 30 October 2020

Abstract: Deep neural networks have demonstrated impressive results in various cognitive tasks such as object detection and image classification. This paper describes a neuromorphic computing system that is designed from the ground up for energy-efficient evaluation of deep neural networks. The computing system consists of a non-conventional compiler, a neuromorphic hardware architecture, and a space-efficient microarchitecture that leverages existing integrated circuit design methodologies. The compiler takes a trained, feedforward network as input, compresses the weights linearly, and generates a time delay neural network reducing the number of connections significantly. The connections and units in the simplified network are mapped to silicon synapses and neurons. We demonstrate an implementation of the neuromorphic computing system based on a field-programmable gate array that performs image classification on the hand-wirtten 0 to 9 digits MNIST dataset with 99.37% accuracy consuming only 93uJ per image. For image classification on the colour images in 10 classes CIFAR-10 dataset, it achieves 83.43% accuracy at more than 11× higher energy-efficiency compared to a recent field-programmable gate array (FPGA)-based accelerator.

Keywords: deep learning; deep neural networks; efficient deep learning; neuromorphic computing system

1. Introduction

Deep convolutional neural networks (CNNs) have shown state-of-the-art results on various tasks in computer vision, and their performance has become comparable to humans in some specific applications [1]. However, they contain a huge number of weight parameters (e.g., 10^8, [2]), and the inference by the models is computationally expensive. This makes it problematic to deploy these models to embedded platforms, where computing power, memory, storage, and energy are limited. It is also problematic to evaluate the large models at the server side. For example, processing images and videos uploaded by millions of users require massive amounts of computation, and running a data center that supports such computation carries enormous costs including cooling expenses and electricity bills.

To cope with these issues, there has been an enormous amount of research efforts put into CNN acceleration hardware such as GPUs [3], field-programmable gate arrays (FPGAs) [4,5] and ASICs [6,7] very recently. Traditionally, the processing elements (PEs) of hardware accelerators are complex and large in area, and the design of the accelerators has focused on maximizing the utilization of a small number of the PEs considering the limited external memory bandwidth. Most development of CNN accelerators has also been in that way. At some performance point, most accelerators based on the Von

Neumann architecture become memory bound irrespective of the type and the number of available compute units. To execute CNNs, it is required to fetch a number of weight parameters. In addition, it is required to write a large amount of intermediate data to dynamic random-access memory (DRAM)s and read it back so the performance of CNNs accelerators are also often limited by the off-chip bandwidth. Thus, there have been attempts to reduce the memory access [6]. This traditional approach is practical and can be applicable today.

However, if we have numerous PEs as our brains do, we may need a radically different architecture from the traditional one. Processing elements over several thousands are not considered practical today, but it is becoming so. It is now well-known that 8-bit fixed point arithmetic instead of 32-bit floating point is sufficient to run CNNs without some loss in accuracy [8]. The multipliers on CNN accelerators can be replaced by barrel shifters [9]. In addition, the device community actively performs research on neuromorphic devices such as memristors [10]. Thus, a new computer architecture to exercise millions of PEs needs to be developed for the near future. In such a system, operations can be simply mapped into PEs rather than being scheduled, and the dataflow architecture can be a baseline. Recent neuromorphic architectures are aligned with this direction, although some of them such as BrainScaleS [11] and Neurogrid [12] are designed for brain simulation. TrueNorth [13] aims at both real-world applications and brain simulation [14] and equips with 256 million synapses, which not only store the synaptic weights but also serve 1-bit compute units. Recent neuromorphic systems employ detailed neural models such as the leaky integrated-and-fire model [15], in part because they should be used for brain simulation. However, the recent success of deep learning tells us that the detailed model may not be necessary for a high predictive performance for today's applications.

This paper presents a novel neuromorphic computing system that is designed solely for the execution (i.e., inference or prediction) of deep neural networks. Since our neuromorphic system is not made for brain simulation, we employ the perceptron as the neural model. Even if we do not employ a detailed neural model at the neuron (circuit) level, and we adopt FPGAs as the backend, we are inspired by our brain at the architectural level and our system is fundamentally closer to the neuromorphic systems than the traditional accelerators in the sense that it is designed for many, small processing elements. Thus, we will call our system the neuromorphic computing system.

The contributions of our neuromorphic computing system are summarized as follows:

- We implement a complete, fully-functional, non-Von Neumann system that can execute modern deep CNNs and compare it with various existing computing systems including existing neuromorphic systems, FPGAs, GPUs, and CPUs. This reveals that the neuromorphic approach is worth to explore despite the progress of the conventional specialized systems.
- The dataflow architecture enabled by the one-to-one mapping between operations and compute units has the fundamental scalability issue, although it does not require any array-type memory access. However, this work increases the capacity by adopting model compression and word-serial structures for 2D convolution.
- We demonstrate that neural networks can be implemented in the neuromorphic fashion efficiently without the crossbars for synapses. This is possible because we can convert dense neural networks into sparse neural networks.

The rest of the paper is organized as follows. Section 2 introduces neural networks and neuromorphic systems. Section 3 explains the software part of the system and Section 4 discusses the hardware part. Section 5 shows experimental results, and Section 6 concludes the paper.

2. Background

2.1. A Neuron Model

The neuron model commonly used in the machine learning community is called the perceptron. In this basic model, a neuron (a.k.a., unit) performs the computation

$$y = f(\sum_{i}^{N} w_i x_i + b) \tag{1}$$

where y is the *activation* of the neuron, N is the number of inputs, x_i is the input activation, w_i is the weight, b is the bias, and f is the activation function. This model is loosely connected to biological neurons and synapses. The weight represents a synaptic strength. The activation represents the firing rate of a neuron. The product and the sum are associate with the post-synaptic current and the membrane potential, respectively. The bias is associated with a threshold value, above which a neuron starts firing. Although this model is coarse and highly abstracted, it provides the best predictive performance in practical machine learning applications.

2.2. Neural Networks

A neural network is a composition of perceptrons. Feedforward neural networks are the commonest type of neural networks in real-world applications, and neurons are organized into distinct layers using the topological order. The first layer is the input layer, the last layer is the output layer, and the other layers are the *hidden layers*. If a feedforward network has more than one hidden layer, we call it the *deep* neural network. The most common layer type is the fully-connected layer where neurons between two adjacent layers are fully pairwise connected.

A layer of M neurons fully connected to N neurons performs a non-linear transformation. Let $x \in \mathbb{R}^N$ and $y \in \mathbb{R}^M$ be the input and the output of the layer, respectively. Then, the non-linear transformation can be represented by

$$y = f(xW + b) \tag{2}$$

where $W \in \mathbb{R}^{N \times M}$ is the weight matrix, $b \in \mathbb{R}^M$ is the bias, and f is the activation function. Figure 1 depicts a directed acyclic graph that represents a feedforward neural network of 3 layers. The circles (lines) represent neurons (connections). The hidden and output layers are fully-connected.

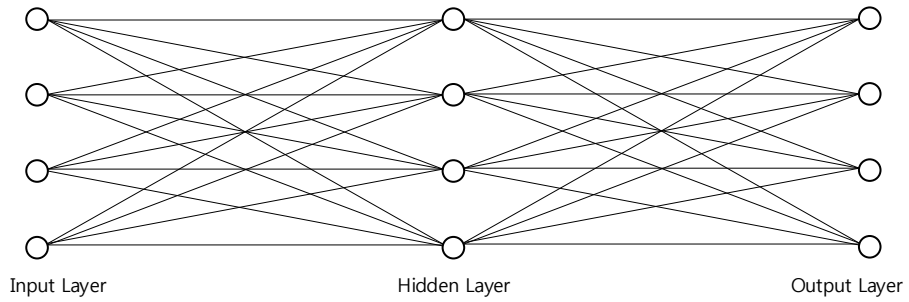


Figure 1. A feedforward network of 3 layers.

