A Survey on the Optimization of Neural Network Accelerators for Micro-AI On-Device Inference

Arnab Neelim Mazumder, *Graduate Student Member, IEEE*, Jian Meng, *Graduate Student Member, IEEE*, Hasib-Al Rashid, *Student Member, IEEE*, Utteja Kallakuri, Xin Zhang, *Senior Member, IEEE*, Jae-Sun Seo, *Senior Member, IEEE*, and Tinoosh Mohsenin, *Senior Member, IEEE*

***Abstract*— Deep neural networks (DNNs) are being prototyped for a variety of artificial intelligence (AI) tasks including computer vision, data analytics, robotics, etc. The efficacy of DNNs coincides with the fact that they can provide state-of-the-art inference accuracy for these applications. However, this advantage comes from the high computational complexity of the DNNs in use. Hence, it is becoming increasingly important to scale these DNNs so that they can fit on resource-constrained hardware and edge devices. The main goal is to allow efficient processing of the DNNs on low-power micro-AI platforms without compromising hardware resources and accuracy. In this work, we aim to provide a comprehensive survey about the recent developments in the domain of energy-efficient deployment of DNNs on micro-AI platforms. To this extent, we look at different neural architecture search strategies as part of micro-AI model design, provide extensive details about model compression and quantization strategies in practice, and finally elaborate on the current hardware approaches towards efficient deployment of the micro-AI models on hardware. The main takeaways for a reader from this article will be understanding of different search spaces to pinpoint the best micro-AI model configuration, ability to interpret different quantization and sparsification techniques, and the realization of the micro-AI models on resource-constrained hardware and different design considerations associated with it.**

***Index Terms*— Deep neural networks, hardware accelerators, quantization, model compression, neural architecture search, inference engines.**

I. Introduction

OUR modern-day lifestyle is supremely influenced by artificial intelligence (AI). This influence has been so overwhelming that there is hardly any sector today where there is no evident effect of AI-based processes. The prime advantage that AI has brought to our day-to-day life is convenience and ease of operation. This convenience is the result of the AI devices being able to perform compute-intensive tasks and replace the human error from the system to a large extent. These days we see AI techniques and devices being used in the fields of knowledge reasoning, medical diagnosis, robotics, vision analytics, e-commerce, navigation, etc. Such proliferation of AI is the consequence of years of research in the domain of traditional AI techniques, classical machine learning (ML), and more modern deep learning (DL) processes. AI has evolved in the last few decades from being able to do small object-oriented tasks to executing large compute-intensive tasks in a matter of minutes or seconds. The latter upshot comes from the introduction of deep neural networks (DNNs) which is the building block of modern DL. The popularity of DNNs has allowed the research community to take massive strides in the domains of image classification, segmentation, human activity recognition, natural language processing, text authentication, etc. What makes it even more interesting is that these DNN frameworks can exceed human-level accuracy when optimized and used in a precise fashion.

The flip side to these advantages is that DNNs become more accurate at the cost of large computation and storage overhead. A practical solution that has been used over the years is to deploy graphics processing units (GPUs) for these computationally heavy applications. However, modern-day DNN applications require more efficient acceleration of DNN computation, which is where DNN accelerators come into the picture. DNN accelerators provide specialized hardware architectures to account for the computational complexity and the billions of multiply-accumulate (MAC) operations it requires to process. However, the aspect of huge overhead of device utilization and energy consumption is still an issue for such accelerators. To tackle this, existing research is being streamlined to make DNN frameworks scale down to the micro-AI level where tiny frameworks get accelerated to compensate for the stringent device constraints of latency, power consumption, and memory. In summary, existing DNN accelerator designs need significant compression and scaling to be able to process complex DNNs in a resource-efficient fashion while also maintaining a reduced power envelope.

There has been a large amount of research towards efficient hardware designs for DNN accelerators and there have been important compression and quantization strategies that have been introduced to facilitate the implementation of these designs. This paper aims to describe the existing approaches and provide extensive details of practical techniques for micro-AI implementation in the manner of a survey. It is imperative to note that this work does not intend to act as

Manuscript received October 13, 2021; revised November 8, 2021; accepted November 14, 2021. Date of publication November 25, 2021; date of current version December 13, 2021. This article was recommended by Guest Editor A. Wu. *(Corresponding author: Arnab Neelim Mazumder.)*

Arnab Neelim Mazumder, Hasib-Al Rashid, Utteja Kallakuri, and Tinoosh Mohsenin are with the Computer Science and Electrical Engineering Department, The University of Maryland Baltimore County, Baltimore, MD 21201 USA (e-mail: arnabm1@umbc.edu; hrashid1@umbc.edu; ukalla1@umbc.edu; tinoosh@umbc.edu).

Jian Meng and Jae-Sun Seo are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: jmeng15@asu.edu; jaesun.seo@asu.edu).

Xin Zhang is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: xzhang@us.ibm.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JETCAS.2021.3129415.

Digital Object Identifier 10.1109/JETCAS.2021.3129415

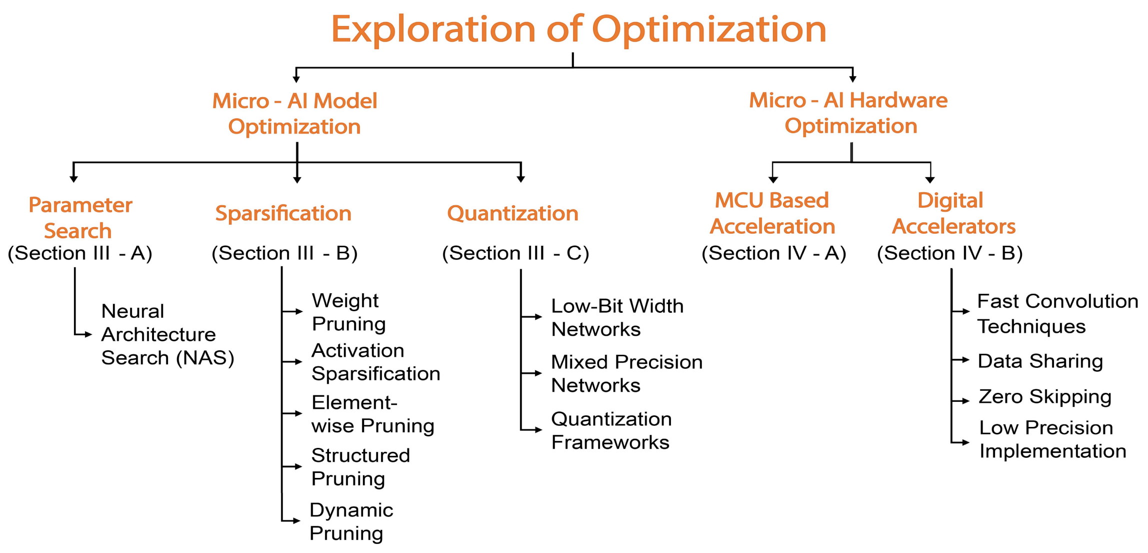

Exploration of Optimization
Micro - AI Model Optimization
Micro - AI Hardware Optimization
Parameter Search (Section III - A)
Sparsification (Section III - B)
Quantization (Section III - C)
MCU Based Acceleration (Section IV - A)
Digital Accelerators (Section IV - B)
Neural Architecture Search (NAS)
Weight Pruning
Activation Sparsification
Element-wise Pruning
Structured Pruning
Dynamic Pruning
Low-Bit Width Networks
Mixed Precision Networks
Quantization Frameworks
Fast Convolution Techniques
Data Sharing
Zero Skipping
Low Precision Implementation

Fig. 1. Illustration of the flow of optimization exploration for energy-efficient Micro-AI deployment in this manuscript.

a replacement for contemporary survey works [1], [2] but provides an extension into newer approaches by building upon the existing ones. Fig. 1 visualizes the structure that is followed in terms of the model design and hardware approaches for micro-AI application. The remainder of the paper is organized as follows.

- Section II provides background for the different DNN components, the process flow, and micro-AI perspectives on inference versus training.
- Section III-A presents the DNN model development techniques and tools used in the software aspect to address the process of neural architecture search.
- Section III-B describes the compression approaches used to reduce memory footprint and improve inference latency of micro-AI architectures.
- Section III-C surveys the quantization techniques in practice and different frameworks that are used to accelerate the process of quantization.
- Section IV presents details about the DNN accelerator designs for micro-AI, and the optimizations used to improve latency and reduce energy consumption with fast convolution, data sharing, zero skipping and low precision implementation.

II. Background

DNNs include several constructs for feature extraction and problem solving for tasks related to image classification, real time risk inference [3], visual question answering [4] etc. The most common constructs in practice range from vanilla convolutional neural network (CNNs) layers, their altered structures in depthwise convolution, fully-connected layers (FC) mimicking multi-layer perceptron operation, to recurrent neural network (RNN) layers and their advanced counterparts in long short-term memory (LSTM) networks.

CNNs can take an image as input and assign importance to various parts of the image which in essence is called feature extraction. CNNs are able to isolate spatial and temporal relevance in an image with the help of filters. A kernel of a predefined shape iterates through the whole image with a predefined stride set by the user. With this traversal of kernels, a wholesome understanding of the image feature is created. As the image becomes more complicated, more filters are required to learn the sophisticated differences in features. In practice, a number of these layers are stacked together with multiple filters to create a neural network which makes the computations to increase by orders of magnitude depending on the parameters specified. One way to reduce computation is to downsample the image space with pooling layers. Popular pooling layers either generate downsampled patches by selecting the maximum value (max-pooling) or the average value (average-pooling) in a patch. These layers act as a process for both noise suppression and dimensionality reduction. Convolution layers coupled with pooling layers make up the backbone of traditional CNN frameworks.

In the construct of depthwise separable convolution (DS-CNN), an image is not convolved according to the number of output channels, but instead transformed only once in depthwise convolution and then elongated to the number of desired output channels in pointwise convolution. Thus, not repeating the same process for each channel saves up on computational overhead by reducing expensive multiplication operations.

Another popular type of DNN is RNN. RNNs are suitable for handling sequential data and hence are used in text and voice authentication applications such as Apple's Siri and Amazon's Alexa [5], [6]. RNNs have internal memory in their architecture that allows them to remember the sequential features of the input. However, in case of RNNs when the gap between the relevant information and the place that needs that information for inference becomes larger, there ensues an issue called vanishing gradients which makes the model stop learning. To address this challenge an advanced variant of RNN called long-short term memory (LSTM) was introduced. LSTMs extend the memory cells for existing RNNs to account for the long-term dependencies.

