2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00098

TermiNETor: Early Convolution Termination for Efficient Deep Neural Networks

Uday Mallappa, Pranav Gangwar, Behnam Khaleghi, Haichao Yang, Tajana Rosing
ECE & CSE Departments, UC San Diego, La Jolla, CA 92093, USA
Email: {umallapp, pgangwar, bkhaleghi, hcyang, tajana}@ucsd.edu

***Abstract*—The efficiency of convolutional neural networks (CNNs) is dictated by the computationally intensive convolution operations. Thereby, extensive research has sought to alleviate the cost of convolution. In this work, we opt to skipping the *ineffectual outputs*, which can be used along with the other approaches such as quantization and pruning. Contrary to the previous works that skip ineffectual inputs, i.e., zero activations and weights, we predict and skip the ineffectual outputs by exploiting two key attributes of the state-of-the-art CNN architectures: ReLU (which filters out the negative outputs) is the widely-used activation function, and max-pooling (which discards the non-max outputs) is the commonly used down-sampling layer. To realize that, we propose** TermiNETor**, which breaks the convolution to bitwise shift-and-add operations. During the inference, weights are processed bit by bit, whereby** TermiNETor **uses a simple heuristic to predict whether the final output will be ineffectual. Upon prediction,** TermiNETor **terminates the convolution for ineffectual outputs. We propose a calibration flow that takes the prediction-based forward propagation into account and realizes more accurate predictions. In addition, we propose a novel hardware architecture to take advantage of early termination by feeding new convolution to the released processing elements. We demonstrate the effectiveness of** TermiNETor **on various networks and datasets. Experimental results show that our framework achieves up to 1.7× reduction of operation count compared to non-skipping baseline without accuracy degradation.** TermiNETor **accelerator, augmented with the skipping flow, improves the average energy efficiency by 3.84× over the zero weight/input skipping accelerator SCNN, and by 1.98× over FuseKNA which skips zero inputs as well as repetitive computations in a bit-serial manner. The proposed architecture exhibits good throughput and energy consumption scalability by increasing the processing elements, meaning that it can also target high-performance applications while preserving the energy efficiency.**

I. Introduction

A plethora of studies have aimed to alleviate the computation and energy cost of deep neural networks (DNNs) by hardware-friendly algorithmic innovations such as weight or filter pruning [1], [2], [3], and weight or activation quantization [4], [5], [6], [7], [8]. More strict hardware techniques skip the operations with ineffectual operands, i.e., zero weights and/or input activations[1] [9], [10], [11]. Some studies even exploit bit-granular sparsity in the weights or inputs by breaking the operations into bit level and skipping if either operands' bit is zero [12], [13].

The sparsity-aware accelerators aim to skip the zero weights and/or inputs, where the zero inputs are produced by the preceding ReLU layer (Fig. 1). Skipping the ineffectual outputs is more beneficial than the ineffectual inputs as the number of zero outputs is more than zero inputs (most of the zero outputs are naturally discarded from the next layer's inputs after passing through the pooling). Several studies opt to skip the computation of negative outputs. The majority of these works split the input or weight bits into two significant and insignificant parts [14], [15], [16], [17]. The result of computation using the significant bits creates a mask that reveals which outputs need full-bit computation. Instead of splitting into bits, the work in [18] partitions the sorted weights into small groups and selects a representative weight from each one. The result of convolution (dot-product) with these select weights and the corresponding inputs determines the potentially negative outputs.

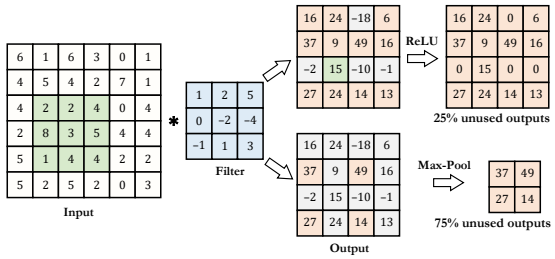

Fig. 1. Ineffectual outputs due to ReLU and max pooling.

In these studies, the prediction procedure, i.e., splitting of bits or partitioning the weights, is determined *statically* while the inputs' statistics can vary significantly from image to image (e.g., [8] points out this observation and proposes dynamic quantization based on input sensitivity). Thus, such prediction approaches have to be pessimistic to retain the accuracy, which results in low performance gain, e.g., $\leq 30\%$ in relatively complex networks [18], [16], [17]. Better improvement ($\sim$60%) is reported in [15]. However, [15] uses non-complex networks as well as high bitwidth for the weights and activations, which make the negative output prediction simpler. Also, when the bitwidths are already low, the impact of LSB bits becomes substantial, i.e., the likelihood of output sign change (misprediction) by continuing the operation on the four LSB bits of 8-bits weights is higher than eight LSB bits of 16-bits weights (as $2^{-5} + \cdots > 2^{-9} + \cdots$).

In this work, we propose algorithmic innovation and hardware support, dubbed TermiNETor, for *dynamically* predicting and skipping the ineffectual outputs, including negative

[1]We will refer to input activations as *inputs*, and output activations as *outputs*.

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00098

outputs preceding the ReLU layers and non-max outputs preceding the max-pooling layers. To this end, we leverage bit-serial operation whereby a convolution, represented as dot-product of flattened activations and weights $'A \cdot W'$, is split to $\sum A \cdot (W_i \times 2^i)$ over the weight bits W_i. After processing an i^{th} bit of the weight vector, TermiNETor checks if the *partial output*[1] falls behind a designated threshold. This threshold varies for bit indexes; partial output produced using the initial (MSB) weight bits needs a more conservative threshold to ensure safe termination since the probability of a negative-to-positive flip of the output is higher as we move from MSB towards the LSB. The threshold also varies between the layers of a network.

There are three principal differences between TermiNETor and previous works: (i) TermiNETor uses bit-slicing and can flexibly terminate the operations at any index upon prediction, and notably, (ii) TermiNETor *operates in the granularity of output level*, meaning that it can terminate the operations for *a small group of output pixels* independent of the others, and (iii) Computations used for mask generation are reused for effective or useful convolutions; obliterating the computation overhead of standalone mask generation. Nevertheless, realizing a non-pessimistic threshold mechanism that universally works for various inputs is challenging. The algorithmic contribution of TermiNETor is to make such a partial-output-based prediction mechanism viable. Moreover, architectural support to take utmost performance and energy gain of these fine-grained terminations with high resource utilization is as critical, which is the focus of TermiNETor's hardware novelty.