2.3. Convolutional Neural Networks

Convolutional neural networks (CNNs) are multi-layer neural networks. They are commonly used for visual recognition tasks and take an image as input. CNNs have made breakthroughs in various image recognition tasks, and have fired recent enthusiasm for deep learning. A CNN is made up of two main types of layers: convolutional layers and fully-connected layers. Neurons in a convolutional layer are arranged in a three-dimensional space. The dimensions are height and width (the spatial dimensions), and depth. For example, input neurons activated by the pixels of each channel of an input image can be laid out according to vertical and horizontal positions of the pixels and the channels. Each convolutional layer of a CNN transforms the input activations, represented by a 3D volume, into a 3D output volume. The activations at the same depth form an *activation map* (or, a channel). Consider a convolutional layer that takes N activation maps and produces M activation maps. The height and the width of the input (output) maps are denoted by H (H') and

W (W'). This may result in $(HWN) \times (H'W'M)$ weights, but neurons in convolutional layers have spatially local connectivity and neurons at the same depth share the weights for the local connections. The spatial extent of this local connectivity is called the receptive field of the neuron. Let K_h and K_w denote the height and the width of the receptive field, respectively. Thus, the weights in the convolutional layer can be described as a 4-th order tensor $W \in \mathbb{R}^{K_h \times K_w \times N \times M}$. Note that the sizes of the first and the second dimensions are K_h and K_w instead of H and W, respectively, due to the local connectivity. The convolutional layer is depicted in Figure 2.

Let $x_k \in \mathbb{R}^{H \times W}$ be the k-th input activation map and $y_l \in \mathbb{R}^{H' \times W'}$ be the l-th output activation map. Then, the l-th output activation map equals the elementwise sum of $x_l * W_{k,l}$ for $k = 1, \ldots, N$ where $*$ denotes the 2D convolution and $W_{k,l}(i,j) = W(i,j,k,l)$ for $i = 1, \ldots, H, j = 1, \ldots, W$. Thus, it requires $N \times M$ 2D convolutions to compute the whole output activation volume.

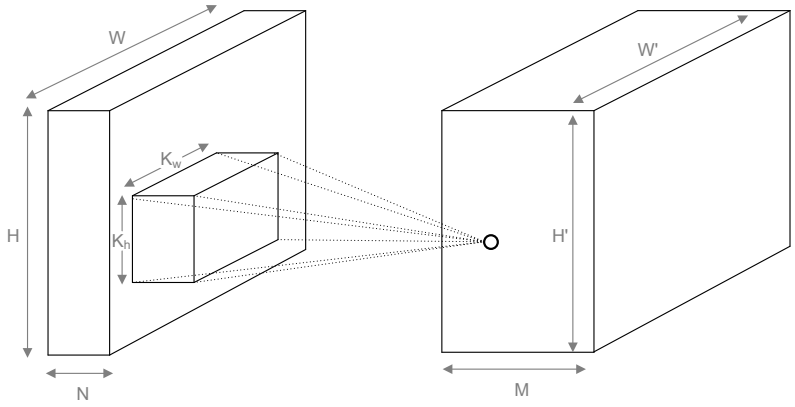


Figure 2. Neurons in convolutional layers are arranged in a 3D space and have spatially local connections.

2.4. Neuromorphic Architectures

In recent times, the term neuromorphic is used broadly to describe hardware and software systems that implement models of neural systems. However, in this paper, we use the term to describe a specific type of hardware architectures. The neuromorphic hardware is capable of evaluating a neural network and consists of neurons and synapses, which are the processing elements of the hardware. A synapse stores a weight parameter and performs synaptic operations. Each unit and each connection in the model are mapped onto a neuron and a synapse, respectively. Figure 3 illustrates the neuromorphic architecture and Von Neumann architecture (More precisely, stored-program computers).

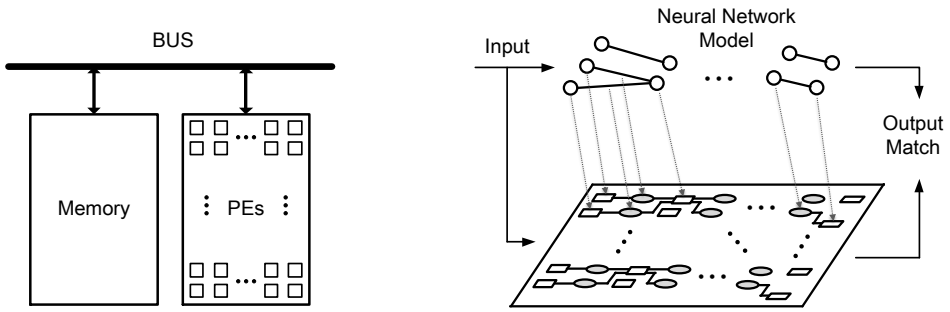


Figure 3. In Von Neumann architecture, the input/output(I/O) operations between the memory and the processor lead to processing bottlenecks and significant energy consumption. In neuromorphic architectures, neural network models are considered the software and mapped onto hardware neurons and synapses.

In Von Neumann architecture, the weight parameters are stored in the off-chip memory and the intermediate results of computation are written to the memory and read it back. In the neuromorphic architecture, data flow from input to output continuously. In the traditional computers, a small number of processing elements are time-multiplexed, whereas in the neuromorphic architecture, synaptic operations associated with a connection are dedicated to the corresponding synapse. Thus, it requires at least as many synapses as the number of connections. TrueNorth [16] belongs to this neuromorphic architecture.

3. Neural Network Compiler

3.1. Model Compression (Simplification)

Practical deep neural networks have a huge number of weight parameters. For example, VGG-16 [17] has 130 million parameters and if each parameter is represented in the single-precision floating-point format, we need a memory of 520 MBytes to execute the network. Considering a high-end microprocessor for servers equips with a large cache memory of at most several tens of megabytes, this amount of data cannot be stored on-chip with today's complementary metal–oxide–semiconducto(CMOS) technology. Thus, we have taken it for granted to store those parameters in DRAMs. However, model compression techniques, which reduce the number of parameters [18] or bits per parameter [19], have changed the situation. Based on these techniques, a previous work tries to store them in on-chip static random-access memory (SRAM)s [6]. We take one step further and will store them in registers since the word-serial access of array-type memories, including on-chip SRAMs, is a potential limiting factor to both performance and energy-efficiency. This might be a wild and crazy idea. However, it might not be much so. In addition to the model compression techniques, there are several other reasons why we believe the idea may not be that bad. First, most of the large number of parameters are in the first fully connected layer, but it does not seem to be important. For example, GoogleNet [20] uses average pooling instead of fully-connected layers at the top of convolutional layers, eliminating a large number of parameters and it only has 4 million parameters. Second, while we will demonstrate that a network is executed end-to-end by the proposed approach, we can apply the proposed approach to a few early convolution layers, which are usually most computation-intensive but have the fewest parameters among layers. Third, neuromorphic memory devices such as memristors are being developed actively in the device community. Therefore, we will assume that weight parameters can be stored in registers and for now, we will increase the capacity of our system by employing model compression.

We preprocess a neural network to be executed in our neuromorphic hardware by a program called the neural network compiler (NNC). It takes a feedforward neural network and generates a neural net with a reduced number of parameters. To reduce the number of parameters, we combine matrix (tensor) factorization and pruning using the method of [21] for fully connected layers (convolutional layers). An example for a fully-connected layer is shown in Figure 4. The matrix (tensor) factorization allows us to remove the redundancy in weight parameters. Pruning zeros out parameters of small magnitude and convert a densely connected neural net into a sparsely connected one. Pruning is performed not only to reduce the number of parameters, but also the sparsity of weights is essential in our approach, which will be shown later. Most existing neuromorphic computing systems are designed for densely connected nets, while we consider the weight sparsity in the first place when designing our system.