DNN layers are stacked to form a feedforward network and such networks require training to perform the desired AI task. The complete flow of training for a classification task is detailed in Algorithm 1. The *epochs* in Algorithm 1

Algorithm 1 Train a Network to Classify Some Aspect

```
Input: Consider a network $N$ with an input dataset containing
       $A$ training samples.
Output: Predict the class label for each of the training samples.
# Consider the training batch size to be $B$ and the learning
rate to be $LR$.
# epochs is the number of forward passes required for which
the model is trained.
# $x_{train}$ is an allocation of input objects that comes serially
and $y_{train}$ contains the corresponding labels to the objects.
# Train the model.
for $e \leftarrow 1$ to epochs do
    for $i \leftarrow 1$ to $\lfloor \frac{A}{B} \rfloor$ do
        batch = $x_{train}[i * B : (i + 1) * B]$
        $y_{pred}$ = forwardpass(N, batch)
        loss = L($y_{pred}$, $y_{train}[i * B : (i + 1) * B]$)
        g = backwardpass(loss)
        gradientupdate(g)
return $N$
```

refer to the iterations during training. During each epoch of training, the DNN makes a prediction for the input object it encounters which is named as *forward pass* in Algorithm 1. If the prediction is wrong, then the error is back propagated through the DNN which is known as a *backward pass*. In a *backward pass*, the gradients of the weights are updated per the loss function specified. In this case, the gradient is a function that updates the weights using the predefined learning rate of the optimizer. This step is known as the *gradient update*. In Algorithm 1, y_{pred} signifies the predicted labels after each epoch. This combination of forward and backward passes strengthens the interconnection between the artificial neurons so that the error is reduced, and the model learns to identify the correct label for the object it already encountered. To this extent, the *loss* function calculates the error during each epoch and feeds that information to the optimizer. The goal of the optimizer is to figure out the best parameters that change the attributes of the network to reduce loss. Some of the popular loss functions used in DNN networks are Mean Squared Error (MSE), Mean Absolute Error (MAE), Categorical Crossentropy, etc. Similarly, some common optimizers are RMSprop, Adam, Stochastic Gradient Descent (SGD), etc. The layers in a DNN utilize non-linearity through different activation functions to understand the complex patterns in a data stream. These activation functions range from traditional nonlinear functions in sigmoid and hyperbolic tangents to the more popular rectified linear unit (ReLU) and its variants such as leaky ReLU and parametric ReLU.

The complete process of training requires intermediate weights to be used for gradient update. This becomes a bottleneck for DNN training accelerator designs since these intermediate weights need to be stored separately, increasing the memory overhead for the design. Another bottleneck for DNN training is that the typically higher floating-point precision is required especially for gradient and weight updates, whereas DNN inference is usually performed with a specific fixed-point low-precision for acceleration. Additionally, maximizing the throughput on resource-constrained devices with training is a challenge. In summary, DNN training acceleration is computationally expensive and requires precise hardware design approaches with considerable resource utilization. This article thus focuses on DNN inference engines, which are more suitable to be deployed on micro-AI platforms with low power consumption.

III. Micro-AI Model Optimization

Conventional AI models are usually developed with the sole focus on making them accurate in terms of their specific application regardless of their computational complexity and memory overhead. This is justified for tasks where there is no restriction of computational cost and the availability of fast and efficient GPUs to accelerate the inference process. However, when considering frameworks like micro-controllers, memory-constrained drones, low power edge devices, and small field programmable gate arrays (FPGAs), these bulky AI models do not suit the deployment criteria. In Fig. 2, some of the popular models trained on ImageNet and Google Speech Commands datasets have been illustrated, where models less than 4 MB are deployable on ARM micro-controllers and FPGAs with limited on-chip memory. All models except Hello Edge [7] and its compressed counterpart is accelerated on ImageNet dataset for image classification. Hello edge is targeted for keyword spotting on Google Speech Commands dataset which is a simpler problem than image classification and hence the frameworks demonstrate a high accuracy. DNNs that are accelerated on micro-controller units range from a few KBs (Hello Edge on Cortex-M7 based STM32F746G-DISCO) to a maximum of 4 MB (MCUNet on Cortex-M7 based STM32H743). Models with higher memory footprint require more resources and hence are accelerated through GPUs or FPGAs coupled with off-chip DRAMs.

Furthermore, Fig. 2 also points to the reduction brought about in model size of state-of-the-art networks through low precision acceleration to infer that smaller models shown in Fig. 2 can be suitably implemented on the resource-constrained edge devices. For example, AlexNet is compressed down to 50× its original size in SqueezeNet albeit with considerable accuracy drop. Similarly, ResNet-18 is shrunk down to 14× its actual size in T-DLA [8] (Ternarized Deep Learning Accelerator). However, in order to deploy them on to micro-AI platforms that contain very limited memory, usually not exceeding 1 MB, there needs to be further compression and processing. In this section, we describe the three broad categories of micro-AI model optimization in the lines of parameter search, sparsification, and quantization.

A. Parameter Search

DNN inference engines require the software frameworks to be optimized to allow energy-efficient micro-AI implementation. In DNNs there are a number of variables related to each layer along with different combinations of optimizer and learning rates known as hyperparameter. Traditional methods of hyperparameter search include brute force algorithms such as grid search, random search and genetic algorithms. However, there have been recent developments in the domain of

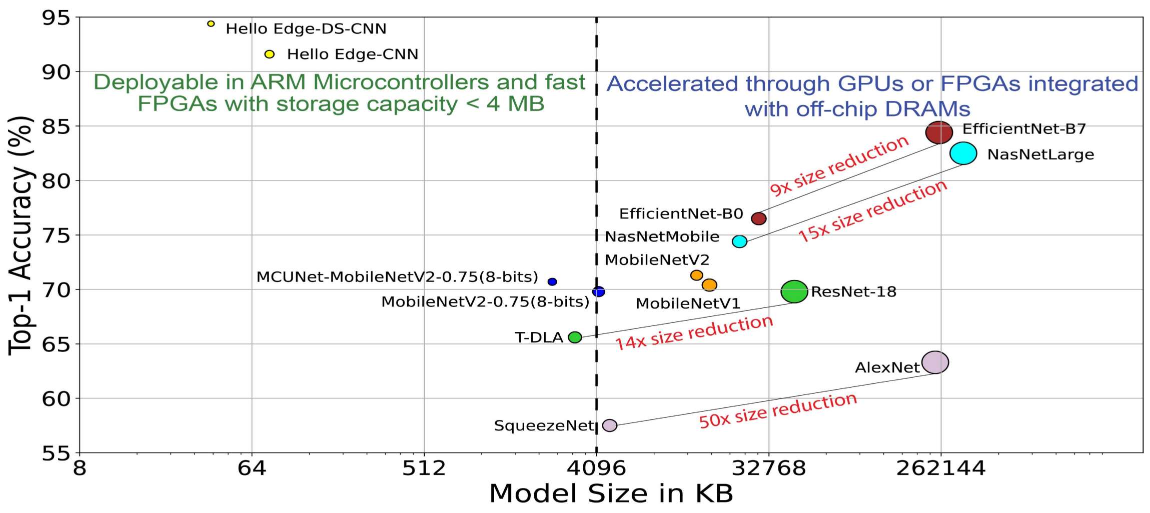


Fig. 2. The accuracy and model size correlation of popular DNNs on ImageNet and Google Speech Commands (Hello Edge). Compressed counterparts of the state-of-the-art frameworks are signified by the same color as the main framework (i.e. AlexNet and SqueezeNet denoted by pink color).

automated frameworks and tools with regards to neural architecture search (NAS) that allow such broad search approaches to be streamlined conveniently and promptly. In this section, we delve deeper into the NAS strategies that are adopted to find the best DNN configuration.

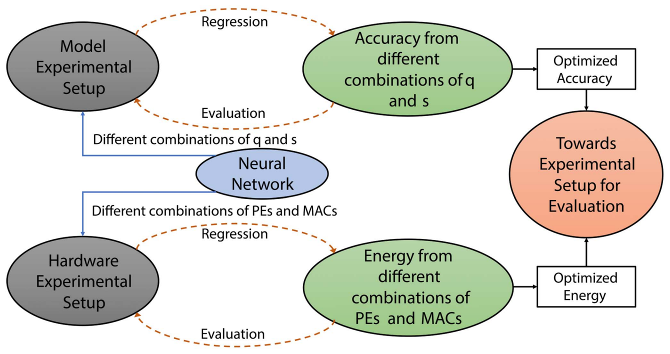
Traditional approaches towards parameter search result in the discovery of architectures with high inference accuracy. However, it comes at the cost of a comprehensive search procedure in most cases which may be infeasible in terms of computation for certain resource-constrained applications. Thus, an alternative and less compute-intensive model design method known as neural architecture search (NAS) is being widely used currently. Modern NAS approaches are based on reinforcement learning (RL) that was introduced in [9] and [10].

The first NAS setup was introduced in [9] that proposed a search strategy based on RL to find the best parameters of an RNN. This was later extended to CNNs in [10] where the authors proposed a sequential model-based optimization (SMBO) approach along with a reinforcement algorithm that searches for cells to find the best configuration. This process involves searching for constructs with high dimensionality while a surrogate model learns and adapts the results for the constructs to guide through the structure space. References [10] and [11] introduces two types of cells: a normal cell and a reduction cell. The normal cell maintains the input dimensionality while the reduction cell decreases spatial dimensions to reduce complexity. The only thing that varies in CNN architectures is the number of normal and reduction cells, which is determined by a controller RNN or the RL algorithm. The ImageNet framework used in NASNet [11] uses the organization of both these cells where the final architecture is a stacked version of these basic blocks. The upside to this approach is that it allows flexibility as architectures coming from cells can be conveniently transferred to other datasets by simply changing the number of cells and filters [11]. Moreover, the size of the search space is reduced since the cell-oriented search strategy consists of shallow networks compared to large architectures.

Even though this strategy of determining the best model is efficient, it does not consider other model constraints such as model size and latency for mobile or edge devices. This is addressed in the works of DPP-Net [12], Pareto-NASH [13], MoNAS [14], RNAS [15] and MnasNet [16], where the authors define this as an optimization problem and find the most compatible setting for implementation. MnasNet is such an automated mobile neural architecture search (Mnas) approach, which also considers model latency to identify a configuration that can achieve a feasible trade-off between the parameters of interest. MnasNet follows a similar RL based search strategy as adopted in [11].

The popularity of using NAS has led to this being used as a profiling technique for mapping hardware power and latency performance. An automated algorithm-hardware co-design scheme is introduced in [17], where the authors search for the best parameters for both software and hardware deployment to boost inference accuracy and energy efficiency. This approach, also known as Codesign-NAS, formulates an optimization problem that has multiple objectives based on different ways of iterating through the search space. More specifically, the authors aim to find a general formulation to automatically find the best algorithm-hardware pairs through RL while optimizing the objective functions of area, latency, and accuracy. In line with that, the authors in [17] follow the NASBench search space used in [18] to develop the CNN search space and find the precise points to be used as the database for Pareto-optimal point search.

The modeling process for NAS can involve advanced techniques like RL and RNN based learning or more basic approaches like regression and linear programming. In [19] and [20] the authors aim to profile and minimize the FPGA energy consumption of DNNs through a regression-based NAS approach where the optimization parameters in consideration are accuracy, power, and energy. There lies a sweet spot between accuracy and model performance in terms of CNNs. Thus, a resource-bound FPGA will perform best when this sweet spot is perfectly utilized. To achieve this, the authors in [19] create an experience space of inference accuracy for



Model Experimental Setup
Regression
Evaluation
Accuracy from different combinations of q and s
Optimized Accuracy
Different combinations of q and s
Neural Network
Towards Experimental Setup for Evaluation
Different combinations of PEs and MACs
Hardware Experimental Setup
Regression
Evaluation
Energy from different combinations of PEs and MACs
Optimized Energy

Fig. 3. The regression based *(q, s)* NAS in [19] iterates through different combinations of scaling *(s)* and quantization *(q)* of the layers to model the accuracy space of the network. With a network of optimized *(q)* and *(s)* set, the regression based method iterates through different combinations of processing engines *(PEs)* and MAC units *(MACs)* to model the energy of the optimized configuration.