II. Ineffectual Output Skipping

Convolutional layers are the core building block of DNNs and contribute to $> 99\%$ of these networks computations [19]. These layers are followed by a nonlinear activation function, which has significant impact on the accuracy of these networks. The rectified linear unit (ReLU), simply defined as $f(a) = \max(0, a)$, is the widely-used and most successful activation function and yields better and more consistent performance than others such as hyperbolic tangent or augmented alternatives like leaky ReLU [20]. The ReLU function introduces sparse representations, i.e., zero output activation values, into the network and steers the observation that computations spent for the convolutions that generate negative outputs are unnecessary or *ineffectual*.

Besides the convolution and nonlinear activation layers, a downsampling operation is used intermittently to reduce the dimensionality of the output activation map. The downsampling is achieved via either a patch-wise (or window-wise) max-pooling or an average-pooling layer. In the former case, the pooled activation map highlights the largest feature of the patch and has been found to work better than average pooling for computer vision tasks like image classification [21]. In a 2×2 patch, three out of the four activations are discarded through the pooling operation. Therefore, another key observation is that 75% of the computations spent for the convolution operations that are preceded by a max-pooling layer are ineffectual.

[1]A *partial output* is the dot-product of the whole activation vector and sub-bits of the whole weight vector (while a *partial sum* refers to *sub-vectors* dot-product result).

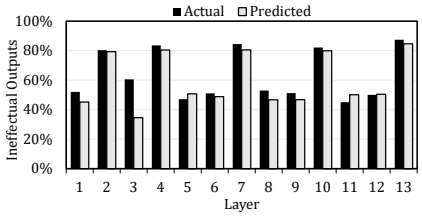

Fig. 2. Ineffectual outputs of VGG16 on CIFAR-100 images.

Fig. 2 shows the percentage of ineffectual outputs for different layers of the VGG16 network on CIFAR-100 images. In this case, ineffectual outputs account for 61.8% of the produced outputs. The figure also shows the number of predicted ineffectual outputs by TermiNETor after using the first three bits (towards MSB) of the weight. Certain convolution layers are followed by a max-pooling layer, wherein $> 80\%$ of the layer outputs become ineffectual. The percentage of ineffectual outputs increases with the image size since a major portion of such images consists of a background, which is filtered out by the convolution. For instance, in the same VGG16 network, 77.4% of the outputs become ineffectual for 224×224 ILSVRC-2012 images. We exploit the presence of ineffectual outputs which are useless for the next layer's operations to improve the inference performance by skipping the computations that lead to such outputs. This is accomplished by predicting the ineffectual outputs using the proposed *weight-bit-serial* TermiNETor framework and architectural support to skip them with the fine granularity for maximal performance gain.

A. Inference Framework

In TermiNETor, the weights and input activations are quantized to standard 8-bits INT8 representation, bit 7 (MSB or sign) to bit 0 (LSB). Nevertheless, our bitwise prediction can be indeed employed with any $>$ 1-bit quantized model (though the efficiency gain will vary). Note that previous work such as [15] use over-provisioned bitwidths (e.g., 16) which makes the prediction much simpler. For instance, they can split the activations and/or weights into two 8-bits pieces and accurately predict the output sign by using 8-bits of each (75% effective operation reduction) since the lower 8-bits, in most cases, only adjusts the precision of the output rather than flipping the sign. In contrast, TermiNETor operates on the weights *bit by bit* and uses the complete 8-bit activations for each weight bit. Generating one convolution output begins with the MSB weight bit (bit 7) and continues till bit 0 of the weight.

Since only one bit of weights are processed at a time, the partial output is computed using the typical shift-and-add multiplication similar to bit-serial-based works [12], [13]. Regardless of output skipping, an advantage of such an architecture over conventional fixed-point designs is supporting

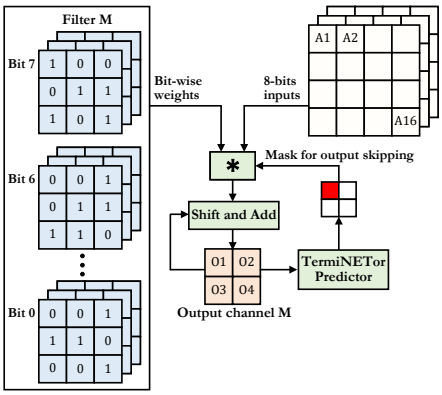

Fig. 3. Early output prediction of TermiNETor.

arbitrary and layer-wise weight quantization [22]. A typical convolution takes eight iterations. The key idea is to use the partially computed outputs after C cycles of the bitwise shift-and-add operations and predict (i) if the final output activation is leaning towards a negative value, and (ii) if the convolution output ends up unused because of a succeeding 2×2 max-pooling window. Accordingly, if we predict an ineffectual output at the $C = 3^{\text{rd}}$ cycle (i.e., after bit index 5), we can save five cycles by terminating the subsequent bitwise computations. Note that consuming three bits for prediction is principally different from 3-bits weight quantization. Unlike quantization, here, the model still operates with 8-bits weights for useful outputs and fewer bits are only used for ineffectual outputs. Our bitwise prediction can be employed with any >1-bit quantized model (though the efficiency gain will vary). Fig. 3 depicts TermiNETor's inference scheme. TermiNETor uses a masking operator to skip the convolutions that lead to ineffectual outputs. The mask is being updated by adding more skippable outputs after processing a new weight bit index.