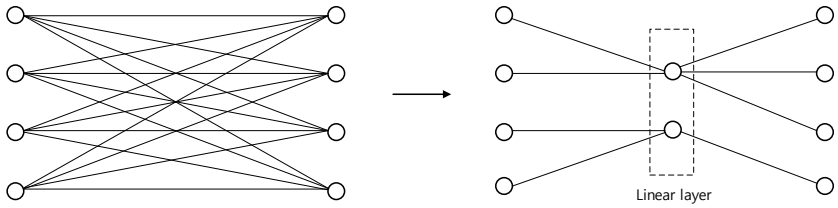


Figure 4. Fully-connected layers are simplified by matrix factorization and pruning.

3.2. Time Delay Neural Net Conversion

Convolutional neural networks require a number of two-dimensional (2D) convolution operations. In neuromorphic architectures, the 2D convolution with a $K \times K$ kernel for an image of $W \times H$ pixels is commonly reduced to a matrix–vector multiplication by unrolling the kernel. The matrix is roughly of the size $WH \times WH$ and approximately has WHK^2 non-zero elements. This matrix–vector multiplication is assumed to be performed in the fully parallel manner in most neuromorphic architectures, and requires a huge number of synapses even if the neuromorphic architectures exploit the weight sparsity well. The huge number of synapses still becomes a burden even if very low-cost processing elements are available. In addition, the input activations should come from outside the chip or come from a processing core dedicated to a previous layer. However, the bandwidth of both off-chip I/O and on-chip buses may not be sufficient to feed the whole activation map in a time step. Thus, in spite of the high theoretical throughput of the fully-parallel structure, the actual throughput can be degraded.

To implement convolutional layers efficiently in the neuromorphic fashion, the NNC converts each convolutional layer into a time delay neural net (TDNN). This TDNN conversion is inspired by the conventional 2D convolution hardware [22]. We serialize a two-dimensional activation map (e.g., an image) into a data stream, and the activations (e.g., pixels) of the map are fed line by line, from top to bottom. From the data stream, the input activations that are needed to produce an output activation (i.e., the activations in the convolution window) are collected by a series of delay units. After the first $K_h - 1$ lines and the first K_w activations of the K_hth line are fed to the TDNN, the first output is available. From that moment on, each new activation fed into the chain of delay units displaces the convolution window to the next adjacent position until the whole input has been processed. The approach based on matrix–vector multiplication requires HW neurons and $WH \times WH$ synapses in the worst case, while the TDNN has 1 neuron, $W(K_h - 1) + K_h(K_w - 1)$ delay units, and $K_h K_w$ synapses. Figure 5 shows an example of the TDNN conversion when $K_h = K_w = 3$.

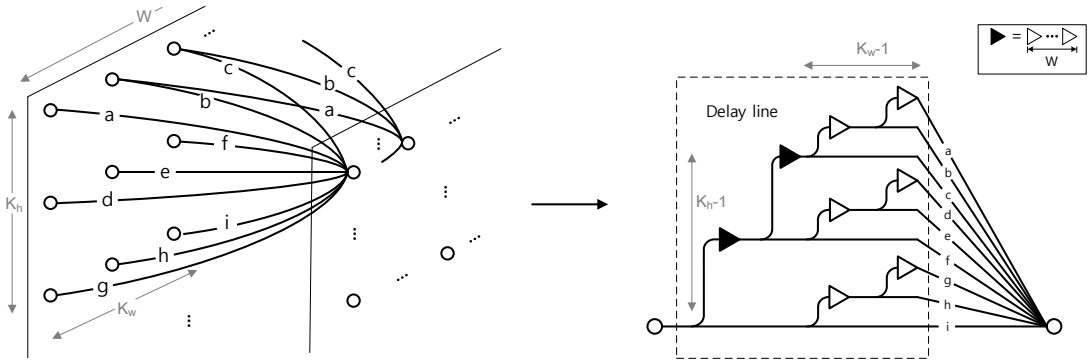


Figure 5. Convolutional layers are converted into time delay neural nets (TDNNs).

Despite the activation-serial processing of the TDNN, our neuromorphic system will have a much higher performance if it is compared to the conventional accelerators. In the neuromorphic

architectures, each TDNN corresponding to a convolutional layer is mapped to a set of independent hardware resources and they are run concurrently. Then, the processing of each layer is pipelined and the throughput for the whole network is determined only by the size of the largest activation map, usually the input image, in the network. In addition, unlike the conventional accelerators, the output activations of a layer are not collected in an on-chip buffer or off-chip memory, and each produced output activation goes to the TDNN for the next layer and is consumed immediately. This eliminates array-type memory access completely together with the use of registers for weights. Figure 6 shows how the convolutional layers of a net are converted into a multi-layer TDNN. The convolutional layers of an original net are typically fully-connected channel-wise, but due to the pruning, they can have sparse channel-wise connections. In addition, the figure is depicted assuming that each neuron has a processing delay of 1 time step and the tensor factorization is not performed.

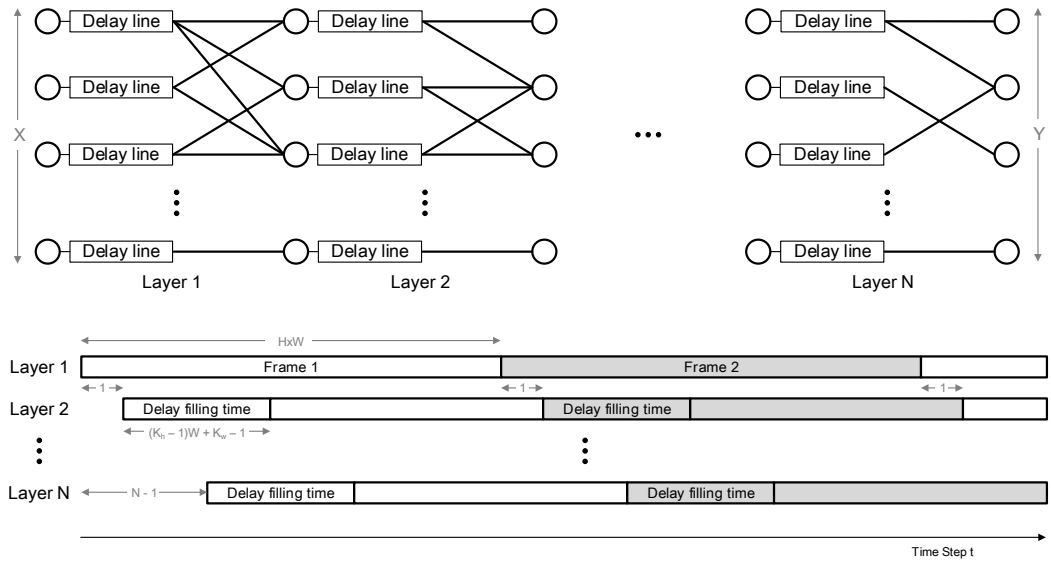


Figure 6. The TDNN conversion allows us to have simple structures even for convolutional layers. In spite of the word-serial processing of the TDNN, the performance is much higher than those of the conventional accelerators because the processing of each layer is pipelined.