TABLE I

A Summary of Popular NAS Methods

Name	Approach	ImageNet Top-1 Acc. (%)	Params. (M)
NAS [10]	RL	96.4 (CIFAR-10)	37.4
NASNet [11]	PPO	74 (ImageNet)	5.3
MnasNet [16]	SMBO	76.7 (ImageNet)	5.2
DPP-Net [12]	PPO	74 (ImageNet)	4.8
Pareto-Nash [13]	LE	71.7 (ImageNet)	4 to 6
MoNAS [14]	MO-RL	87 (CIFAR-10)	-
RNAS [15]	PPO	96 (CIFAR-10)	2.2
Co-design NAS [17]	MO-RL	74.2 (CIFAR-100)	-
(q,s) NAS [19]	Regression	88.7 (CIFAR-10)	0.06

Abbreviations:
Acc: Accuracy, Params: Parameters
RL: Reinforcement Learning, LE: Lamarckian Evolution
PPO: Proximal Policy Optimization
SMBO: Sequential Model-Based Optimization
MO-RL: Multi-Objective Reinforcement Learning

different configurations of VGG-16 architecture on the SVHN and CIFAR-10 dataset. The different configurations correspond to different scaling *(s)* of filters and quantization *(q)* of the layers and thus they name it as *(q, s)* NAS. From the generated experience set the authors model the accuracy results for the configurations through the non-least squares method of polynomial regression. This provides a contour for the accuracy space and allows inference of unknown combinations of scaling and quantization. Once the architecture to be implemented is decided, the authors iterate through different combinations of processing engines and MAC units and create an experience space for power consumption of the hardware. At this point, the authors model the FPGA power consumption based on the power consumption of the feature maps, multiplication operations, logic, and static power of the device.

Thus, the regression-based approach shown in Fig. 3 considers different combinations of q and s to generate the model experimental setup. With the optimum q and s decided from an accuracy perspective, the technique looks for the best possible combinations of *PEs* and *MACs* through the hardware experimental setup. In this way, the approach determines the optimal unknown variables for a given network so that it can be streamlined for energy-efficient implementation. Along with this, it can also utilize input resolution *(r)* as an independent variable instead of quantization and scaling which can lead to different optimization procedures for *(r, s)* NAS and *(q, r)* NAS. Table I shows the a brief summary of the aforementioned NAS techniques based on different strategies and their performance on benchmark datasets.

B. Sparsification

The hardware benefits (e.g., energy, latency reduction) obtained from the sparse neural network could be different with various sparsification schemes. In general, network sparsification consists of two different categories: weight pruning and activation pruning. To establish a deeper understanding of sparsity in hardware acceleration, we investigate a series of recent sparsification strategies from algorithm and hardware perspectives.

1) Weight Pruning Framework: The pioneering research works have shown that DNNs can still retain the performance even if plenty of network weights are removed [21]. Mathematically, let's assume a neural network $f(W)$ where f and W represent the model architecture and weights. Pruning the neural network entails generating a sparse model $f(W \odot M)$, where $M \in \{0, 1\}$ is the binary mask that forces a certain amount of "unimportant" weights to zero. The accuracy degradation caused by the pruning often requires further training (fine-tuning) to recover the accuracy.

More recent research demonstrated the effectiveness of training a sparse candidate network $f(W_0 \odot M)$ from the random initialization weights W_0 [22], [23]. Such findings imply the possibility of pruning the model in an one-shot manner instead of using the over-parameterized pre-trained model as the starting point. However, the iterative "searching and fine-tuning" process is still required.

The pivotal question of weight pruning is: *How to generate the proper binary weight mask M to remove the unimportant parameters with minimum accuracy degradation?* Regardless of the granularity of sparsity, there are two typical importance evaluation schemes: (a) score-based ranking and (b) regularization-based weight penalty. The simplest way to quantify the weight importance is using the absolute magnitude as the significance score, e.g., compare the scores either layer-wise [24] or model-wise [22] then remove the weights with the lowest importance. Furthermore, the magnitude-based score is also valid for the dropout-based sparsification method [25]. Besides the score-based pruning, exploiting the sparsity via regularization [26] during training enables the model to proactively find the optimal sparse pattern by penalizing the unimportant weights close to zero.

2) Activation Sparsification Framework: Storing the intermediate activation/feature map in hardware usually consumes more memory space than the weights. As the weight kernel slides over the feature map, the partial sum computation requires frequent memory access. Therefore, exploiting the activation sparsity could induce more energy and latency reduction compared to static weight pruning. In addition to the memory savings, one of the most favorable advantages of activation pruning is that the sparsity of the feature map enables the hardware to allocate the computation selectively. Characterizing the uninformative features with the sparse representation allows the hardware to spend less computational resources on the corresponding canonical features, leading to improved computational efficiency.

Early studies investigated the prunability of the neurons (feature maps) via stochastic neuron dropout [27], magnitude-aware pruning [28], or regularization-based solver [29]. These activation sparsification methods require the complete activation computation before pruning, which cannot simplify the computation for the current layer. As the CNN models get deeper and wider, the same feature channel could demonstrate different importance for different input samples. Such potential data-dependent channel-wise redundancy motivates researchers to exploit the activation sparsity along the channel dimension in a dynamic manner.

3) Element-Wise Pruning: In general, it is unlikely that the redundant neural network parameters are concentrated in a specific layer or feature channel. The widely distributed redundancy is usually removed through the highly selective sparsification methods. Element-wise pruning considers each weight as a single element and forces the unimportant parameters to zero. Some prior works use L_1 norm as an importance score to evaluate the significance of the weights [24], [30].

The success of magnitude based pruning is built upon the assumption of "magnitude-equals-significance" [31]. However, the heuristically selected pruning ratio and sparsity threshold could be sub-optimal. To compensate the limitation of the magnitude-based pruning, the recent score-based pruning methods evaluate the significance of the weights by considering the distortion that is caused by pruning out the weights [32].

On the hardware side, element-wise pruning can achieve high sparsity with the cost of large index memory storage and irregular memory access. Although some of the recent DNN accelerators support the element-wise sparsity with special hardware designs, employing the non-structured sparsity to hardware might be naturally cumbersome for emerging efficient computational platforms such as SRAM or RRAM-based in-memory computing (IMC) hardware [33]–[35].

4) Structured Pruning: Structured pruning [36], [37] eliminates the unimportant weights in a group-wise manner and allows the resulting sparse model to have better hardware compatibility due to the negligible sparse index overheads. The structured weight sparsity empowers the model to skip computations with respect to the processing elements (PEs), leading to the energy and latency reduction. Similar to element-wise pruning, the essence of structured pruning is removing the unimportant weights in a larger granularity. The pruning structure can be flexibly defined among the different dimensions of the convolution layer. Fig. 4 shows the typical pruning granularities with different group selections. Filter-wise pruning [38] removes the filters with small l_1 norm. However, the pruning sensitivity can be varied for different layers and determining a proper pruning ratio for different layers could be challenging. On the contrary, the regularization-based pruning algorithm aims to automatically find a proper layer-wise sparsity during the training process. Group Lasso [36] penalizes the weight groups W_g by adding the corresponding l_2 norm to the loss $\mathscr{L}$. By doing so, the sparsity searching problem now becomes an optimization task.

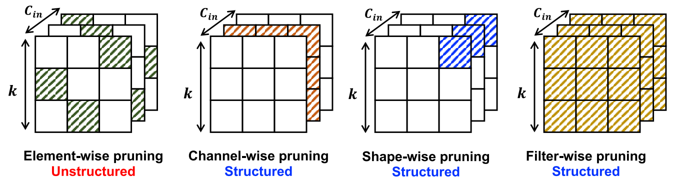


Fig. 4. Different pruning granularities for weight pruning, where k and C_{in} represents the kernel size and input channel size, respectively.

$$\hat{\mathscr{L}} = \mathscr{L}(\mathscr{W}) + \lambda \sum_{g=1}^{G} ||W_g||_2, \quad (1)$$

Specifically, exploiting the sparsity via regularization can be referred to as a *sparse regression* problem [39], group Lasso relaxes the sparsity constraint with the differentiable l_2 norm so that the stochastic gradient descent (SGD) can be employed as the optimization tool. Alternatively, Alternating Direction Method of Multipliers (ADMM) [40] finds the optimal structured sparsity using both SGD and analytical solver. Compared with the original group Lasso algorithm, ADMM achieves 3.2× compression ratio improvements based on AlexNet with the ImageNet dataset. In practice, tuning the optimal group Lasso penalty level for the targeted sparsity could be time-consuming. On the other hand, a proper layer-wise pruning ratio is also concealed for ADMM before training. To address this, the recent works focus on the global optimization for different sub-networks with different accuracy-sparsity trade-off [41].

For the hardware implementation, different edge-AI hardware could have different architectures. Naïvely pruning the model with arbitrary structures might not be able to achieve the theoretical performance with the given hardware. Motivated by this, various hardware-aware pruning algorithms have been proposed. AMC [42] leveraged reinforcement learning by using both accuracy and hardware resources as the reward to compress the model. Cluster pruning [43] first optimizes the filter cluster size based on accuracy and inference latency changes, then prunes the clusters to maximize the hardware performance. On the other hand, the jointly optimized algorithm-hardware design aligns the structured sparsity together with the hardware architecture. With the properly designed crossbar mapping scheme, the high structured sparsity obtained from the small group size can be utilized to improve the energy efficiency of RRAM-based IMC [44].

5) Dynamic Pruning: As opposed to static weight pruning, dynamic pruning exploits the sparsity among the activations. Generally, the pruning granularity can be categorized into 1) element-wise, 2) channel-wise, or 3) block-wise sparsification. Since the importance of the channels varies for different input features, the essence of dynamic pruning is activating the computation for the salient features while ignoring the trivial characteristics. Therefore, the most important step of dynamic pruning is predicting the feature salience before computing the output feature map.