Formulation: Consider the bitwise generation of a convolution output activation (pixel) $O^{p,q}$ at location (p, q) of a certain output channel M (being produced by filter M). Each convolution is multiplication of a three-dimensional filter with the corresponding part of the input feature map, which can be shown as a flattened dot-product (vector-vector multiplication). We start with the $i = 7^{\text{th}}$ (MSB) of the weight and build up the partial output using the shift-and-add operation (the first bit determines the sign). Notice that in INT8 representation, the weights are considered 8-bits integers.

$$O^{p,q} = A \cdot W[7:i] = \sum_{i=7}^{0} A \cdot (W^{(i)} \times 2^i \times -1^{i==7})$$

If $i \leq 8-T$ (i.e., after processing T weight indexes without skipping), we start masking the outputs that are predicted to be negative. Nevertheless, the partial output $O^{p,q}$ is incomplete and its sign might be changed if more indexes are processed. Fig. 4 demonstrates such a scenario. After the first bit, $A \cdot W[7:7]$ is negative but it increases and eventually its sign flips after using four bits ($A \cdot W[7:4]$).

We observed that adding a $\Delta^{p,q} \geq 0$ term that accounts

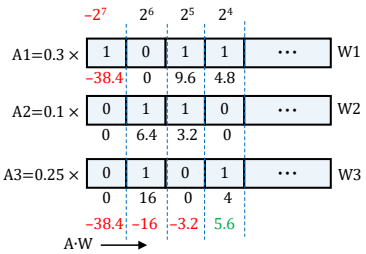

Fig. 4. An example of negative-to-positive sign flip.

Algorithm 1: TermiNETor calibration framework

Require: Pretrained weights Θ_0, train_data = $\{X, y\}$
model $\leftarrow$ SGD(Θ_0, X, y)
for each epoch in total_epochs **do**
  for each layer ℓ in model.layers() **do**
    for each bit_idx in $[7, \cdots, 0]$ **do**
      if bit_idx $< 8 - T^{\ell}$ **then**
        if $\ell.index = 1$ **then**
          Y^{ℓ} += ℓ.forward(X)
        end if
        if $\ell.index > 1$ **then**
          Y^{ℓ} += ℓ.forward($Y^{\ell-1}$)
        end if
      end if
      mask$^{\ell}$ $\leftarrow$ genMask(Y^{ℓ}, bit_idx, ℓ.pool)
      if bit_idx $\geq 8 - T^{\ell}$ **then**
        if ℓ.index == 1 **then**
          Y^{ℓ} += ℓ.forward(X, mask$^{\ell}$)
        end if
        if $l.index > 1$ **then**
          Y^{ℓ} += ℓ.forward($Y^{\ell-1}$, mask$^{\ell}$)
        end if
      end if
    end for
  end for
  $\Theta_{t+1} \leftarrow$ SGD(Θ_t)
end for

for the residual convolution yields better prediction. We can think of $\Delta^{p,q}$ as an inexpensive positive bias term that is easy to compute and is also spatially aware of the input activation map. Thus, we have:

$$\text{if } i \leq 8-T \text{ and } O^{p,q}+\Delta^{p,q} < \text{threshold} \Rightarrow mask^{p,q} = 1 \quad (1)$$

Essentially, $\Delta^{p,q}$ ensures that the partial output is negative enough to avoid potential sign change. Also for the convolution layers that are followed by a max-pooling, we mask the outputs that are predicted to be ineffectual (i.e., are smaller than the maximum element of the pooling window $\mathcal{P}$):

$$\text{if } i \leq 8-T \text{ and } (p,q) \neq \text{argmax}\,\mathcal{P}_{(p,q)} \Rightarrow mask^{p,q} = 1 \quad (2)$$

B. Model Calibration

We observe accuracy degradation when the inference procedure of TermiNETor, explained above, is performed using a baseline pretrained model. This is because the pretrained model is not tuned for the imposed inference approximations; similar to, e.g., quantization wherein the weights require post-tuning. Also, the hyperparameters T (that allows skipping after a particular weight bit index) and Δ (prediction bias) are layer-dependent and requires a rigorous design-space exploration.

To address the former issue, we calibrate the pretrained model to compensate the approximations imposed by

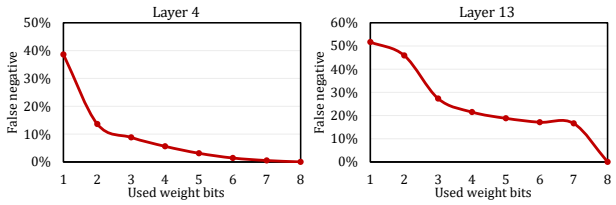

Fig. 5. Percentage of false negative for two layers of a pretrained VGG16 model on CIFAR-100 dataset. The X-axis is the number of used weight bits.

TermiNETor's inference, summarized by Algorithm 1. During the calibration process, the model tunes the weights toward the solution space that is highly sparse in the lower portions of the 8-bits weights. Such a weight calibration makes the lower portion of the weights to be less impactful as compared to the upper portion. Notice that the calibration procedure is different from quantization. In quantization, cutting the discarded LSB bits does not affect the model accuracy. However, in our case, the LSB bits still play a major role to retain the model accuracy (which we further analyze in Section IV); *by calibration, only the probability of negative-to-positive sign flip is decreased* Algorithm 1 processes the weight bits one by one (line 4) and updates the forward propagation outputs of the layer ℓ (denoted by Y^{ℓ}) after each bit. If more T^{ℓ} or more bits are consumed, the algorithm applies the mask on the forward propagation values, which replaces the masked values with 0.

C. Hyperparameters Exploration

Minimum prediction index T: In bit-serial processing of the weights, the confidence of outputs estimation improves as we move from the MSB bit towards the LSB bit. The key metric to evaluate the confidence of early termination is the *false negative* percentage, i.e., the percentage of outputs that are predicted negative but would turn to positive if all the weight bits are used. As shown in Fig. 5, the percentage of false negatives is layer-dependent, and it decreases as we consume more bits. Therefore, the value of T is first derived from the plots of the pretrained networks, and then we use model calibration to mitigate the accuracy loss. For example, $T = 4$ might be acceptable for layer 4 (the left figure) but it can incur accuracy loss for layer 13.