3.3. Overall Procedure

The overall compilation procedure is depicted in Figure 7. It takes a common feedforward net used in the machine learning community as input. For the network, we use a data structure similar to other deep learning frameworks, and the weights in a layer are stored in a multi-dimensional array. For each convolutional layer, we perform tensor factorization. For each fully-connected layer, we perform matrix factorization (Step 1). Both are based on singular value decomposition (SVD). Then, all weight parameters in the network are ranked together using the metric proposed in [21], and all but top k parameters are set to zero (Step 2). After the pruning, the predictive performance of the network can be degraded. We thus fine-tune the simplified network again against the training data, recovering the lost performance (Step 3). We rely on existing deep learning (DL) libraries for the fine-tuning and the network is converted into a network of a target DL library. This fine-tuning step allows us to approximate the network very aggressively. In order to prevent the fine-tuning from reverting the truncated parameters, we use a mask matrix or tensor. We can multiply the weight matrix or tensor by the mask element-wise, and can set the mask as non-learnable parameters before the fining-tuning. Alternatively, we can mask the weight update using the mask during the fine-tuning. When the fine-tuning is done, the weights in the NNC are updated from the weights in the target DL library. The TDNN conversion is performed after the factorization and the pruning (Step 4). The time delay neural network generated in this step is represented in a directed acyclic graph (DAG),

and a delay or a neuron (a connection) is represented in a vertex (an edge). The weights are annotated to the edges. The flow after the TDNN synthesis is similar to that for typical DSP custom hardware design methodology [23]. Floating-point weights are converted into fixed-point weights (Step 5). For the fixed-point weights, we use a given fractional bit-width, while the integer bit-width is determined automatically per layer to be large enough not to cause overflow. Therefore, the total bit-width (i.e., the word-length) of weights varies per layer. Then, the TDNN simulator written in a high-level language runs for a given bit-width of activations and given inputs (Step 6). During this process, it checks if any overflow occurs in the activations. It also evaluates the final accuracy reflecting the finite-precision effects of both the weights and the activations, and generates the expected outputs for the given inputs. Finally, the DAG is converted into a Verilog netlist (Step 7). Each vertex in the DAG corresponds to an instance of pre-designed modules in the netlist and the instances are connected following the edges in DAG. The inputs and the expected outputs are used for the functional verification of the netlist and pre-designed modules.

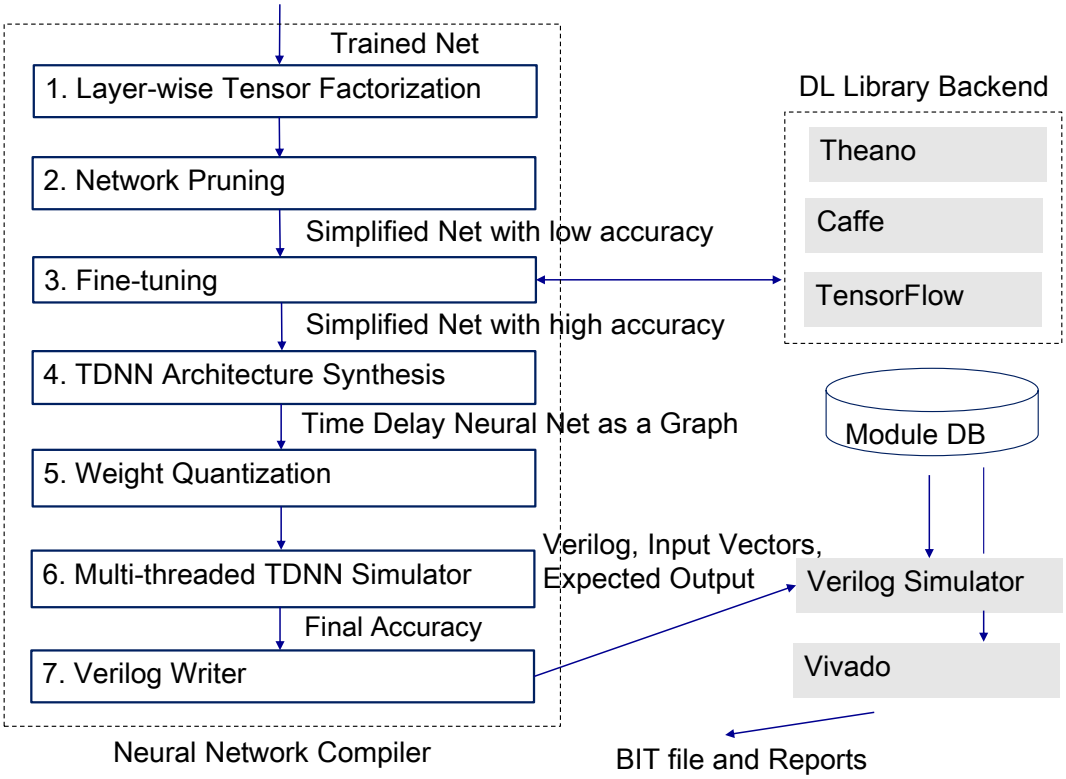


Figure 7. Toolflow.

4. Neuromorphic Hardware

A fully connected neural net with N input neurons and N output neurons has N^2 connections, and existing neuromorphic systems equipped with large arrays (e.g., memristor, X-bar, or SRAM) to implement these dense connections efficiently. However, to implement sparsely connected nets in the neuromorphic fashion, it is more efficient to use a set of small arrays than a large array as also pointed out in [24]. Thus, for the compressed (simplified) nets, we need to design a new neuromorphic hardware architecture different from existing large array-based architectures. However, instead of designing a new neuromorphic architecture, we can leverage FPGAs since they are already designed for multi-level circuits with sparse connections. In addition, they have programmable interconnects. All we need to turn an FPGA into a neuromorphic computing system is to implement the neuron model in logic. We implement the neuron and the synapse as a bit-serial adder and a bit-serial multiplier, respectively. Then, we can convert a simplified neural net into a logic circuit as

shown in Figure 8. The sparse connection is essential because otherwise the resulting logic circuit cannot be routed due to the routing congestion. The bit-serial multipliers and adders not only reduce the circuit area, but also allow us to manage the interconnection between layers. In addition, the buffer size for the pipelining across layers becomes only M bits for a layer of M neurons. Note that a convolutional layer with M output channels is converted into a TDNN with M neurons. Thus, the convolutional layer requires only a M bit buffer to enable the pipelining. Once the convolutional layer produces a M-bit code, the next layer can start some processing irrespective of the output map size. This is possible because our system employs the word-serial and bit-serial architecture for 2D convolution.

Let n_w and n_a be the bit-widths of weights and activations, respectively. The proposed neuromorphic hardware is a composition of building blocks and the building blocks in our system transmit and receive each digit of an activation serially. In our implementation, the least significant bit (LSB) of an activation comes first. To represent signed real numbers, we use the fixed-point format and let m_w and m_a denote the fractional bit-widths of weights and activations, respectively. We manually design the following modules as the building blocks:

- **Synapse:** consists of a n_w-bit register to store the weight and a bit-serial multiplier, which mainly comprises full adders and a register to store intermediate results. For minimum area, we employ the semi-systolic multiplier [25], which approximately requires n_w AND gates, n_w full adders, and $2n_w$ flip-flops.
- **Neuron:** with k inputs comprises a k-input bit-serial adder to sum up the outputs of the k synapses connected to the neuron. We prepare neurons with various k. Adding a bias and evaluating the activation function is related to the function of biological neurons, so it may be natural for the neuron to have units for those functions. However, in our system, many neurons do not require those functions and we implement them in separate modules.
- **Delay element:** is realized by a 1-bit n_a stage shift register.
- **Bias:** consists of an n_w-bit register to store the bias value and a full-adder, whose inputs are fed by a module input, a selected bit of the register, and the carries out at the previous cycle.
- **Relu:** zeros out negative activations. Since the sign bit comes last, this module should have n_a cycle latency at minimum.
- **Max:** compares k input activations by using bit-serial substractors. If $k = 2$, one bit-serial substractor is used. The comparison is done after the MSB of activations is received, so it should also have n_a cycle latency at minimum. If $k > 2$, we can create a tree of the two-input max modules, but this increases the latency. For minimum latency, we can perform $k(k-1)/2$ comparisons in parallel.
- **Pool:** performs subsampling, handles borders, and pads zeros. It keeps track of the spatial coordinates of the current input activation in the activation map and invalidates the output activations depending on the border handling and the stride. It can also replace invalidated activations by valid zeros for zero-padding.