TABLE II
POPULAR NETWORKS UTILIZING SPARSIFICATION AND QUANTIZATION FOR IMAGE CLASSIFICATION

Network	Dataset	Pruning	Precision	Top-1 Acc. (%)
ResNet-18 [36]	CIFAR-10	Structured	FP-32	92.6
VGG-16 [47]	CIFAR-10	Dynamic	FP-32	93.4
VGG-16 [32]	ImageNet	Magnitude	FP-32	64.2
0.5 MobileNet [49]	ImageNet	Structured	FP-8	58.7
VGG-16 [50]	ImageNet	-	FP-16	74.4
ResNet-18 [8]	ImageNet	-	Ternary	65.6
ResNet-18 [51]	ImageNet	-	Binary	51.2
DoReFaNet [52]	ImageNet	-	Binary	50.3

Unlike static weight pruning where the importance could be directly derived from the magnitude scores, the feature maps contain a large amount of high dimensional information, which usually requires gating or encoding to generate the binary decision mask or low dimensional salience vectors. Channel gating neural network (CGNet) [45] first executes a subset of convolution layers then passes the resulting partial sum through a dynamic gating function to generate the element-wise decision masks for the rest of the computation. However, the activated input channels of the CGNet is inconsistent for different output channels, leading to inefficient data reuse. Furthermore, the non-differentiable gating function requires complex approximation during training. The hard gating of the CGNet has been replaced by either a tunable threshold [46] or ReLU function [47], [48], while the saliency predictors are implemented by the light-weight CNN or FC layers. Among all these evolved algorithms, FC layers have less computational costs compared to light-weight CNNs. However, incorporating fully connected layers as the saliency predictor requires dimension reduction (e.g., average pooling) on feature maps [47], [48], which also could be expensive for hardware implementation.

Compared to static weight pruning, implementing dynamic sparsity requires extra computation, which introduces additional memory and power consumption. Thus, the trade-off between dynamic pruning overhead and network performance should be wisely considered in future research.

C. Quantization

The perennial problem of quantization is particularly relevant whenever memory and/or computational resources are severely restricted, and it has been investigated in many DNN models in computer vision, natural language processing, and related areas. Even though binary and ternary networks reduce the model size significantly, there is also considerable accuracy loss due to the compression. In practice, intermediate formats such as 16-bit and 8-bit full precision to mixed-precision formats are comprehensively used. Table. II denotes the popular networks that utilize different quantization and sparsification strategies for image classification. For reference, the ResNet-18 network loses around 14% accuracy when it goes from ternary to binary networks as shown in Table. II. On the other hand, VGG-16 with 16-bit fixed point precision can reach near state-of-the-art accuracy on ImageNet albeit with 16× the model size compared to binary networks. Moving from floating-point representations to low-precision fixed integer values represented in four bits or less holds the potential of reducing the memory footprint and latency by a factor of 16× or more, and reductions of 4× to 8× are readily realized in practice in these applications. In this section, we survey approaches to the problem of quantizing the numerical values in DNN computations, covering the advantages/disadvantages of current methods.

1) Low-Bit Width Networks: Micro-AI inference devices have extremely low memory space for storing the weights and data for DNN models. That is why reducing the data bit-width to the extreme extent is a very popular and demanding task during DNN inference. The most severe quantization method is binarization, which reduces the memory requirement by 32× through constraining the quantized values to a 1-bit representation. A uniform binarization strategy, on the other hand, would result in severe accuracy loss. As a result, a substantial body of literature has presented several solutions to this problem. BinaryConnect [53], which constrains the weights to either $+1$ or -1, is an important study in this area. To replicate the binarization effect, the weights are preserved as actual values and binarized only during the forward and backward passes in this method. Binarized neural networks [54] (BNN) takes this concept a step further by binarizing both the activations and the weights. Because the expensive floating-point matrix multiplications can be substituted with lightweight XNOR operations followed by bit-counting, binarizing weights and activations together offer the added benefit of reduced latency. Another noteworthy work [51] proposes Binary Weight Network (BWN) and XNORNet, which offer improved accuracy by integrating a scaling factor to the weights.

Furthermore, motivated by the fact that many learned weights are near to zero, attempts to ternarize networks have been made by restricting the weights/activations with ternary values, such as $+1$, 0 and -1, specifically allowing the quantized values to be zero [55]. Ternarization, like binarization, dramatically reduces inference latency by avoiding the unnecessary need for costly matrix multiplications. Later, Ternary-Binary Network (TBN) [56] demonstrated that integrating binary network weights and ternary activations may reach the best accuracy and computational efficiency trade-off. Apart from the memory benefits, binary (1-bit) and ternary (2-bit) operations may frequently be implemented efficiently with bit-wise arithmetic, resulting in significant performance gains over higher precisions like FP32 and INT8.

Because uniform binarization and ternarization methods often result in considerable accuracy loss, especially for DNNs for large-scale datasets such as ImageNet, several ways to mitigate accuracy loss in extreme quantization have been presented [57]. Moreover, several works specifically exploit low-precision quantization to reduce the inference time latency. DoReFa-Net [52] stands out to accelerate the training as well by quantizing the gradients in addition to the weights and activations.

For many CNN models used in computer vision tasks, extreme quantization has been successful in dramatically lowering inference/training time as well as model size. There have been recent attempts to apply this concept to Natural

Language Processing (NLP) tasks [58] as well. Extreme quantization is emerging as a powerful method for moving NLP inference tasks to the edge, given the prohibitive model size and inference latency of state-of-the-art NLP models that are pre-trained on a significant quantity of unlabeled data.

2) Mixed Precision Networks: It is prominent that lower precision quantization improves hardware performance. However, uniform quantization of a DNN model to very low precision might result in severe accuracy loss. Mixed precision (MP) quantization, where different bit precision would be allocated to different layers based on their importance, could be beneficial in addressing this problem. More important layers would be assigned higher precision, and comparatively less important layers would be in lower precision. However, the search space for selecting this bit precision per layer grows exponentially with the number of layers, making it challenging to implement. To address this massive search space, various techniques have been proposed. In [59], MP configuration searching problem is formulated as NAS-based problem where Differentiable NAS (DNAS) was utilized to explore the search space efficiently. Reference [60] offered a reinforcement learning (RL) based method to automatically calculate the quantization policy, where the authors employed a hardware simulator to incorporate the hardware accelerator's data into the RL agent feedback. Reference [61] extends the method proposed in [60] considering the limited memory and computational characteristics of tiny edge micro-controllers. Exploration-based methods like [59]–[61] have the drawback of requiring a lot of computational resources and being extremely sensitive to hyperparameters and even initialization. HAWQ [62] proposes a method based on the model's second-order sensitivity for automatically finding the MP settings, which opens up a completely different approach to explore the MP search space. HAWQv2 [63] was extended to MP activation quantization which was presented as 100× faster compared to the RL-based [60] approach. HAWQ-v3 [64] introduced integer only hardware-aware quantization, which includes Integer Linear Programming (ILP) based approach to find the best MP configuration considering application-specific constraint (e.g., model size or latency).

3) Quantization Frameworks (System Integrator Tools and Profiler): DL compilers have become increasingly popular in recent years as a flexible solution to optimizing and deploying DL models. Apache TVM [65], Glow [66] and XLA [67] are three DL compilers that provide optimization and code creation for different hardware platforms. On the other hand, Intel nGraph [68], Nvidia TensorRT [69] and Xilinx Vitis AI [70] are compilers that focus on a single class of hardware systems. Also, recently introduced Q-Keras [71] library extended the Keras library to enable quantization-aware training with MP settings for conventional DNN layers.

IV. Micro-AI Hardware Optimization

The design techniques and optimization strategies discussed thus far only considered the development of an efficient model that provides a reasonable trade-off for different optimization criteria. This, however, represents the aspect of a design from the software perspective only. Actual deployment of such architectures onto micro-AI platforms requires unique hardware design techniques that will lead to fast execution and low power implementation of the models. In this section, we look at accelerator designs for different neural networks on both MCU (Micro-controller Unit) and FPGA, review ways to improve their latency, and analyze different quantization approaches in place to make them lightweight and suitable for micro-AI implementation.

A. MCU Based DNN Acceleration

Micro-controllers are severely limited by their storage capacity to facilitate DL frameworks in their design. Hence, there needs to be a comprehensive reduction of memories from both the network and the libraries that are being used to fit the memory budget.

There is limited literature for DL inference on micro-controllers. Most notable micro-controller-based works include ShuffleNet [72], Hello Edge [7], SpArSe [73], IoTNet [74] etc. ShuffleNet [72] considers CNN architectures for mobile devices with very limited memory where the computation workload is only 40 MFLOPs (mega floating-point operations) compared to AlexNet. Hello Edge [7] focuses on keyword spotting with the Google speech commands dataset using depthwise separable convolutions for memory-limited micro-controllers. In SpArSe [73], the authors use a NAS-based approach to find superior CNN architectures by taking into account the memory constraints of micro-controllers. The aspect of quantization on micro-controllers is addressed in the work of [75] where the authors exploit low-bit width quantization down to 2-bit integer-only operations to deploy the MobileNetV1 model onto the STM32H7 micro-controller. IoTNet [74] on the other hand compromises accuracy for lower computational complexity in a different fashion. The authors in this case factorize 3×3 convolution into 1×3 and 3×1 standard convolution to reduce the number of operations.

Several frameworks and integrated libraries allow DNN acceleration on microcontrollers, namely, TensorFlow-Lite Micro [77], CMSIS-NN [78], and Micro-TVM [65]. These frameworks perform network optimization by considering the best parameters for a memory-efficient design. However, the optimization adopted by these frameworks has a few limitations. First, these frameworks depend on an interpreter for optimization during run-time which increases memory consumption. Second, the optimization is on a layer basis rather than a network basis, which makes it difficult for these frameworks to save further memory by considering a network-oriented memory budget.

These issues have been addressed in [76] where the authors propose a system-algorithm co-design framework to optimize the TinyNAS architecture. They utilize the inference library named TinyEngine to reduce memory efficiently and deploy ImageNet scale DNN on these resource-constrained devices. As shown in Fig. 5, [76] demonstrates the efficacy of their approach by comparing their latency and memory usage against state-of-the-art integrated inference libraries (i.e. MicroTVM Tuned, CMSIS-NN and TF-Lite Micro) for MCU acceleration. The performance boost brought about by

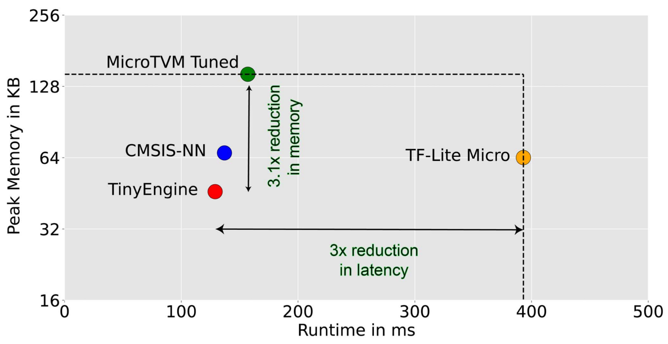


Fig. 5. The reduction in memory usage and latency of TinyEngine [76] when compared to other MCU based inference engines for inference on the CIFAR-10 dataset.

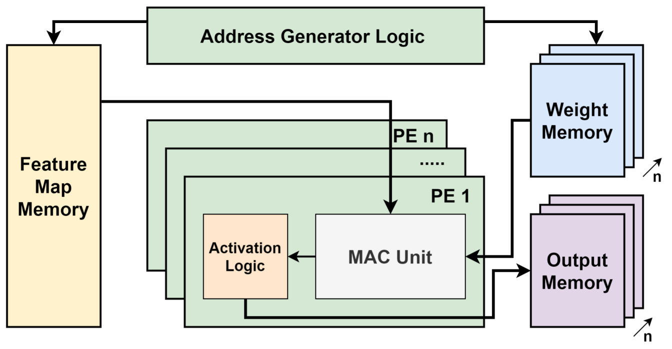


Fig. 6. The general framework for a CNN accelerator consists of memory units (feature map, output and weights), *n* number of processing engines (PE) each of which houses MAC units and activation logic for output formulation and parallelization, and a control logic that generates the addresses to be extracted from the feature map and weight memory.