Mask generation and bias term Δ: The partial output prediction mechanism uses a simple addition and comparison. It takes the partial output sum, adds a pre-computed value Δ and checks if the result is greater than a desired threshold. The pre-computed value Δ *is unique for each output sum*, while the threshold is the same for all the outputs of a layer. Both the parameters are computed offline and are loaded for comparison during run-time. To ensure that minimum number of outputs will use all the weights' bits, the network is calibrated to make the weights' bits more sparse, i.e., more bits in a weight parameter are zero. This helps in obtaining a threshold value that gives a higher prediction confidence corresponding to each weight bit. As we showed in the example of Fig. 4, zero bits do not affect the partial output. At the same time, the more zeroes present in the weight, the higher the probability of terminating

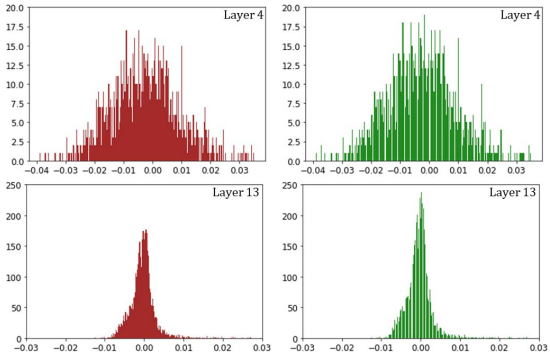

Fig. 6. Weight histograms before model calibration (left) and after (right) for two layers of VGG16.

that computation earlier. Fig. 6 shows the impact of model calibration on the distribution of two representative layers of VGG16. The histogram is sparse around certain values which have large number of '1' bits as the frequency of such values is minimized in the calibrated model.

For every layer of the network, the input activations corresponding to the training data set are analyzed (offline) by taking their product with the kernel weights to determine an appropriate bias $\Delta^{(p,q)}$ for each output of the layer. After this, the threshold value (see Equation (1)) is obtained heuristically by analyzing the output predictions considering the obtained $\Delta^{(p,q)}$. We performed multiple experiments with the partial output prediction mechanism and realized that the prediction mechanism should not be employed as soon as the partial outputs are generated, since initially the confidence in the prediction is quite low and degrades the accuracy.

To improve hardware utilization, TermiNETor also implements group-termination, whereby processing a chunk of adjacent outputs (pixels with the same (i, j) position but different channels) are terminated together. We elaborate group-termination in Section III.

III. TermiNETor Architecture

A. Overview

The efficiency of the TermiNETor architecture is critical since, on the one hand, the processing elements should be able to operate on generating a new output upon early-terminating a particular output (rather than simply staying idle to simplify dataflow), and on the other hand, data reuse is a key efficiency factor which may contradict the first desideratum. This section elaborates on how TermiNETor can meet both criteria.

Fig. 7 shows the overview of TermiNETor datapath. It comprises a two-dimensional array of processing elements (PEs). The baseline architecture is an 8×16 array to consume a comparable power to previous works, but the architecture is scalable as we examine in Section IV. Each input lane broadcasts an input sub-image brick by brick to all the PEs in the same row (we use *lane* to distinguish between logical and physical memory entities). A *brick* (input, weight, or output) is composed of consecutive elements in the Z axis, e.g.,

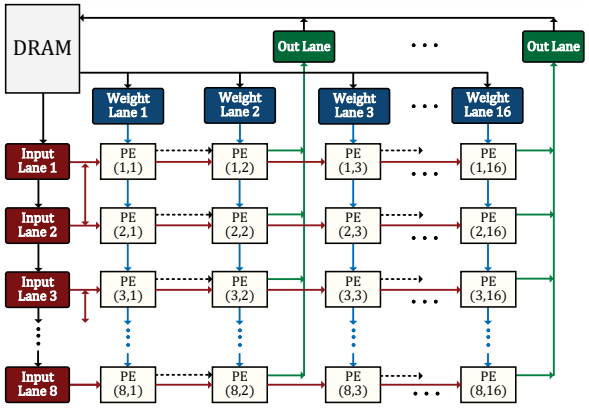

Fig. 7. Overview of TermiNETor's baseline architecture.

channels of an image. Similarly, each weight lane broadcasts a unique filter (occasionally, up to three different filters) at a time to all the PEs of the same column. Therefore, a PE row produces the same output indexes of different channels. The operations are bit-serial on the weights, so the weight lanes only transfer one bit of each weight. In our setting, a brick consists of eight elements. Therefore, each activation lane supplies 8×8-bits inputs, and each weight lane provides eight single bits (occasionally 16 or 24) of different weights (i.e., one bit of each). The activations and weights correspond to the same indexes; hence, the PEs can perform immediate MAC operations, which in practice is just a shift-and-add operation. The operations are orchestrated in a fashion that all PEs of a row terminate at the same time and move to the next sub-image, independent of the other rows, and only infrequent halts are needed to coordinate different PE rows. TermiNETor benefits from multiple levels of data reuse and sharing to reduce both on-chip and off-chip memory accesses, which we elaborate in subsection III-C.

B. Dataflow

TermiNETor partitions an $X \times Y \times \mathcal{C}$ input feature map into multiple $x \times y \times \mathcal{C}$ sub-images, as shown in Fig. 8. Each sub-image is stored in a separate input lane and is shared with all PEs of a row. The values of x and y depend on the PE's local register file (RF) size as well as the number of filters a PE receives. For instance, with 128 filters, each PE of the 8×16 array will visit $\frac{128}{16} = 8$ filters. Thus, an RF with a depth of 32 words can allocate four words per filter (i.e., 2×2 output pixels for eight channels); hence, the sub-images can be up to $4 \times 4 \times \mathcal{C}$ assuming filters of shape $3 \times 3 \times \mathcal{C}$ with sliding stride of 1. The number of sub-images can be more than the input lanes (PE rows). In that case, the remaining sub-images will be processed in multiple similar iterations after the currently stored sub-images are processed for all the filters. TermiNETor processes the stored sub-images as follows.