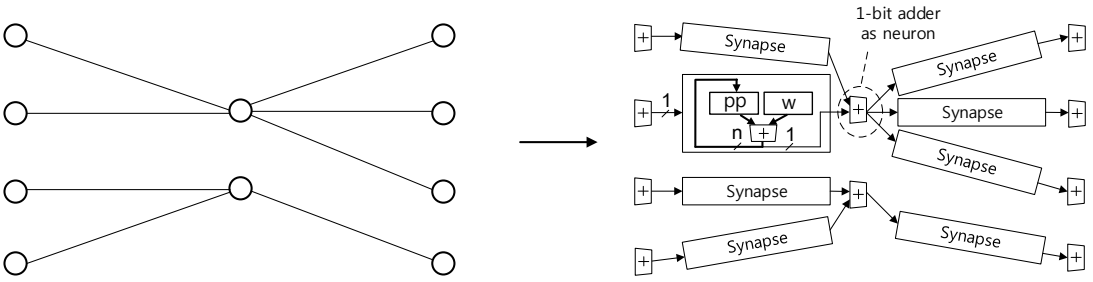


Figure 8. The simplified network is converted into a logic circuit. The 4-bit output of this layer is sent to the next layer and is consumed immediately, keeping the buffer size between layers to a minimum.

Figure 9 depicts a timing diagram of our implementation. It takes $n_a + n_w$ cycles to compute the product of a n_a-bit activation and an n_w-bit weight. To put the product back to the n_a-bit format, we truncate the least significant m_w-bits, and the first valid output becomes available m_w cycles later. Thus, the synapse comes to have an inherent m_w cycle delay. To simplify the design, we pad a $n_a + n_w - m_w$ cycle delay at the output of the neuron, and the registers for this delay can be moved forward to improve timing (i.e., they are used as pipeline registers and can be re-timed). Then, the outputs of a layer become available $n_a + n_w$ cycles after the inputs arrive, and all the synapses in the system start a synaptic operation at the same time at every $n_a + n_w$ cycle. Delay elements perform shift only for the first half of the $n_a + n_w$ cycle period.

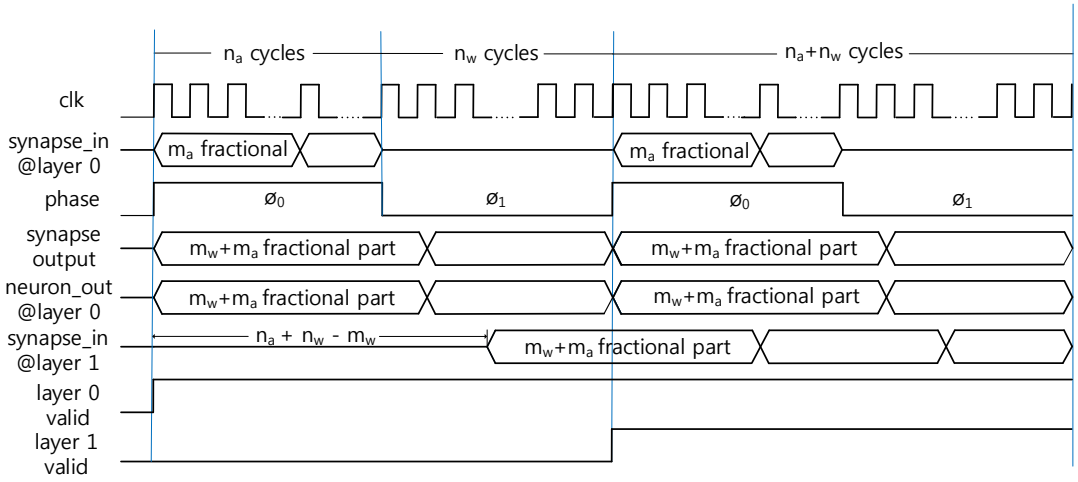


Figure 9. The $(n_a + n_w)$-bit result is aligned by the pipeline registers, which truncates the least significant m_w bits.

5. Experimental Results

We implement the neural network compiler (NNC) in Python language. Neural networks are trained off-line using NVIDIA Titan X GPU and are fed into the compiler. We chose Artix 7 100T and Kintex 7 325T field-programmable gate array (FPGA) as the target platforms. Our experiments can be considered in two ways. First, FPGAs can be considered a general-purpose, not-highly-optimized neuromorphic processor, and the experiments are regarded as making software for the processor in part using existing hardware synthesis tools. FPGAs already have plastic connections, and processing elements and memories are mixed in space. The SRAM-based look-up tables (LUTs) serve as memories as well as processing elements. Second, our experiments can also be considered to build a prototype of an application-specific neuromorphic processor. While the neural network compiler generates a hardware model in Verilog in an application-specific fashion (e.g., weight parameters are hard-coded), we can easily extend it into a general-purpose hardware model if programmable interconnects are available. We elaborate the generated hardware models for a target platform using Xilinx Vivado. The shift registers in delay elements are refined into a LUT-based shift register, not a chain of flip-flops, and even long shift registers are implemented efficiently. We measure the power consumption of the FPGA boards at a 12 V power supply. For the dynamic power consumption of the FPGA chips, we turn on and off the entire clock distribution and measure the difference. The static power consumption is obtained from the power report of Vivado. The FPGA logic operates at 1 V and the efficiency of 12 V–1 V conversion is assumed to be 85% for the chip power measurement.

5.1. Benchmarks

To demonstrate our approach, we use three neural networks. To describe the NN architectures, the fully-connected layer and the convolutional layer are denoted by F and C, respectively. For the

MNIST hand-written digit classification, we use a softmax regression (1F) and a 3-layer convolutional neural network (2C1F). They achieve 92.23% and 99.57% for the test set of MNIST, respectively. For the CIFAR-10 natural image classification, we use a 6-layer convolutional neural network (4C2F), which is trained with data augmentation. We pad zeros to make the size of the input 40×40, and randomly crop it to be 32×32. We also use random flip. The 4C2F has 4 convolutional layers, a fully connected layer and a final softmax layer. Each convolutional layer has filters of size 3×3. The 2nd and the 4th convolutional layers are followed by 4×4 max pooling layers with a stride of 2. We use the rectified linear unit as the activation function. We preprocess the data using global contrast normalization only. We train this network for 150 epochs. The weight decay (l_2 regularization) is set to 0.002; the learning rate is set to 0.1 initially; we use batch normalization. This network achieves 89.10% classification accuracy in the test set.

5.2. Network Simplification

We map the two nets for MNIST into Artix 7, and the net for CIFAR-10 into Kintex 7. To fit the three networks in the target FPGAs, we reduce the numbers of parameters to 3 K, 4 K, and 18 K for 1F, 2C1F, and 4C2F, respectively. Then, the accuracies of the nets become 92.65% (+0.42%), 99.38% (−0.19%), and 83.58% (−5.52%), respectively. Since the number of parameters are determined by the capacity of FPGAs, we lose some accuracy for the two nets. In particular, we require a larger FPGA to maintain the accuracy of 4C2F, but considering the expense, we use the affordable FPGA. Table 1 compares the original 4C2F and the simplified 4C2F in detail.

Table 1. We combine tensor factorization with pruning and simplify the convolutional net.