TinyEngine relates to the implementation being on average 3× faster than TF-Lite Micro and consume 3.1× less peak memory usage compared to MicroTVM Tuned. Furthermore, the TinyEngine reduces memory usage by 2.1× in comparison to the interpreter based CMSIS-NN library utilization.

B. Digital Accelerators

Inference engines are usually feed-forward networks since they require no back-propagation and only depend on trained weights to determine the prediction of an input sample. Such a feed-forward setup allows several advantages concerning different design perspectives. First of all, the whole architecture can be prototyped to have a unique data precision which reduces the amount of design logic. Secondly, since the data flow is on one way only, the layers and filters used in the layers can be parameterized easily to allow the feasibility of different sets of implementations. And finally, the overall amount of data access and memory footprint will be considerably smaller compared to training networks which need back-propagation and intermediate weights. Such architectures are also known as systolic architectures because the arrangement of the processing engines, mimics the rhythmical data flow system. Traditional feed-forward engines are based on the commonly used CNN and RNN models. More details about the general setup of such accelerators are elaborated below:

CNN accelerators replicate the kernel traversal operation of CNNs to generate intermediate feature maps. The operation of the design is based on Equation 2.

$$O_{m,n} = \sum_{c=1}^{C_{in}} \left(\sum_{f=1}^{F} \left(I_{f+mS,c} W_{n,c,f} \right) \right) \quad for\ m = 0 \ldots N-1, \quad n = 1 \ldots C_{out} \tag{2}$$

Here, I, O, W, C, and F correspond to input, output, weights, channels and filters respectively and S signifies the stride for the filters. The general CNN accelerator architecture contains the components shown in Fig. 6 and are detailed below.

PE array comprises the main logic modules for an accelerator design. This is where essentially the computations occur between the feature map values and the weights. The operation of conventional CNN accelerators is usually easier to parallelize. In essence, the structure of the PE array defines the parallelization technique that is adopted in the design. Each PE array block also contains at least one MAC unit along with the activation logic to truncate the data values to fixed-point representation. Depending on the size of the workload and scalability of the design there can be different combinations of MAC units and PEs. In addition, the activation function that is typically used is ReLU or hyperbolic tangents.

MAC units are generally a combination of adders and multiplier units. The fixed-point multiplication operation requires the partial sums to be stored temporarily in a register and then fed to the adder along with the next set of multiplication output to generate the result of one specific patch of the input filter.

Address generator units conform to the precise addressing procedure for convolution and fully connected layers. This module extracts the indexes of the feature map values for 2D or 3D convolution and feeds it to control logic to fetch the data from the memory units. Furthermore, the addressing process also considers variable striding of the input filters and padding of the generated patches.

Memory units store the feature map values, weights values, and temporary results in the feature map memory, weight memory, and output memory respectively. The traditional way of keeping memory accesses limited is to swap memories per process termination of every layer belonging to the network.

Some of the representative works following the systolic array architectures for CNN implementation are detailed in Table III. In [79], the authors present a CNN accelerator for time-series classification on MFCC (Mel Frequency Cepstral Coefficients) processed data. In [80], the authors introduce the Winograd algorithm and a fusion architecture to maximize the throughput of the CNN accelerator. Following this, authors in [81] proposed an automated workflow called fpgaConvNet that maps CNN models on FPGAs by considering hardware design space constraints. The key goal of accelerator implementation is to have high energy efficiency, and this is achieved in Eyeriss [50] where the authors introduce a new data-flow called row stationary on spatial architectures. The aspect of deploying large CNNs on to medium density and low-density FPGAs has been explored in Angel-eye [82] and lite-CNN [83] where the authors consider quantization of the data bits down to 8-bit fixed point precision. Recent

TABLE III
POPULAR CNN AND RNN ACCELERATORS IN LITERATURE

Name	Application	Dataset	Architecture	FP-Precision (bits)	Power (W)	E. Eff. (GOP/J)	BRAM (%)	Device
CNN [80]	Image Classification	ImageNet	VGG-16, AlexNet	16	9.4	24.4	15.2	XC7Z045
fpgaConvNet [81]	Image Classification	ImageNet	CFF, LeNet-5, MPCNN etc.	16	1.8	7.3	4.3	XC7Z020
Eyeriss [50]	Image Classification	ImageNet	VGG-16, AlexNet	16	0.3	166	-	VC707
Angel-Eye [82]	Image Classification	ImageNet	VGG-11	8	3.5	24.1	61.4	XC7Z020
Lite-CNN [83]	Image Classification	ImageNet	VGG-16, AlexNet	8	-	33	94.3	XC7Z020
CNN [85]	Object Detection	COCO	YOLO	16	11.8	8.6	55	XZCU102
DeepRnn [86]	Text Generation	DJIA	DeepRnn	16	2.3	0.5	-	XC7Z045
ESE [87]	Speech Recognition	TIMIT	ESE	16	41	61.5	87.7	XCKU060
DeltaRNN [88]	Speech Recognition	TIDIGTS	DRNN	16	7.3	164.1	60.7	XC7Z100

Abbreviations:
FP-Precision: Fixed-Point Precision, E. Eff.: Energy Efficiency

works [84], [85] also implemented reconfigurable CNN accelerators for object detection models such as SSD and YOLO on the Pascal VOC and COCO datasets.

On the other hand, RNN accelerators are usually designed serially as showed in the works of [86]–[88]. RNNs have straightforward addressing unlike convolution and only require a serial pipeline to implement the gate logic. The MAC units in an LSTM accelerator use kernel memory and recurrent kernel memory to perform the matrix-vector multiplication of the equations described in [89]. The activation logic in the case of LSTM is hard sigmoid and hard tanh.

The systolic array inference engines can be further optimized to allow better flexibility, improve run-time and finally to ensure energy-efficiency. To this end, we analyze the processes of fast convolution, zero skipping, data sharing and low precision implementation in the next set of subsections.

1) Fast Convolution: Digital accelerators are designed to make the inference process of bulky DNNs fast and avoid the costly usage of GPUs. This need for fast computation has led to various techniques being explored to make CNNs less compute-intensive. Two of these fast convolution algorithms are known as fast Fourier transform (FFT) and Winograd.

FFT Convolution has a different spatial traversal pattern than traditional convolution. According to [90], FFTs of the image and the kernel can produce a tiled convolution algorithm that mimics the original process. In this case, the transformation matrices are replaced with FFT of both the image and the kernel. The FFT counterparts then go through pointwise multiplication followed by the inverse Fourier transform of the multiplication result to generate the feature map of the same shape. This reduces the arithmetic computational complexity which was further refined by fbfft [91] with regards to GPU implementation. A library that considers FFT convolution of kernels and images was later introduced in cuDNN [92] by NVIDIA. The efficacy of the FFT algorithm for deployment on embedded systems was first evaluated in [93] where the authors showed that such an algorithm can result in considerable storage reduction while not sacrificing accuracy significantly.

For an $N \times N$ image and $K \times K$ filter, the computational complexity in traditional convolution is given by $O(N^2K^2)$, whereas FFT has a computational complexity of only $O(N^2 logN)$, which is much smaller compared to that of traditional convolution layers. One bottleneck of the FFT convolution is that with smaller filters, it is not as fast when compared to larger filters. This aspect is addressed in [94] with input splitting where the data is divided into chunks and called fast Fourier transform overlap-and-add (FFT-OVA), which has the computational complexity of $O(N^2 logK)$. The implementation of the FFT algorithm onto resource-bound FPGAs in the form of a hardware accelerator is delineated in [95], where the authors considered two variants of the FFT convolution along with the vanilla convolution operation deployed through the SPARCNet [96] design for the ResNet-20 architecture with the CIFAR-10 dataset.

Winograd is based on the minimal filtering algorithm and was first introduced to accelerate CNNs in [97]. The 2D convolution of a feature map X with a kernel K through the Winograd algorithm produces the output feature map of Y using Equation 3.

$$Y = A^T[(GXG^T) \odot (BXB^T)]A \tag{3}$$

Here, A, G, and B correspond to the transformation matrices used to generate the Winograd output and $\odot$ denotes element wise multiplication of the matrices. The shape of these matrices are dependent on the shape of X and K.

According to [97], the multiplications required by the minimal filtering algorithm for one dimensional $F(m, r)$ is signified by $m + r - 1$ where m is the size of the output and r denotes the r-tap FIR filter. This also holds for two dimensions when the r-tap filter and output is represented by $F(m \times n, r \times s)$. In this case, the total multiplications amount to $(m + r - 1)(n + s - 1)$. For a 3×3 feature map and 2×2 filter, $F(3 \times 3, 2 \times 2)$, the total multiplications for a traditional convolution is $3^2 \times 2^2 = 36$ which on the other hand is only $(3 + 2 - 1) \times (3 + 2 - 1) = 16$ for the Winograd algorithm. This allows a $2.25\times$ reduction in computational complexity of the network that can be exploited in hardware to introduce fast frameworks. However, the reduction in the computational complexity with the Winograd algorithm is largely dependent on the size of the filters that are being considered. This algorithm is particularly fast when the filters in question are of a small shape and need a lot of strides to iterate through the whole image. One hindrance with the Winograd algorithm is that it eradicates the data sparsity of pruning approaches. To accommodate the Winograd algorithm with sparsity there needs to be a transformation in the activation functions. In line with this, in [98] the authors introduce low-latency sparse Winograd convolution (LSW-CNN) that decreases the latency of the VGG-16 network by 5.1×. On the other hand, [99] used an alternate sparsity exploitation method called sub-row balanced sparsity pattern (SRBS) along with the Winograd algorithm to reduce latency and irregular memory accesses. Another issue with accelerators that try to exploit sparsity with Winograd is that they suffer from stable load balancing. This issue is considered in [100] where the authors use dynamic scheduling to improve the load balance. The modifications allow the accelerator in [100] to achieve the performance of 7.6 TOP/s with the VGG-16 architecture. A systolic PE structure called WinoPE is also introduced in [101] that allows the flexibility of implementing different kernel sizes for the complex Winograd convolution. WinoPE outperforms state-of-the-art designs in terms of throughput and DSP efficiency with 1.33 GOPS/DSP on the ZCU102 FPGA.

2) Zero Skipping: As discussed in Section III-B, various sparsification algorithms introduce zero values into both weights and activation with different granularities. On the hardware side, the sparse elements/groups can be further utilized to skip the convolution operations partially. However, efficiently locating the sparse elements could be challenging for hardware implementation.

For both weight and activations, properly saving the sparse patterns is necessary before the efficient computation. Storing the sparse matrix with a simple coordinate format could be a huge storage burden. Various simplified sparse format reduces the storage costs [87], which allows the accelerators to efficiently perform the computation by skipping the sparse elements among weights and activations [87], [102]. However, converting the sparse coordinates to the compressed indexes (e.g., compressed sparse row/column format) still leads to non-negligible index memory, which causes irregular memory access [103]. To that end, structured sparse patterns have been promoted as an attractive solution, which minimizes the index storage, enables regular memory access, and enhances hardware acceleration with the simplified zero skipping patterns [49].