First, brick 1 of filters 1 to 16 are loaded into PE columns 1 to 16, respectively. As alluded above, a brick consists of eight consecutive elements in the Z axis (channels). Operating at the brick level avoids local data storage and simplifies the compute

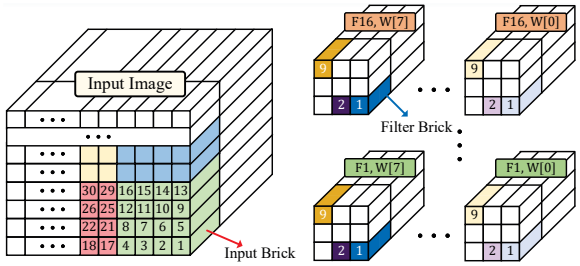

Fig. 8. A sample input and filters for TermiNETor dataflow explanation.

elements. Initially, all PEs of a column share the same filter brick. The filters' bricks contain eight weights, but only one bit of each. Hence, only $W[7]$ of all weights are loaded at this stage. We pack the same index of the weights in the same memory word for regular memory accesses. The sub-images are scanned from the bottom row toward the top. Overlapping bricks of adjacent sub-images will be transferred from one input lane to the other. E.g., in Fig. 8, input lane 1 stores the 4×4 green sub-image (indexes 1 to 16); to produce output pixel index 3, it needs to read some of the inputs from the red sub-image. After applying brick 1 to the *corresponding* bricks of the input sub-image (i.e., to the inputs that need the brick 1 to generate the 2×2 output in the above example), bricks 2 to 9 of the current filters are applied to their corresponding input bricks, as well. Thereafter, we move to the next bricks (deeper channels) of the sub-image as well as of the loaded filters. Finally, after processing all the channels, we load the next batch of the filters (17–32) and repeat the same procedure.

So far, one bit of all the filters ($W[7]$) is used. Before repeating the whole procedure with $W[6]$, we examine if certain PEs can be released. We perform the PE release in the granularity of a row. All PEs of a row generate the same (i, j) positions but for different channels. Since the outputs among the same positions of different channels are more correlated, we enforce this row-termination during the TermiNETor calibration. That is, all outputs produced with the PEs of a row try terminating at the same weight index, which we call row-termination or group-termination. In particular, the outputs window generated by a row is small (e.g., 2×2 when there are 128 filters, or 2×1 for 256 filters), which keeps the value correlated.

Once a row is terminated, its input lane loads another sub-image and starts over with $W[7]$ of filters 1–16 for the new sub-image. Nevertheless, the weight lanes are shared among all the rows. Therefore, the other rows might need to continue processing with $W[x \neq 7]$ and cannot supply $W[7]$. However, all the weight lanes supply the same filter at a given time. That is, the terminated row needs to load filters 1–16 with $W[7]$, and the rest of the rows need to also load filters 1–16, albeit weight $W[x \neq 7]$. Also, all the rows and PEs use the same brick index, which facilitates address generation. Therefore, at the cost of increasing the weight lane size, we use multiple banks so a weight lane can provide different weight bits when certain rows advance the other ones. It is straightforward as the brick index, and hence, the memory address will be the

same for all rows. Note that the weight lanes are small as they store a single bit of a filter at a time. We limit the number of different indexes to three. On the rare occasions that more than three different weight bits are needed, we halt the outpaced rows to coordinate the rows.

The number of cycles to process an $X \times Y \times \mathcal{C}$ input image by the 8×16 TermiNETor array can be calculated as follows:

$$\text{cycle count} = \frac{X \times Y}{8} \times k_1 \times k_2 \times \frac{\mathcal{F}}{16} \times \frac{\mathcal{C}}{8} \times W_{\text{avg}} \quad (3)$$

where $k_1 \times k_2$ is the filters' kernel size (sliding stride of 1), $\mathcal{F}$ is the number of filters, and W_{avg} is the average number of weight bits consumed among the outputs. The 8 in $\frac{X \times Y}{8}$ indicates the number of rows that can independently process a sub-image, 16 in $\frac{\mathcal{F}}{16}$ corresponds to the number of columns which enable loading the filters as batches of 16, and 8 in $\frac{\mathcal{C}}{8}$ denotes the brick size, since we process eight channels at once.

C. Data Reuse

TermiNETor takes advantage of different types of data sharing and reusing provided by its architecture and/or dataflow. Since a weight lane is shared among all the PEs within the corresponding column, a weight fetched from these memories will be used with multiple PEs of a column. The same also holds for the input activation lanes, i.e., the activations of a lane are broadcast to all 16 PEs of a row. In addition, a loaded weight brick stays in the PE until all the input activations that it can use are processed (e.g., in Fig. 8, after loading brick 1, all the sub-image inputs it can use are fetched in a row-major manner before loading brick 2). Furthermore, the sub-images that are stored in the input lanes are used for all filters before evicting, and there is no need to re-fetch from the DRAM. These activations remain in the input lanes until all the filters are processed. A difference between input bricks and weight brick is that a weight brick stays in the PE's local register, while new activations are constantly read from the local input lanes. Note that the input lanes consume significantly smaller power than DRAM.

Another type of filter reuse that TermiNETor supports is via batch processing. In the last convolution layers the image size becomes smaller, e.g., $4 \times 4 \times \mathcal{C}$. In such a case most of the PE rows become unused, but are still supplied with the filters that are shared across the columns. Hence, they can operate on different images using their independent input lanes. In batch processing mode, TermiNETor processes each image layer by layer before a layer ℓ causes PE underutilization. TermiNETor saves the inputs of the layer ℓ in the DRAM, processes other image(s) in the same fashion with full resource usage until layer ℓ, and continues to process the layer ℓ and subsequent ones for the stored feature maps by loading multiple of them at a time. This helps to maximize PE utilization with a small increase of the embedded DRAM size as only one (and small) layer per image needs to be temporarily stored.

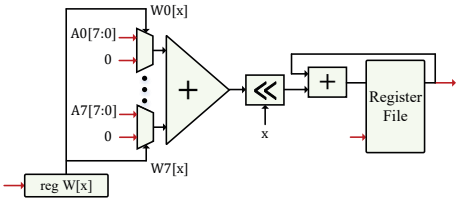

Fig. 9. A processing element (PE) of TermiNETor.