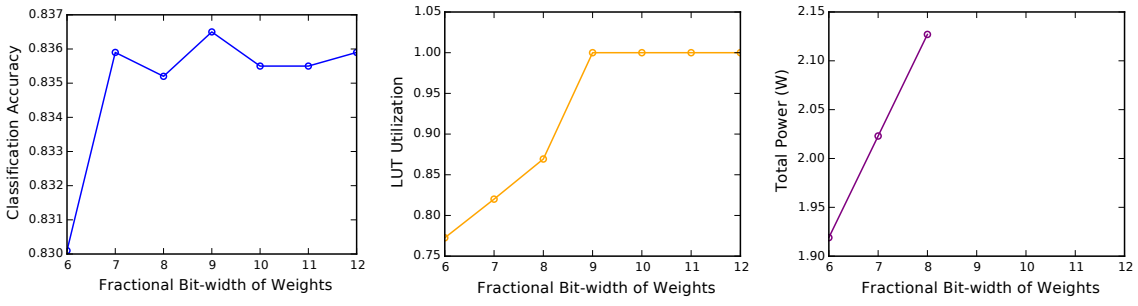
Original			**Simplified**		
Config.	**Params**	**SOPs**	**Config.**	**Params**	**SOPs**
			conv1-3	9	9.22 K
conv3-64	1.73 K	1.56 M	conv3-27	186	167 K
			conv1-64	351	316 K
Subtotal	1.73 K	1.56 M	Subtotal	0.55 K	493 K
			conv1-64	719	647 K
conv3-64	36.9 K	28.9 M	conv3-64	705	553 K
			conv1-64	847	664 K
Subtotal	36.9 K	28.9 M	Subtotal	2.27 K	1.86 M
maxpool					
			conv1-64	1263	213 K
conv3-128	73.7 K	8.92 M	conv3-128	899	109 K
			conv1-128	1586	192 K
Subtotal	73.7 K	8.92 M	Subtotal	3.75 K	514 K
			conv1-128	2689	325 K
conv3-128	147 K	17.8 M	conv3-128	964	117 K
			conv1-128	2717	329 K
Subtotal	147 K	17.8 M	Subtotal	6.37 K	771 K
maxpool					
			conv1-128	1617	26 K
FC-256	524 K	524 K	conv4-256	701	701
			conv1-256	1497	1497
Subtotal	524 K	524 K	Subtotal	4 K	28 K
FC-10	2.56 K	2.56 K	FC-10	1166	1166
			FC-10	84	84
Subtotal	2.56 K	2.56 K	Subtotal	1.25 K	1.25 K

Table 1. *Cont.*

Original			Simplified		
Config.	**Params**	**SOPs**	**Config.**	**Params**	**SOPs**
Total					
Conv total	260 K	57.2 M	Conv total	13 K	3.64 M
FC total	527 K	527 K	FC total	5 K	29.3 K
Net total	787 K	58 M	Net total	18 K	3.7 M

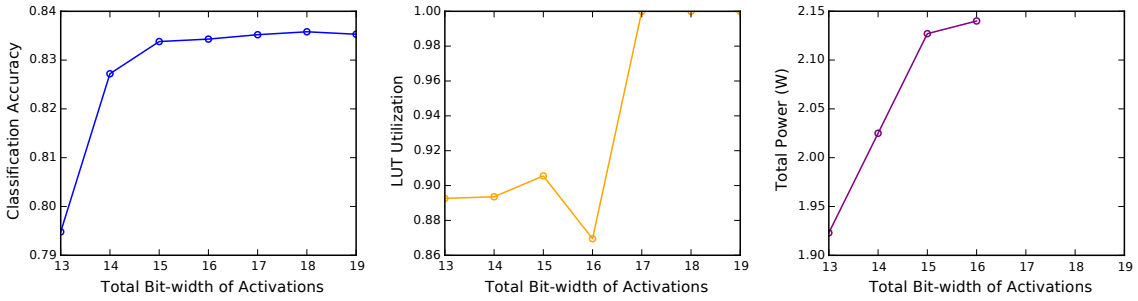
5.3. Bit-Width Determination

We determine the bit-widths of activations and weights after the network simplification. The integer bit-widths of activations and weights are fixed to the minimum values not to cause overflow, but we can freely determine the fractional bit-widths of them, denoted by m_a and m_w, which are some of the major design parameters of our system, and allows us to trade off the implementation cost against the accuracy. Figure 10a shows the accuracy varying m_w for 4C2F. The accuracy is obtained without quantizing activations. Figure 10b,c show the LUT utilization and the total power consumption of the FPGA varying m_w when the activation bit-width is 16. When $m_w > 8$, the design becomes too large to fit the FPGA. The weight bit-width directly affects the area of the bit-serial multipliers, and the LUT utilization decreases substantially as the bit-width is reduced. However, to maximize the accuracy, we use $m_w = 8$ for 4C2F, although $m_w = 7$ results in a higher accuracy in this specific case. The integer bit-width of weights varies per layer and they are between 1 and 3. Thus, the total weight bit-width is between 10 and 12 including the sign bit. We obtain the dynamic range of activations using the training set of CIFAR-10 and the integer bit-width of activations is set to 6 for 4C2F. Figure 10d–f show the accuracy, the LUT utilization, and the total power varying m_a (thus, the total bit-width) when $m_w = 8$. When the bit-width is larger than 16, the design becomes too large to fit the FPGA. The activation bit-width does not affect the area of the bit-serial multipliers, although it may reduce the area of the shift registers for delay elements. In the FPGA, 32-stage 1-bit shift registers are implemented by using one LUT, and in terms of area, it is rather better to align the activation bit-width to that than simply minimizing it. Reducing the activation bit-width allows us to run the design at a lower clock frequency, which can reduce the power consumption substantially. We use $m_a = 9$ for 4C2F, and the total bit-width of activations becomes 16 (i.e, $n_a = 16$). Similarly, we use $n_a = 16$ and $m_a = m_w = 7$ ($m_a = m_w = 8$) for 1F (2C1F).



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

Figure 10. *Cont.*



(**d**) (**e**) (**f**)

Figure 10. The weight bit-width affects the space that the circuit takes up, whereas the activation bit-width affects the computation time.

5.4. Comparison Baseline

We compare our system with CPU, GPU, an FPGA accelerator, and existing neuromorphic systems. For CPU and GPU, our neural network model is converted to a Caffe model. Then, the forward propagation time is measured using Caffe's `time` command. For CPU, we use an Intel Xeon E5-2650 processor, and during the inference, the CPU power and the DRAM power are obtained by Intel's `pcm-power` utility. For GPU, we use NVIDIA GeForce GTX Titan X GPU. The GPU power is obtained by `nvidia-smi` utility. Note that this utility reports the entire board power. The model simplification reduces the amount of required computation, and the throughput should be increased for the simplified models taking advantage of zeros. However, all existing deep learning libraries support dense tensors only, and our simplified models rather worsen the performance due to the increased number of layers. Thus, we cannot measure the real performance of the simplified models in CPU and GPU (This is possible only for fully-connected layers using NVIDIA cuSPARSE library at the moment). The simplified 4C2F requires about 16× less amount of computation than that of the original net, and ideally, the throughput should be increased by about 16×. However, in practice, this is not feasible because of the overhead of the sparse representation such as indexing. For example, in [6], a compressed fully-connected layer requires 11× less amount of computation, but going sparse matrices from dense matrices gives only 4× speedup. Thus, we will estimate the performance when using the sparse tensors by $P \times (1 - \alpha)$ where P is the ideal throughput and α is the sparse overhead factor, which is assumed to be 0.65. For the FPGA accelerator, we compare our system with the one proposed in [4], which achieved 61.62 GFLOPS. The original 4C2F requires 58 × 2 MFLOPs, so we assume that its throughput is 61,620/(58 × 2) = 531 Images/sec. For the simplified net, we assume the same sparse overhead factor for the accelerator. For the accelerator, we also assume that the other components than the FPGA and the DRAM in the board consumes 4 W. The results of existing neuromorphic systems are obtained by the recent literature.