Besides the efficient pruning granularity and data compression techniques, the recent works aim to eliminate the sparse indexes completely. Cyclic sparse CNN [49] sparsifies the dense CNN model into the structurally cascaded sparse layers, resulting in full connectivity between the input and output, which FPGA can efficiently implement without any sparse indexes. Such a similar fashion is also feasible for element-wise sparsity. FixyFPGA [104] encodes every weight element as a fixed-weight multiplier (scaler) in hardware design. As a result, pruning out weight elements is equivalent to removing the corresponding hardware operands without introducing any index overhead.

3) Data Sharing: For efficient implementation of accelerators, different data-flow techniques have been presented to increase the reuse of data read from memories higher up the hierarchy. Since memory access cost increases as the hierarchy level increases, data reuse becomes a vital aspect of an accelerator design.

Within a DNN accelerator, the MAC modules, require three inputs (i.e., input feature map, filter weight value and the corresponding generated partial result) to generate a single output. The generated output is stored back either as a partial sum or as the result of convolution. Since contents on an FPGA are read from memories, the worst-case scenario occurs when all the three data values must be read from the memories. Without an efficient memory hierarchy and data flow design all the reads will have to come through the external DRAM (or large on-chip BRAM). Considering the sheer number of MAC operations that must be performed, these accesses will greatly affect the performance, energy efficiency and throughput of the accelerator. Additionally, since MAC operations on an FPGA can be parallelized, parallel compute paradigms including temporal and spatial architectures are explored for highly parallel solutions. Temporal architectures aim at employing the Single Instruction Multiple Data (SIMD) or the Single Instruction Multiple Thread (SIMT) execution model. CPUs and GPUs are generally categorized under the temporal architecture. Spatial architectures on the other hand, employ data flow techniques where the data in the ALUs (Arithmetic Logic Units) are shared amongst each other in a well-defined manner. FPGA based accelerators, ASIC based accelerators are common examples of such spatial architectures.

In DNN accelerators, the different data-flow techniques revolve around the reuse of 3 values (feature maps, filter weights and the generated intermediate partial sums). Authors in [1] use these reuse techniques as, feature map reuse, filter reuse and convolutional reuse. In feature map reuse, a single feature map is selected and partial sums are generated by processing the single feature with multiple filters. This allows the reuse of a single feature that has been read from the memory. Similarly, in the case of filter reuse, as shown in Fig. 7(a) (top), a single filter is selected and partial sums are generated by processing the single filter with multiple features. This scheme allows the reuse of a single filter that has been read from the memory. Finally, in case of the convolutional reuse, as shown in Fig. 7(b) (top), corresponding feature map and filter weights within a single channel are utilized in all combinations to generate the final weighted sum result for that channel. By storing and efficiently reusing data within the memory hierarchy, the number of times accesses are made to costlier memories can be greatly reduced. Additionally, authors in [1] and [105] also provide different data-flow schemes that exploit the aforementioned data reuse opportunities,

In *weight stationary* data-flow, the aim is to reuse the filter values read from the larger memory. In weight sharing, the energy cost for accessing weights is reduced. This is achieved by reading a set of weights into the register files in the PE array and holding the weights here until all the processing corresponding to that set of weights has completed. Thus,

TABLE IV
FRAMEWORKS FOR MAPPING LOW-PRECISION DNN ON FPGA

Framework	Supported Precisions	Platform	Frequency	Power	Performance
FINN [107]	Binary	Xilinx Zynq ZC706	200 MHz	8.8 W	2465 GOPS
TNN [111]	Ternary	Xilinx Virtex-7 VC709	250 MHz	13W	8360 GOPS
Unrolling TNN [112]	Ternary weights, 16-b activations	Amazon AWS F1	125 MHz	-	2500 GOPS
NN2CAM [113]	Binary	Xilinx XC7K325	100 MHz	13.2 W	337.8 GOPS
FracBNN [108]	Binary	Xilinx Zynq ZU3EG	250 MHz	4.1W	2806.9 GOPS
Filter-wise Bit Precision [114]	1-16b weights, 8-b activations	Xilinx Zynq XCZU9EG	100 MHz	-	47.7 GOPS
RL Autoencoder [115]	Ternary	Xilinx Artix-7	100 MHz	0.37 W	410 GOPS
FP-BNN [116]	Binary	Stratix-V 5SGSD8	150 MHz	26.2W	358.64 GOPS
FCA-BNN [117]	Binary	Xilinx XC7Z100	166 MHz	5.7W	3,322 GOPS

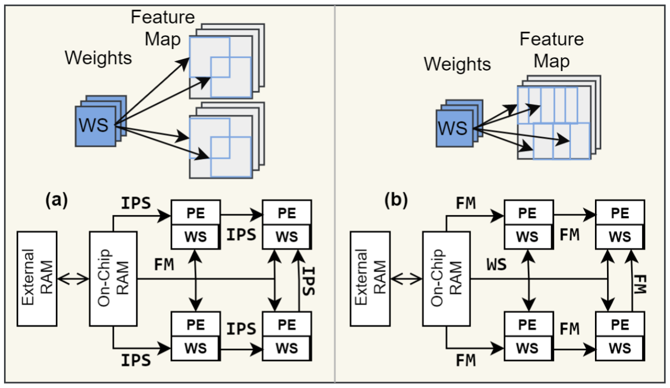


Fig. 7. Data Sharing and data-flow techniques explored. (a) Filter reuse opportunities and Weight Stationary data-flow technique exploiting the filter reuse. A single Weight (WS) is stored in the PE register files while the Intermediary Partial Sums (IPS) are streamed and the Feature Maps (FM) are broadcast. (b) Convolutional Reuse opportunities and Output Stationary data-flow technique exploiting Convolutional Reuse. The Intermediary Partial Sums (IPS) summing up to a single output is stored in the register files in the PE while the Feature Maps (FM) are streamed and the Weights (WS) are broadcast.

increasing the reuse of the weights that have been read. Additionally, while the weights are being held in the register files, the corresponding features required for the multipliers in the PE array is broadcast from the memory immediately higher in hierarchy. The partial sums to the adders is streamed while the new partial sums are accumulated across the PE array. Fig. 7(a) (bottom) represents this technique.

In *output stationary* data-flow, the aim is to cut down on the energy cost for reading and writing the partial sums to and from the memory. This is achieved by holding the partial sums that account up to a single output activation in the register file while the filters and feature activations are reordered in the PE array. One of the common approaches to accomplish this is to broadcast the weights to the PE array while the features across the PE array are streamed. There are additional delay registers to delay the data values in the PE array to make sure the required data is within the PE array until all processing related to it is completed. Fig. 7(b) (bottom) represents this technique.

In the aforementioned data-flow techniques, the memory hierarchy is divided as, external DRAM (very large memory), on chip buffer (large memory), internal PE registers (very small memory). The drawback of such a hierarchical approach is that while the register files consume very low power when compared to external DRAM, their area footprint is much more inefficient. As a result, in the *no local* reuse data-flow, the internal PE array has no register files. The PE array in this case, does not hold any data within it but travel back and forth from the on-chip memory. This results in a much greater bus traffic between the buffer and the PE array.

In case of *row stationary* data-flow,, the aim is to reuse the data in the PE. Unlike the other techniques the reuse is aimed at all types of data values within the PEs' register files. The operations can be parallelized by assigning a 1-D convolution to each PE. Each PE holds the contents from an entire row of the filter while the feature maps are streamed into the PEs. Within the PEs, MACs are computed and the partial sums are accumulated. By reusing the contents in the register file the sliding operations can be mimicked. Additionally, a PE array arranged in a 2-D matrix fashion can be used to reuse data more efficiently. This is achieved by reusing the filter row in each PE along the entire row of the PE array. Also, the feature activations are shared diagonally within the PE array. While the weight stationary and the output stationary techniques consume the least energy for the access of the weights and the partial sums, the row stationary technique could offer the least overall energy consumption [1].

4) Low Precision Implementation: Quantization optimizes parameters at the individual layer level. Replacing the floating point representation with low-bit and fixed-bit data reduces bandwidth utilization and memory storage space while also simplifying calculation and lowering the cost of each operation, albeit at the expense of accuracy. Nowadays, most of the general purpose processor architectures have adopted INT8 quantization. It has become a pre-requisite for the micro-AI devices to adopt at least INT8 bit precision, and further lower precision as the processing unit allows.

FPGAs are more popular for supporting quantized networks with less than 8 bits. BNN and TNN have 1-bit and 2-bit activations and weights which enables them to be computed using bit-wise arithmetic. These networks are well suited for FPGAs, and their memory requirements are also greatly decreased. However, BNN and TNN produce much lower accuracy on realistic datasets (i.e. ImageNet). For example, Xilinx Zynq heterogeneous FPGA platform supports BNN [106] and FINN [107] inference framework deployed BNN on a Zynq ZC706 SOC FPGA acquired 12.36M image classification per seconds allowing around 4% accuracy degradation. Reference [108] introduces fractional activations and binarizes the input layer with thermometer encoding which improves accuracy of BNNs for ImageNet.

TNN [109] uses optimized ternary value multiplication while deployed on Sakura-X FPGA and achieved 255k image classification per second with 98% average accuracy. Both [106] and [109] experimented on MNIST dataset. Reference [8] introduces a ternary hardware accelerator along with TNN training framework for efficient processing of ImageNet scale datasets. Table IV shows recent FPGA works that implemented low-precision DNNs. All of them are using 2-bits or even 1-bit data precision while still achieving comparable state-of-the-art performance. Another representative low precision work is [110] where the authors demonstrate the efficacy of low bit width design in the form of a 4-core AI chip in 7nm EUV (Extreme Ultraviolet Lithography) technology to facilitate both training and inference for a variety of fixed point and floating point precisions. Such specialized low-precision ASIC and FPGA accelerators will continue to pave the path for micro-AI inference tasks.

V. Summary and Future Trends

With regards to making networks lightweight, quantization and pruning of network parameters are the way to go. However, any sort of compression procedure is associated with some loss of accuracy. The goal is to find the best trade-off for compression against accuracy levels. To this extent, one needs to delve deeper into the quantization levels for individual networks and figure out which uniform or mixed precision configuration is the most suitable since networks appear to have different levels of sensitivity for different compression processes. The recent structured sparsification methods also provide more flexibility towards achieving the desired compression and should be used in conjunction with traditional quantization approaches. For example, AlexNet is compressed down to 50$\times$ its original size in SqueezeNet which allows it to be implemented via hardware frameworks like Eyeriss and fpgaConvNet. Similarly, ResNet-18 is shrunk down to 14$\times$ its actual size in T-DLA through ternary neural networks. Such networks can be accelerated through micro-AI devices when compressed further through the structured compression approaches. Additionally, one can formulate an optimization problem with sparsity and quantization variables as constraints and use the conventional RL-based NAS or the more recent NAS strategies as a way of coming up with the best configuration. About the deployment of the network onto resource-constrained devices, the goal is to reduce the number of MAC operations, and thus using the fast convolution techniques could be one solution. Another aspect that becomes an issue during real-time deployment is frequent memory accesses that increase the power envelope. As we have already described in Section. IV-B.3, data sharing strategies can help in reusing data from the memory in this regard.