D. Processing Elements

TermiNETor benefits from uncomplicated processing elements, which is shown in Fig. 9. As explained in subsection III-B, a brick of eight different weights, one bit of each, is loaded into the PE and is applied on all required bricks of the row sub-image. Therefore, an eight-bit register stores the 8×1 bit weights. At each cycle, eight 8-bits activations are read from the input lane and broadcast into the PEs. We only latch these activations at the output of the lane SRAM using 8×8 flip-flops which drive the whole row; hence, no input register is needed in the PE. The 32-words RF accumulates the partial sums associated with the PE (e.g., for eight filters, four outputs per each). The address of the RF row, input from the controller, depends on which of the assigned filters and outputs are being processed. The $0 \leq \text{x} \leq 7$ denotes the weight bit index that is being processed and determines the amount of shifts.

Upon termination signal after a certain weight bit index, the content of the RF is transferred to the output lanes. For implementation purposes (to make the RFs compact), we share a $4\times$ wide RF between four PEs. Therefore, as shown in Fig. 7, an output lane is shared between several columns, and the links between PEs in a row show writing the computation result in the shared RF (note that RFs use the same address).

IV. Experimental Results

A. General Setup

We implemented the algorithmic flow of TermiNETor, i.e., bit-level mask and threshold generation, early convolution termination, and calibration (including enforcing row-termination) using PyTorch. For training, we used SGD optimizer, momentum of 0.9 with weight decaying, and learning rate from 0.1 down to 0.0002 over 100 epochs. To recap, the weights and activations are quantized to eight bits and the convolution operation is implemented by bitwise shift-and-add operations. Using the pretrained model, we derive the layer-wise bit index T at which the dynamic mask (for ineffectual output predictions) generation starts. Our experiments showed that the mask generation can start after consuming the first three bits (towards MSB) for most of the layers. For certain layers (e.g., layer 4 of VGG16 on CIFAR-100), we can start the mask generation after consuming the first two bits of weights.

We implemented the TermiNETor accelerator in SystemVerilog and verified its functionality using Modelsim. We used Synopsys Design Compiler O-2018.06 to synthesize the RTL code, and Cadence Innovus 2019 for placement and

TABLE I
ACCURACY AND OPERATION COMPARISON OF TermiNETor WITH BASELINES.

	Network	Accuracy				Operation (M)			
		Base	SeerNet	4-bits	TermiNETor	Base	SeerNet	4-bits	TermiNETor
CIFAR 10	VGG16	91.65%	91.30%	90.82%	91.52%	314	262 (1.20×)	157 (2×)	187 (1.68×)
	ResNet18	93.81%	93.38%	93.02%	93.56%	555	499 (1.11×)	277 (2×)	384 (1.45×)
	ResNet50	94.58%	93.20%	93.24%	94.08%	1298	1103 (1.18×)	649 (2×)	885 (1.47×)
CIFAR 100	VGG16	70.40%	70.12%	69.01%	70.07%	318	267 (1.19×)	159 (2×)	191 (1.66×)
	ResNet18	75.41%	74.96%	73.15%	75.13%	555	492 (1.13×)	277 (2×)	381 (1.46×)
	ResNet50	77.30%	76.84%	75.12%	77.01%	1298	1056 (1.23×)	649 (2×)	893 (1.45×)
Image-Net	VGG16	56.91%	56.04%	55.15%	56.84%	1256	1017 (1.24×)	628 (2×)	833 (1.51×)
	ResNet18	61.84%	61.01%	59.92%	61.66%	2222	1990 (1.12×)	1111 (2×)	1568 (1.42×)
	ResNet50	64.13%	63.32%	61.90%	64.02%	5194	4672 (1.11×)	2597 (2×)	3685 (1.41×)

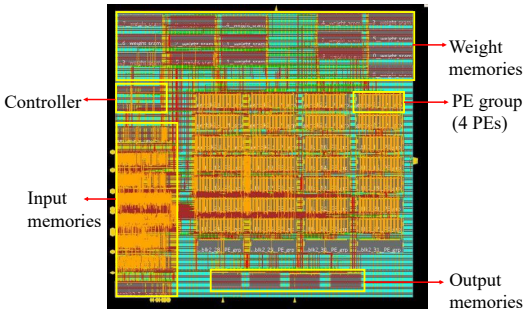

Fig. 10. Post-placement and routing layout of TermiNETor accelerator, implemented in TSMC 40 nm technology.

routing using TSMC 40 nm 0.9V library (using a combination of HVT and RVT) for the typical (TT) process corner. We used TSMC 0.8V high-performance memory compiler to generate the dual-port register file (for PEs) and single-port SRAMs (for input activation lanes, weight lanes, and output lanes). Thanks to the simplicity of the processing elements, TermiNETor could achieve an operating frequency of 1 GHz. We used the post placement and routing netlist to report the area and power using Design Compiler. To calculate the DRAM access energy, we used Destiny [23] to generate a 32 MB (128 bit words) eDRAM model for 40 nm and added the DRAM power based on its aggregate read and write throughput. The 32 MB DRAM can store the whole input/output feature maps of any layer, as well as the model weights.

Fig. 10 illustrates the post placement and routing layout of the baseline TermiNETor (8×16 array), which occupies $0.95\,\text{mm}^2$. Each of the eight rows are divided into groups of four PEs, that share a wide register file. The input activations memories (left), weight memories (top), and output memories (bottom) are also visible.

B. Operation Reduction

We evaluate the algorithmic effectiveness of TermiNETor by comparing it with SeerNet [16], which is a two-stage ineffectual output predictor. The first stage of SeerNet uses a 4-bits quanitized inference for mask generation, and then it uses the generated mask for full-precision (8-bit) inference as the second stage. The performance improvement, defined as effective operations, is reported with respect to the 8-bits baseline implementation (multiplication of a w-bits weight is considered as $\frac{w}{8}$ of a 8-bits weight multiplication). We also include the performance and accuracy metrics for the 4-bit weight quantization (which uses $\frac{1}{2}$ effective operations compared to 8-bits weights). In our evaluation, we considered VGG16, ResNet-18, and ResNet-50 networks using CIFAR-10 and CIFAR-100 datasets, as well as a 200-class subset of ImageNet (Tiny ImageNet).

Table I summarizes the accuracy and effective operation count for the aforementioned models and datasets. The *Base*, *SeerNet* and *4-bit* columns represent the baseline 8-bit quantized model, SeerNet inference [16] and 4-bit quantized models. SeerNet, which is only software-centric optimization, does not reuse the computations performed in the mask generation stage (4-bits inference), leading to computation overhead. This overhead in computation nullifies the gains of skipping ineffectual outputs when the output sparsity is less than 50%. It is worth noting that the results presented in SeerNet [16] do not account for the computations performed in the first phase (mask generation). In the following we summarize the results of different datasets.