5.5. Results

Table 2 summarizes the results for the three networks. All the implementations operate at 160 MHz. Thus, the designs take one input pixel at 5 MHz (160 MHz/2 × n_a) speed seamlessly. The classification accuracy (Accu) is evaluated for the original models, the simplified models and the implementations using the test set. Both CIFAR-10 and MNIST have 10,000 samples in the test set. The implementation accuracy is the predictive performance of the actual system and reflects the finite-precision effects. Our Python/numpy-based simulator runs for an implementation model of the TDNNs and measures the accuracy generating expected outputs of the final softmax layer. These outputs are validated through RT-level simulation using Synopsys VCS. The two convolutional NNs have a larger number of connections (Conn) than the number of parameters (Param) due to the weight sharing. For the TDNNs, the number of units (Unit) and the number of delay units (Delay) are shown as well. Note that the units, connections, and delay units of the TDNNs are mapped one-to-one

onto the neurons, synapses, and delay elements in the implementations, respectively. Thus, they also indicate the numbers of neurons, synapses and delay elements. For the original NNs, the number of connections equals the number of required synaptic operations (SOPs), while they are different in TDNNs. One synaptic operation (SOP) is translated to 1 MAC in fixed-point systems and 2 FLOPs in floating-point systems. The total area (Area) of the implementation is measured by the number of look-up tables (LUTs). Most LUTs in the FPGA are used to implement synapses. The hardware cost is mainly determined by the number of synapses (thus, the number of connections in the TDNN). Since each synapse performs 5M SOPs per second, the theoretic (actual) performance of 4C2F comes to 87 G (19 G) SOPs per second.

Table 3 compares our system with existing neuromorphic systems for the MNIST task. Throughput is measured by images per second. Energy-efficiency (Energy) is evaluated by energy per image. Table 4 compares our system with existing computing systems and shows that it is a very different type of computing systems from existing ones. As in [6], we assume that our system targets at latency-critical applications, and compare it to the other systems when the batch size is one. In addition, when we compare the energy-efficiency of our system with that of the GPU, we use the board power instead of the chip power since it is not available for the GPU. Our system gives 125×, 4.7×, and 9.2× speed up over the CPU, the desktop GPU, and the FPGA accelerator for the entire network, respectively. It also provides 2168×, 105×, and 62.5× higher energy-efficiency over the CPU, the GPU, and the FPGA. Even if we exclude the improvement by the simplification, our system provides 1.68× speed up over the FPGA, and 19.1×, and 11.4× higher energy efficiency over the GPU and the FPGA. The improvements over the FPGA accelerator have been achieved by using almost the same FPGA devices in a different way. Our system based on the off-the-shelf chip is even comparable to TrueNorth that is based on a custom chip. Our system provides a slightly higher accuracy and 3.9× speed-up at only 2.75× lower energy-efficiency.

Table 2. An original feedforward neural network is simplified and rolled into a TDNN by the neural network compiler (NNC). The units, connections, and delays of the TDNN are mapped to neurons, synapses, and delay elements in the field-programmable gate array (FPGA).

Net	Task	Original Model					
		Accu	Param	SOPs	Conn	Unit	Delay
1F	MNIST	0.9234	7840	7840	7840	-	-
2C1F	MNIST	0.9957	0.1 M	6 M	6 M		
4C2F	CIFAR	0.8910	0.8 M	58 M	58 M		
Net	Task	Simplified TDNN Model					
		Accu	Param	SOPs	Conn	Unit	Delay
1F	MNIST	0.9265	3000	17 K	3161	243	7047
2C1F	MNIST	0.9938	4000	0.4 M	4567	1173	5020
4C2F	CIFAR	0.8358	18K	3.7 M	17338	3125	4274
Net	Task	FPGA Implementation					
		Accuracy		Area		Power	
1F	MNIST	0.9209		29 K		0.72 W	
2C1F	MNIST	0.9937		49 K		0.54 W	
4C2F	CIFAR	0.8343		177 K		2.14 W	

Table 3. Comparison with existing neuromorphic systems for MNIST dataset.

	TrueNorth [26]	SpiNNaker [4]	Minitaur [27]	This Work
Neural Model	Spiking	Spiking	Spiking	Non-spiking
Archtecture	Non-von Neumann	Von Neumann	Von Neumann	Non-von Neumann
Technology Node	28 nm	130 nm	45 nm	28 nm
Off-chip IO	No	Yes	Yes	No
Device	Custom	Custom	Spartan 6	Artix 7
Accuracy	99.42%	95.01%	92%	99.37%
Power	0.12W	0.3W	1.5W [a]	0.59W
Throughput	1000 FPS	50 FPS	7.54 FPS	6337 FPS
Energy	0.121 mJ	6 mJ	200 mJ	0.093 mJ

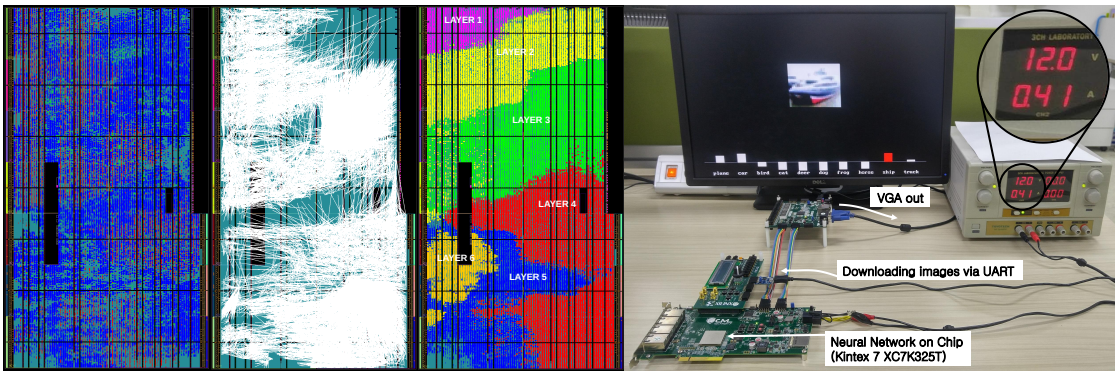
[a] Memory power is not included.

A TDNN model is refined to a network of neurons, synapses, and delay elements preserving the topology, and the IC design tool places them in a space and connects them automatically. This provides an interesting visualization of neural networks. The 4C2F on the FPGA is shown in Figure 11a–c. The network on the FPGA is stimulated by images in the test set of CIFAR-10, which are transferred via UART, and the output of the network is visualized in a screen as shown in Figure 11d. The neuromorphic system successfully classifies images in real-time, and the results match with the simulation exactly.

Table 4. Comparison with existing computing platforms for CIFAR-10 dataset.

	CPU	GPU			FPGA Accelerator [4]		TrueNorth [28]	This Work
Device Type	E5-2650	GeForce Titan X			Virtex 7 485T		Custom	Kintex 7 325T
Memory Array	DRAM	DRAM			DRAM		SRAM	None
Technology Node	32 nm	28 nm			28 nm		28 nm	28 nm
Net Model	Original	Original	Original	Simplified	Original	Simplified	-	Simplified
Tensor Type	Dense	Dense	Dense	Sparse	Dense	Sparse	-	-
SOPs	58 M	58 M	58 M	3.7 M	58 M	3.7 M	-	3.7 M
Batch Size	1	1	64	1	1	1	-	1
Accuracy	0.8910	0.8910	0.8910	0.8358	0.8910	0.8358	0.8341	0.8343
Power (B)	-	109 W	190 W	109 W	18.61 W	18.61 W	-	4.92 W
Power (P + M)	37.25 W	-	-	-	14.61 W	14.61 W	0.204 W	2.14 W
Throughput	39 FPS	1040 FPS	13,502 FPS	5824 FPS	531 FPS	2974 FPS	1249 FPS	4882 FPS
Energy (B)	-	105 mJ	14 mJ	18.72 mJ	35.05 mJ	6.26 mJ	-	1 mJ
Energy (P + M)	954 mJ	-	-	-	27.51 mJ (a)	4.91 mJ (b)	0.16 mJ	0.44 mJ (c)
					Improvement by simplification and hardware (a/c)			62.5×
					Improvement by hardware (b/c)			11.2×

B, P, and M are board, processor, and memory, respectively.