VI. Conclusion

The overwhelming use of AI in our everyday life has brought forth the necessity of DNN accelerators. However, the issue of high computational complexity and memory constraints requires DNN accelerator designs to be energy-efficient without compromising accuracy. Thus, the creation of such architecture should begin with a generalized understanding of current optimization strategies and the potential for those strategies to open new pathways into accelerator designs. To this extent, this article surveys several optimization approaches to make software frameworks lightweight and friendly for micro-AI deployment. In addition to this, it also points out the current accelerator design optimization approaches for both FPGA and MCU implementation. Finally, DNN acceleration remains an ever interesting and expanding field of research to this day with opportunities for further innovation.

References

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[2] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks," *Future Internet*, vol. 12, no. 7, p. 113, Jul. 2020.

[3] P. R. Ovi, E. Dey, N. Roy, and A. Gangopadhyay, "ARIS: A real time edge computed accident risk inference system," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, Aug. 2021, pp. 47–54.

[4] A. Sarkar and M. Rahnemoonfar, "Visual question answering: A deep interactive framework for post-disaster management and damage assessment," in *Proc. UMBC Student Collection*, Jul. 2021, pp. 1–14.

[5] "Hey Siri: An on-device DNN-powered voice trigger for Apple's personal assistant," *Mach. Learn. J.*, vol. 1, no. 6, 2017. Accessed: Sep. 30, 2021. [Online]. Available: https://machinelearning.apple.com/2017/10/01/hey-siri.html

[6] Q. Tang, M. Sun, C.-C. Kao, V. Rozgic, and C. Wang, "Hierarchical residual-pyramidal model for large context based media presence detection," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 3312–3316.

[7] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," 2017, *arXiv:1711.07128*.

[8] Y. Chen *et al.*, "T-DLA: An open-source deep learning accelerator for ternarized DNN models on embedded FPGA," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2019, pp. 13–18.

[9] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016, *arXiv:1611.01578*.

[10] C. Liu *et al.*, "Progressive neural architecture search," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 19–34.

[11] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 8697–8710.

[12] J.-D. Dong, A.-C. Cheng, D.-C. Juan, W. Wei, and M. Sun, "DPP-net: Device-aware progressive search for Pareto-optimal neural architectures," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 517–531.

[13] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via Lamarckian evolution," 2018, *arXiv:1804.09081*.

[14] C.-H. Hsu *et al.*, "MONAS: Multi-objective neural architecture search using reinforcement learning," 2018, *arXiv:1806.10332*.

[15] Y. Zhou, S. Ebrahimi, S. Ö. Arık, H. Yu, H. Liu, and G. Diamos, "Resource-efficient neural architect," 2018, *arXiv:1806.07912*.

[16] M. Tan *et al.*, "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 2820–2828.

[17] M. S. Abdelfattah, L. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, "Best of both worlds: AutoML codesign of a CNN and its hardware accelerator," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[18] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "NAS-Bench-101: Towards reproducible neural architecture search," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 7105–7114.

[19] M. Hosseini, M. Ebrahimabadi, A. N. Mazumder, H. Homayoun, and T. Mohsenin, "A fast method to fine-tune neural networks for the least energy consumption on FPGAs," in *Proc. UMBC Student Collection*, 2021, pp. 1–6.

[20] M. Hosseini and T. Mohsenin, "QS-NAS: Optimally quantized scaled architecture search to enable efficient on-device micro-AI," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, early access, Nov. 15, 2021, doi: 10.1109/JETCAS.2021.3127932.

[21] B. Hassibi and D. G. Stork, *Second Order Derivatives for Network Pruning: Optimal Brain Surgeon*. Burlington, MA, USA: Morgan Kaufmann, 1993.

[22] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," 2018, *arXiv:1803.03635*.

[23] J. Frankle, G. K. Dziugaite, D. M. Roy, and M. Carbin, "Stabilizing the lottery ticket hypothesis," 2019, *arXiv:1903.01611*.

[24] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," 2015, *arXiv:1506.02626*.

[25] A. N. Gomez, I. Zhang, K. Swersky, Y. Gal, and G. E. Hinton, "Learning sparse networks using targeted dropout," 2019, *arXiv:1905.13678*.

[26] S. Ye *et al.*, "Progressive DNN compression: A key to achieve ultra-high weight pruning and quantization rates using ADMM," 2019, *arXiv:1903.09769*.

[27] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[28] A. Makhzani and B. Frey, "Winner-take-all autoencoders," 2014, *arXiv:1409.2752*.

[29] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 1389–1397.

[30] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," 2017, *arXiv:1710.01878*.

[31] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," 2019, *arXiv:1902.09574*.

[32] S. Park, J. Lee, S. Mo, and J. Shin, "Lookahead: A far-sighted alternative of magnitude-based pruning," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–20.

[33] S. Yin, Z. Jiang, J.-S. Seo, and M. Seok, "XNOR-SRAM: In-memory computing SRAM macro for binary/ternary deep neural networks," *IEEE J. Solid-State Circuits*, vol. 55, no. 6, pp. 1733–1743, Jun. 2020.

[34] Z. Jiang, S. Yin, J.-S. Seo, and M. Seok, "C3SRAM: An in-memory-computing SRAM macro based on robust capacitive coupling computing mechanism," *IEEE J. Solid-State Circuits*, vol. 55, no. 7, pp. 1888–1897, Jul. 2020.

[35] S. Yin *et al.*, "Monolithically integrated RRAM- and CMOS-based in-memory computing optimizations for efficient deep learning," *IEEE Micro*, vol. 39, no. 6, pp. 54–63, Nov. 2019.

[36] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., 2016, pp. 2074–2082.

[37] D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti, and J.-S. Seo, "An 8.93 TOPS/W LSTM recurrent neural network accelerator featuring hierarchical coarse-grain sparsity for on-device speech recognition," *IEEE J. Solid-State Circuits*, vol. 55, no. 7, pp. 1877–1887, Jul. 2020.

[38] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*.

[39] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables," *J. Roy. Statist. Soc. B, Statist. Methodol.*, vol. 68, no. 1, pp. 49–67, 2006.

[40] T. Zhang *et al.*, "StructADMM: A systematic, high-efficiency framework of structured weight pruning for DNNs," 2018, *arXiv:1807.11091*.

[41] T.-W. Chin, R. Ding, C. Zhang, and D. Marculescu, "Towards efficient model compression via learned global ranking," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 1518–1528.

[42] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: Automl for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 784–800.

[43] C. Gamanayake, L. Jayasinghe, B. K. K. Ng, and C. Yuen, "Cluster pruning: An efficient filter pruning method for edge AI vision applications," *IEEE J. Sel. Topics Signal Process.*, vol. 14, no. 4, pp. 802–816, May 2020.

[44] J. Meng, L. Yang, X. Peng, S. Yu, D. Fan, and J.-S. Seo, "Structured pruning of RRAM crossbars for efficient in-memory computing acceleration of deep neural networks," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 5, pp. 1576–1580, May 2021.

[45] W. Hua, Y. Zhou, C. De Sa, Z. Zhang, and G. E. Suh, "Channel gating neural networks," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 1886–1896.

[46] G. Shomron, R. Banner, M. Shkolnik, and U. Weiser, "Thanks for nothing: Predicting zero-valued activations with lightweight convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.* Glasgow, U.K.: Springer, 2020, pp. 234–250.

[47] Z. Su, L. Fang, W. Kang, D. Hu, M. Pietikäinen, and L. Liu, "Dynamic group convolution for accelerating convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.* Glasgow, U.K.: Springer, 2020, pp. 138–155.

[48] B. E. Bejnordi, T. Blankevoort, and M. Welling, "Batch-shaping for learning conditional channel gated networks," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–14.

[49] M. Hosseini *et al.*, "Cyclic sparsely connected architectures for compact deep convolutional neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 10, pp. 1757–1770, Oct. 2021.

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

[51] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Netw.*, vol. 100, pp. 49–58, Apr. 2018.

[52] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016, *arXiv:1606.06160*.

[53] M. Courbariaux, Y. Bengio, and J. P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 3123–3131.

[54] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengi, "Binarized neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29. 2016, pp. 1–17.

[55] Y. Li, X. Dong, and W. Wang, "Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks," 2019, *arXiv:1909.13144*.

[56] D. Wan *et al.*, "TBN: Convolutional neural network with ternary inputs and binary weights," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 315–332.

[57] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhad, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.* Amsterdam, The Netherlands: Springer, 2016, pp. 525–542.

[58] J. Jin, C. Liang, T. Wu, L. Zou, and Z. Gan, "KDLSQ-BERT: A quantized bert combining knowledge distillation with learned step size quantization," 2021, *arXiv:2101.05938*.

[59] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of ConvNets via differentiable neural architecture search," 2018, *arXiv:1812.00090*.

[60] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 8612–8620.

[61] M. Rusci, M. Fariselli, A. Capotondi, and L. Benini, "Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers," in *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Ghent, Belgium: Springer, 2020, pp. 296–308.

[62] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer, "HAWQ: Hessian aware quantization of neural networks with mixed-precision," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 293–302.

[63] Z. Dong *et al.*, "HAWQ-v2: Hessian aware trace-weighted quantization of neural networks," 2019, *arXiv:1911.03852*.

[64] Z. Yao *et al.*, "HAWQ-V3: Dyadic neural network quantization," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 11875–11886.

[65] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 578–594.

[66] D. P. Kingma and P. Dhariwal, "Glow: Generative flow with invertible 1×1 convolutions," 2018, *arXiv:1807.03039*.

[67] TensorFlow. *XLA: Optimizing Compiler for Machine Learning*. Accessed: Oct. 10, 2021. [Online]. Available: https://www.tensorflow.org/xla

[68] S. Cyphers *et al.*, "Intel nGraph: An intermediate representation, compiler, and executor for deep learning," 2018, *arXiv:1801.08058*.

[69] NVIDIA. *NVIDIA/TensorRT: TensorRT is a C++ Library for High Performance Inference on NVIDIA GPUs and Deep Learning Accelerators*. Accessed: Sep. 30, 2021. [Online]. Available: https://github.com/NVIDIA/TensorRT

[70] V. Kathail, "Xilinx Vitis unified software platform," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2020, pp. 173–174.

[71] C. N. Coelho *et al.*, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors," *Nature Mach. Intell.*, vol. 3, pp. 675–686, Jun. 2021.

[72] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 6848–6856.

[73] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough, "SpArSe: Sparse architecture search for CNNs on resource-constrained microcontrollers," 2019, *arXiv:1905.12107*.

[74] T. Lawrence and L. Zhang, "IoTNet: An efficient and accurate convolutional neural network for IoT devices," *Sensors*, vol. 19, no. 24, p. 5541, Dec. 2019.

[75] M. Rusci, A. Capotondi, and L. Benini, "Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers," 2019, *arXiv:1905.13082*.