CIFAR-10: As compared to the baseline VGG16 network, while SeerNet achieves 1.20× operation reduction with 0.35% accuracy loss, our method offers 1.68× reduction with only 0.13% accuracy loss. For residual networks such as ResNet-18, while SeerNet's speedup is 1.11×, our method offers 1.45× improvement with better accuracy. We observe a similar trend for ResNet50; 1.47× speedup in TermiNETor versus 1.18× of SeerNet, and 0.88% higher accuracy. Since down-sampling in residual networks is manly implemented using a stride-2 convolution, the sparsity of output activation is low, leading to low speedup values in the residual networks as compared to VGG16. Although the 4-bit model offers 2× speedup, it suffers accuracy loss of up to 1.34%.

CIFAR-100: For CIFAR-100, we achieve 1.66× speedup in operation count using VGG16, 1.46× using ResNet-18, and 1.45× on ResNet50, which are on average 34% higher than SeerNet, with 0.1% higher accuracy. The 4-bit quantization offers 2× speedup, but incurs 1.94% accuracy loss (versus 0.3% of our approach).

TinyImageNet: We observe a similar trend with the Tiny ImageNet dataset. Along with an improved operation reduction (up to 1.51× speedup) compared to the baseline, our improvements are 25.2% better than SeerNet with 0.72% higher accuracy (only 0.12% drop compared to the 8-bits baseline).

The 4-bits quantized models offer a constant 2× operation

TABLE II
OPERATION REDUCTION WITH ∼2% ACCURACY BUDGET.

	Network	Accuracy		Operation (M)	
		Base	TermiNETor	Base	TermiNETor
CIFAR 10	VGG16	91.65%	89.90%	314	96 (3.27×)
	ResNet18	93.81%	91.61%	555	214 (2.59×)
	ResNet50	94.58%	92.33%	1298	496 (2.62×)
CIFAR 100	VGG16	70.40%	68.16%	318	98 (3.24×)
	ResNet18	75.41%	73.11%	555	219 (2.53×)
	ResNet50	77.30%	75.16%	1298	503 (2.58×)
Image-Net	VGG16	56.91%	54.94%	1256	502 (2.50×)
	ResNet18	61.84%	59.86%	2222	958 (2.32×)
	ResNet50	64.13%	62.12%	5194	2144 (2.42×)

reduction. However, 4-bits models suffer an accuracy loss of up to 2.2%. Accordingly, we allowedTermiNETor to undergo an accuracy degradation of up to ∼2.2% (similar to 4-bits models) by starting the mask generation early in the bit-serial processing of the weights as well as using more aggressive threshold values. With up to 2% accuracy loss as compared to the 8-bits baselines, we observe an operation reduction of upto 3.3× for VGG16 with CIFAR-10 dataset as compared to 2× speedup of the 4-bit model. As shown in Table II, TermiNETor achieves better speedup values for other datasets and networks, compared to the 4-bit models.

C. TermiNETor *Accelerator Evaluation*

Table III presents the characterization of TermiNETor hardware components. Each PE has a 32×20 bit register file, but as as mentioned in Section III-D, we merge the RFs of four PEs in a row and share a 32×80 bit RF among all. Considering 8-bits weights and activations, 20-bits words are sufficient to accumulate all partials (as most of the products cancel out each other). The entire array needs $32 \times 20\,\text{b} \times 8 \times 16 = 10\,\text{KB}$ of register file. To avoid DRAM stalls when flushing out the output activations from RFs, we use intermediate output lanes consisted of four SRAMs, each of which has a size of 1 KB (256×32-bits). We stall new computations in the architecture during the data transfer from the RFs to the output lanes, which takes $32 \times 8 = 256$ (an output lane is shared between all eight rows). In networks such as ResNet-18 and ResNet-50, this stall is utilized to perform residual layer addition.

The 8×16 architecture of TermiNETor occupies a total area of 0.95 mm^2 (at 40 nm), consumes 4.0 mW leakage power, an average dynamic power of 222.5 mW (at 1 GHz), 9.2% of which is the DRAM access power (average among benchmarks). The baseline 8×16 array of TermiNETor is able to process 12.2 ImageNet-size (224×224 images) implementing VGG16 (large image and network), or 235 smaller 32×32 images running ResNet-18 (small image and network). Fig. 11 shows how the throughput scales by

TABLE III
CHARACTERIZATION OF TermiNETor COMPONENTS.

Module (total)	Size/count	Area (μm^2)	Power (mW)
Processing Element	8×16	81,263	42.8
Activation Memory	16 KB	281,636	59.1
Weight Memory	18 KB	271,596	15.2
Output Memory	4 KB	75,163	1.9
Register File	10 KB	233,987	78.2
Controller	1	9,258	4.78

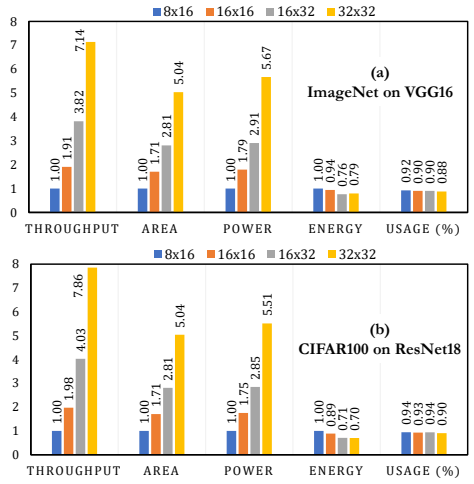

Fig. 11. Scalability of performance, area, power, and utilization of TermiNETor running different image sizes.

increasing the TermiNETor accelerator size up to 32×32 array. The baseline is shown by 1.0× (which denotes 12.2 image/second for VGG16, and 235 image/second for ResNet-18 as mentioned). The largest array, i.e., 32×32, which consists of 8× more PEs, can increase the performance by up to 7.86×, which is almost linear with the number of PEs added.