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

Figure 11. Neural network on a chip and its demonstration. (**a**) Our system is a network of three types of building blocks (delays in red, neurons in green, and synapses in blue). The synapses occupy most area. (**b**) The connectivity of the building blocks is shown. (**c**) A function is dedicated to processing elements in a specific region as in our brain. (**d**) Our neuromorphic system based on an FPGA classifies images into 10 categories at the speed of 4882 images per second consuming only 4.92 W at the board-level. It uses a non-Von Neumann architecture; any external memory is not used in this system. In addition, the internal block RAMs on the FPGA are not used except those for the frame buffer.

6. Conclusions

In this paper, we have presented a neuromorphic computing system that is newly designed from the microarchitecture to the compiler in order to forward-execute neural networks with minimum energy consumption. This neuromorphic system can scale by simply using a larger FPGA. Since more than 10 times larger FPGAs than the target platform are available in the market, with our approach, it now becomes easy to build neuromorphic computing systems that can execute neural networks with more than 7 million real-valued parameters, fully leveraging existing integrated circuit design techniques. We believe that a neuromorphic chip derived from FPGAs (or, an FPGA tailored towards the proposed circuits) serves as a practical processor for large-scale deep neural networks such as AlexNet and VGG. Although we have mapped the time delay neural networks generated by the neural network compiler into FPGAs, they can also be mapped into emerging neuromorphic devices such as memristors. We thus believe that the proposed computing systems can also serve as a research platform for high-level design studies until new neuromorphic devices are available widely.

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

Funding: This work was supported by the Institute for Information and Communications Technology Promotion funded by the Korea Government under Grant 1711073912.

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

References

1. Ciresan, D.; Meier, U.; Schmidhuber, J. Multi-column deep neural networks for image classification. In Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Providence, RI, USA, 16–21 June 2012; pp. 3642–3649.
2. Sermanet, P.; Eigen, D.; Zhang, X.; Mathieu, M.; Fergus, R.; LeCun, Y. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv* **2013**, arXiv:1312.6229.
3. Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E. Cudnn: Efficient primitives for deep learning. *arXiv* **2014**, arXiv:1410.0759.

4. 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 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015.
5. Alwani, M.; Chen, H.; Ferdman, M.; Milder, P. Fused-layer CNN accelerators. In Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–12.
6. Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M.A.; Dally, W.J. EIE: Efficient inference engine on compressed deep neural network. In Proceedings of the 43rd International Symposium on Computer Architecture, Seoul, Korea, 18–22 June 2016; pp. 243–254.
7. Shin, D.; Lee, J.; Lee, J.; Yoo, H.J. 14.2 DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 5–9 February 2017; pp. 240–241.
8. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv* **2016**, arXiv:1603.04467.
9. Miyashita, D.; Lee, E.H.; Murmann, B. Convolutional neural networks using logarithmic data representation. *arXiv* **2016**, arXiv:1603.01025.
10. Jo, S.H.; Chang, T.; Ebong, I.; Bhadviya, B.B.; Mazumder, P.; Lu, W. Nanoscale memristor device as synapse in neuromorphic systems. *Nano Lett.* **2010**, *10*, 1297–1301. [CrossRef] [PubMed]
11. Schemmel, J.; Bruderle, D.; Grubl, A.; Hock, M.; Meier, K.; Millner, S. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS), Paris, France, 30 May–2 June 2010; pp. 1947–1950.
12. Benjamin, B.V.; Gao, P.; McQuinn, E.; Choudhary, S.; Chandrasekaran, A.R.; Bussat, J.M.; Alvarez-Icaza, R.; Arthur, J.V.; Merolla, P.A.; Boahen, K. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* **2014**, *102*, 699–716. [CrossRef]
13. Merolla, P.A.; Arthur, J.V.; Alvarez-Icaza, R.; Cassidy, A.S.; Sawada, J.; Akopyan, F.; Jackson, B.L.; Imam, N.; Guo, C.; Nakamura, Y.; et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* **2014**, *345*, 668–673. [CrossRef] [PubMed]
14. Cassidy, A.S.; Alvarez-Icaza, R.; Akopyan, F.; Sawada, J.; Arthur, J.V.; Merolla, P.A.; Datta, P.; Tallada, M.G.; Taba, B.; Andreopoulos, A.; et al. Real-time scalable cortical computing at 46 giga-synaptic OPS/watt with $\sim$100$\times$ Speed Up in Time-to-Solution and $\sim$100,000$\times$ Reduction in Energy-to-Solution. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; pp. 27–38.
15. Cassidy, A.S.; Merolla, P.; Arthur, J.V.; Esser, S.K.; Jackson, B.; Alvarez-Icaza, R.; Datta, P.; Sawada, J.; Wong, T.M.; Feldman, V.; et al. Cognitive Computing Building Block: A Versatile and Efficient Digital Neuron Model for Neurosynaptic Cores. In Proceedings of the 2013 International Joint Conference on Neural Networks (IJCNN), Dallas, TX, USA, 4–9 August 2013; pp. 1–10.
16. Arthur, J.V.; Merolla, P.A.; Akopyan, F.; Alvarez, R.; Cassidy, A.; Chandra, S.; Esser, S.K.; Imam, N.; Risk, W.; Rubin, D.B.; et al. Building block of a programmable neuromorphic substrate: A digital neurosynaptic core. In Proceedings of the 2012 International Joint Conference on Neural Networks (IJCNN), Brisbane, Australia 10–15 June 2012; pp. 1–8.
17. Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv* **2014**, arXiv:1409.1556.
18. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
19. Gong, Y.; Liu, L.; Yang, M.; Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv* **2014**, arXiv:1412.6115.
20. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 1–9.

21. Chung, J.; Shin, T. Simplifying Deep Neural Networks for Neuromorphic Architectures. In Proceedings of the 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016.
22. Bosi, B.; Bois, G.; Savaria, Y. Reconfigurable pipelined 2-D convolvers for fast digital signal processing. *IEEE Trans. Very Large Scale Integr.* **1999**, *7*, 299–308. [CrossRef]
23. Cmar, R.; Rijnders, L.; Schaumont, P.; Vernalde, S.; Bolsens, I. A methodology and design environment for DSP ASIC fixed point refinement. In Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany, 9–12 March 1999; ACM: New York, NY, USA, 1999; p. 56.
24. Wen, W.; Wu, C.R.; Hu, X.; Liu, B.; Ho, T.Y.; Li, X.; Chen, Y. An EDA framework for large scale hybrid neuromorphic computing systems. In Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, 8–12 June 2015; ACM: New York, NY, USA, 2015; p. 12.
25. Agrawal, E. Systolic and Semi-Systolic Multiplier. *MIT Int. J. Electron. Commun. Eng.* **2013**, *3*, 90–93.
26. Esser, S.K.; Appuswamy, R.; Merolla, P.; Arthur, J.V.; Modha, D.S. Backpropagation for energy-efficient neuromorphic computing. In Proceedings of the Advances in Neural Information Processing Systems, Montreal, Canada, 7–12 December 2015; pp. 1117–1125.
27. Neil, D.; Liu, S.C. Minitaur, an event-driven FPGA-based spiking network accelerator. *IEEE Trans. Very Large Scale Integr. Syst.* **2014**, *22*, 2621–2628. [CrossRef]
28. Esser, S.K.; Merolla, P.A.; Arthur, J.V.; Cassidy, A.S.; Appuswamy, R.; Andreopoulos, A.; Berg, D.J.; McKinstry, J.L.; Melano, T.; Barch, D.R.; et al. Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing. *arXiv* **2016**, arXiv:1603.08270.