[76] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," 2020, *arXiv:2007.10319*.

[77] R. David *et al.*, "TensorFlow lite micro: Embedded machine learning on TinyML systems," 2020, *arXiv:2010.08678*.

[78] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs," 2018, *arXiv:1801.06601*.

[79] A. N. Mazumder *et al.*, "Automatic detection of respiratory symptoms using a low power multi-input CNN processor," *IEEE Design Test*, early access, May 11, 2021, doi: 10.1109/MDAT.2021.3079318.

[80] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[81] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Automated mapping of convolutional neural networks on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 291–292.

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

[83] M. Vestias, R. P. Duarte, J. T. de Sousa, and H. Neto, "Lite-CNN: A high-performance architecture to execute CNNs in low density FPGAs," in *Proc. 28th Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2018, pp. 1–4.

[84] Y. Ma, T. Zheng, Y. Cao, S. Vrudhula, and J.-S. Seo, "Algorithm-hardware co-design of single shot detector for fast object detection on FPGAs," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[85] S. Zhang, J. Cao, Q. Zhang, Q. Zhang, Y. Zhang, and Y. Wang, "An FPGA-based reconfigurable CNN accelerator for YOLO," in *Proc. IEEE 3rd Int. Conf. Electron. Technol. (ICET)*, May 2020, pp. 74–78.

[86] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on FPGA," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.

[87] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 75–84.

[88] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A power-efficient recurrent neural network accelerator," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 21–30.

[89] A. N. Mazumder, H.-A. Rashid, and T. Mohsenin, "An energy-efficient low power LSTM processor for human activity monitoring," in *Proc. IEEE 33rd Int. Syst. Chip Conf. (SOCC)*, Sep. 2020, pp. 54–59.

[90] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," 2013, *arXiv:1312.5851*.

[91] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with *fbfft*: A GPU performance evaluation," 2014, *arXiv:1412.7580*.

[92] S. Chetlur *et al.*, "CuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.

[93] S. Lin *et al.*, "FFT-based deep learning deployment in embedded systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1045–1050.

[94] T. Highlander and A. Rodriguez, "Very efficient training of convolutional neural networks using fast Fourier transform and overlap-and-add," 2016, *arXiv:1601.06815*.

[95] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating convolutional neural network with FFT on embedded hardware," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 9, pp. 1737–1749, Sep. 2018.

[96] A. Page, A. Jafari, C. Shea, and T. Mohsenin, "SPARCNet: A hardware accelerator for efficient deployment of sparse convolutional networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 31:1–31:32, May 2017, doi: 10.1145/3005448.

[97] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021.

[98] H. Wang, W. Liu, T. Xu, J. Lin, and Z. Wang, "A low-latency sparse-Winograd accelerator for convolutional neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 1448–1452.

[99] T. Yang, Y. Liao, J. Shi, Y. Liang, N. Jing, and L. Jiang, "A Winograd-based CNN accelerator with a fine-grained regular sparsity pattern," in *Proc. 30th Int. Conf. Field-Program. Logic Appl. (FPL)*, Aug. 2020, pp. 254–261.

[100] D. Wu, X. Fan, W. Cao, and L. Wang, "SWM: A high-performance sparse-Winograd matrix multiplication CNN accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 5, pp. 936–949, May 2021.

[101] X. Liu, Y. Chen, C. Hao, A. Dhar, and D. Chen, "WinoCNN: Kernel sharing Winograd systolic array for efficient convolutional neural network acceleration on FPGAs," in *Proc. IEEE 32nd Int. Conf. Appl.-Specific Syst., Archit. Process. (ASAP)*, Jul. 2021, pp. 258–265.

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

[103] S. Wang *et al.*, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 11–20.

[104] J. Meng, S. K. Venkataramanaiah, C. Zhou, P. Hansen, P. Whatmough, and J.-S. Seo, "FixyFPGA: Efficient FPGA accelerator for deep neural networks with high element-wise sparsity and without external memory access," in *Proc. 31st Int. Conf. Field-Program. Logic Appl. (FPL)*, Aug./Sep. 2021, pp. 9–16.

[105] R. Wu, X. Guo, J. Du, and J. Li, "Accelerating neural network inference on FPGA-based platforms—A survey," *Electronics*, vol. 10, no. 9, p. 1025, Apr. 2021.

[106] R. Zhao *et al.*, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 15–24.

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

[108] Y. Zhang, J. Pan, X. Liu, H. Chen, D. Chen, and Z. Zhang, "FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2021, pp. 171–182.

[109] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Petrot, "Ternary neural networks for resource-efficient AI applications," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, May 2017, pp. 2547–2554.

[110] A. Agrawal *et al.*, "A 7 nm 4-core AI chip with 25.6 TFLOPS hybrid FP8 training, 102.4 TOPS INT4 inference and workload-aware throttling," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 64. Feb. 2021, pp. 144–146.

[111] A. Prost-Boucle, A. Bourge, F. Petrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on FPGA," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–7.

[112] S. Tridgell *et al.*, "Unrolling ternary neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 4, pp. 1–23, Nov. 2019.

[113] P. Jokic, S. Emery, and L. Benini, "NN2CAM: Automated neural network mapping for multi-precision edge processing on FPGA-based cameras," 2021, *arXiv:2106.12840*.

[114] A. Maki, D. Miyashita, K. Nakata, F. Tachibana, T. Suzuki, and J. Deguchi, "FPGA-based CNN processor with filter-wise-optimized bit precision," in *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, Nov. 2018, pp. 47–50.

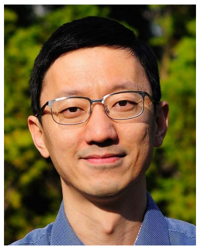
[115] N. K. Manjunath, A. Shiri, M. Hosseini, B. Prakash, N. R. Waytowich, and T. Mohsenin, “An energy efficient EdgeAI autoencoder accelerator for reinforcement learning,” *IEEE Open J. Circuits Syst.*, vol. 2, pp. 182–195, 2021.
[116] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, “FP-BNN: Binarized neural network on FPGA,” *Neurocomputing*, vol. 275, pp. 1072–1086, Jan. 2018.
[117] J. Gao, Y. Yao, Z. Li, and J. Lai, “FCA-BNN: Flexible and configurable accelerator for binarized neural networks on FPGA,” *IEICE Trans. Inf. Syst.*, vol. 104, no. 8, pp. 1367–1377, 2021.

Arnab Neelim Mazumder (Graduate Student Member, IEEE) received the B.S. degree from the Chittagong University of Engineering and Technology (CUET), Bangladesh. He is currently pursuing the Ph.D. degree with the Computer Science and Electrical Engineering Department, The University of Maryland Baltimore County, MD, USA. His research interests focus on deep neural networks, on-device AI, FPGA and ASIC designs of hardware accelerators, and low-power embedded systems.

Jian Meng (Graduate Student Member, IEEE) received the B.S. degree from Portland State University, Portland, USA, in 2019. He is currently pursuing the Ph.D. degree with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA. His current research focuses on the deep neural network compression optimization, hardware–software co-design with neuromorphic hardware acceleration, and event-based object detection.

Hasib-Al Rashid (Student Member, IEEE) received the B.S. degree from the Chittagong University of Engineering and Technology (CUET), Bangladesh. He is currently pursuing the Ph.D. degree with the Computer Science and Electrical Engineering Department, The University of Maryland Baltimore County, MD, USA. His research interests focus on multimodal machine learning and deep learning, model optimization for edge AI, and software hardware co-design.

Utteja Kallakuri received the M.S. degree from The University of Maryland Baltimore County (UMBC), MD, USA, where he is currently pursuing the Ph.D. degree with the Computer Science and Electrical Engineering Department. His research interests revolve around FPGA and ASIC designs of hardware accelerators, deep neural networks, and low-power embedded systems.



Xin Zhang (Senior Member, IEEE) received the B.S. degree in electronics engineering from Xi’an Jiaotong University, Xi’an, China, in 2003, and the Ph.D. degree in microelectronics from Peking University, Beijing, China, in 2008. In 2008, he joined the Institute of Industrial Science, The University of Tokyo, Tokyo, Japan, as a Project Researcher. In 2012, he was a Visiting Scholar with the University of California at Berkeley and then a Project Research Associate with the Institute of Industrial Science, The University of Tokyo. In 2013, he was with the Institute of Microelectronics (IME), Agency for Science, Technology and Research (A*STAR), Singapore, as a Scientist. Since 2014, he has been a Research Staff Member with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. Since 2021, he has been an Adjunct Professor with the Department of Electrical Engineering, Columbia University, New York, NY, USA. He has authored or coauthored over 50 technical articles and has over 30 filed or issued patents. His research interests include analog circuits, power management circuits, dc–dc converters, ac–dc converters, power devices, magnetics, machine learning hardware/accelerators, computer system architecture, and server system power delivery/packaging/cooling. He is currently serving as a Technical Program Committee Member for the Applied Power Electronics Conference (APEC), the IEEE VLSI Symposium on Technology and Circuits, the IEEE Custom Integrated Circuits Conference (CICC), and the IEEE International Solid-State Circuits Conference (ISSCC). He is also an Organizing Committee Member for the IBM IEEE CAS/EDS–AI Compute Symposium. He is also serving as a Technical Advisory Board Member for the Analog-Mixed Signal Circuits, Systems, and Devices (AMS-CSD), Semiconductor Research Corporation (SRC). He has served as a Guest Editor for IEEE JOURNAL ON EMERGING AND SELECTED TOPICS IN CIRCUITS AND SYSTEMS (JETCAS) and IEEE SOLID-STATE CIRCUITS LETTERS (SSC-L).

Jae-Sun Seo (Senior Member, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2001, and the M.S. and Ph.D. degrees in electrical engineering from the University of Michigan, Ann Arbor, MI, USA, in 2006 and 2010, respectively. From 2010 to 2013, he was with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA, where he worked on cognitive computing chips under the DARPA SyNAPSE Project and energy-efficient integrated circuits for high-performance processors. In 2014, he joined the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA, where he is currently an Associate Professor. In 2015, he joined Intel Circuits Research Labs, Hillsboro, OR, USA, as a Visiting Faculty. His current research interests include efficient hardware design of machine learning and neuromorphic algorithms and integrated power management. He was a recipient of the Samsung Scholarship from 2004 to 2009, the IBM Outstanding Technical Achievement Award in 2012, the NSF CAREER Award in 2017, and the Intel Outstanding Researcher Award in 2021.

Tinoosh Mohsenin (Senior Member, IEEE) received the M.Sc. degree in electrical and computer engineering from Rice University in 2004 and the Ph.D. degree in electrical and computer engineering from the University of California at Davis in 2010. She is currently an Associate Professor with the Department of Computer Science and Electrical Engineering, The University of Maryland Baltimore County, where she is also the Director of the Energy Efficient High Performance Computing Lab. She has authored or coauthored over 130 peer-reviewed journals and conference publications. Her research focus is on designing energy efficient embedded processors for machine learning and signal processing, knowledge extraction techniques for autonomous systems, wearable smart health monitoring, and embedded big data computing.