The same Fig. 11 also shows the area sociability, which is independent of image size. While the baseline area is 0.95 mm^2 (denoted by 1.0×), the area of the largest array (32×32) is only 5.04× larger (i.e., $5.04 \times 0.95\,\text{mm}^2 = 4.79\,\text{mm}^2$). The area increase is $< 8\times$ because although the PE count is increased by 8×, the number of input lanes of the 32×32 array is only 4× of the baseline array, and the number of weight and output lanes increases by just 2×. A similar trend can be observed for the power consumption. As alluded above, the leakage and dynamic power of the baseline array is 4.0 mW and 222.5 mW, respectively. The energy consumption reduces by increasing the array size due to better data reuse, i.e., the same weights and inputs are shared among more PEs, and a result, SRAM accesses are reduced. PE utilization is lower for ResNet-18 than VGG16 due to the residual layers, which have a 1×1 kernel that enables the output lane to process more filters at the expense of activations. For residual layers with larger number of input channels (≥ 512), there are a limited number of activations to start with, and early termination combined with the low capacity of the output buffer diminishes it even further. It results in a PE rows remaining unused for a significant time.

D. *Comparison with Previous Work*

Since the performance and power consumption depends on the resources (e.g., PE array size), we compare TermiNETor with state-of-the-art works in terms of energy consumption per classification. Particularly, the results of subsection IV-C showed that the performance of TermiNETor scales well with the PE array size; hence, high performance targets can be

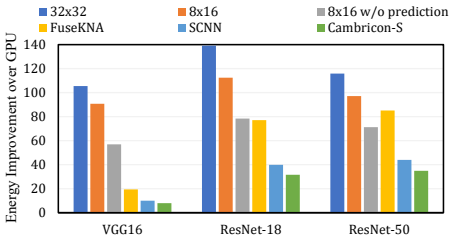

Fig. 12. Energy per classification, normalized to GPU.

meet if the energy consumption is satisfactory. Therefore, we compare the energy consumption of TermiNETor (for both baseline 8×16 and larger 32×32 array, as well as the baseline baseline 8×16 *without* early termination) with SCNN [9], Cambricon-S [11], and FuseKNA [24]. The former two (SCNN and Cambricon-S) are zero weight/input skipping accelerators without bit-serial operations (i.e., sparsity-aware 8-bits accelerators), while FuseKNA skips zero inputs as well as repetitive computations using bit-serial operations. The results of SCNN, Cambricon-S, and FuseKNA are presented in [24] as normalized to Nvidia 1080 GTX GPU; hence, we repeated the experiments using the same GPU and normalized the TermiNETor results to the obtained result of GPU.

Fig. 12 compares the energy usage of TermiNETor and the other accelerators running 224×224 images on VGG16, ResNet-18, and ResNet-50, normalized to Nvidia 1080 GTX Ti GPU. The 32×32 array of TermiNETor achieves the highest energy efficiency and reduces the energy consumption by $120.1\times$ over GPU, $1.98\times$ over FuseKNA, $3.84\times$ over SCNN, and $4.84\times$ over Cambricon-S. Compared to the baseline 8×16 array with (without) early termination, the 32×32 array is $1.20\times$ ($1.75\times$) more energy efficient (with $\sim$5.7$\times$ higher power consumption according to Fig. 11, i.e., $\sim$1290 mW versus 226 mW of the baseline TermiNETor array).

V. Conclusion

In this work, we introduced TermiNETor that accelerates CNN inference by performing dynamic convolution termination for ineffectual output activations. During the weight bit-serial inference data flow, at every weight bit, we predict ineffectual output activations resulting from the ReLU and max-pooling layers. Using these predictions, the donwstream bitwise shift-and-add convolution operations are continued only for the useful output activations. We evaluated TermiNETor across various networks and datasets, demonstrating a significant reduction in operation count (up to $1.7\times$ speedup) with negligible loss of accuracy. We also proposed a novel accelerator that exploits the dynamic bitwise convolution terminations with an average energy efficiency of $120\times$ as compared to GPU, and at least $1.98-4.84\times$ with respect to the state-of-the-art sparsity-aware accelerators.

Acknowledgements

This work was supported by TSMC, in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #1911095, #1826967, #2100237, and #2112167.

References

[1] A. Ren *et al.*, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *ACM ASPLOS*, 2019, pp. 925–938.
[2] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[3] M. Lin *et al.*, "Hrank: Filter pruning using high-rank feature map," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1529–1538.
[4] M. Rastegari *et al.*, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.
[5] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.
[6] A. Zhou *et al.*, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017.
[7] S.-E. Chang *et al.*, "Mix and match: A novel fpga-centric deep neural network quantization framework," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 208–220.
[8] Z. Song *et al.*, "Drq: dynamic region-based quantization for deep neural network acceleration," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1010–1021.
[9] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.
[10] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.
[11] S. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.
[12] M. Mahmoud *et al.*, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 781–795.
[13] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 382–394.
[14] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "Predictivenet: An energy-efficient convolutional neural network via zero prediction," in *IEEE international symposium on circuits and systems (ISCAS)*, 2017, pp. 1–4.
[15] M. Song *et al.*, "Prediction based execution on deep neural networks," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 752–763.
[16] S. Cao *et al.*, "Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 216–11 225.
[17] D. Pinto, J.-M. Arnau, and A. González, "Mixture-of-rookies: Saving dnn computations by predicting relu outputs," *arXiv preprint arXiv:2202.04990*, 2022.
[18] V. Akhlaghi *et al.*, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 662–673.
[19] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 764–775.
[20] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.
[21] D. Yu, H. Wang, P. Chen, and Z. Wei, "Mixed pooling for convolutional neural networks," in *International conference on rough sets and knowledge technology*. Springer, 2014, pp. 364–375.
[22] P. Judd *et al.*, "Stripes: Bit-serial deep neural network computing," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.
[23] M. Poremba *et al.*, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, pp. 1543–1546.
[24] J. Yang *et al.*, "Fusekna: Fused kernel convolution based accelerator for deep neural networks," in *International Symposium on High-Performance Computer Architecture*, 2021, pp. 894–907.