AirNN: A Featherweight Framework for Dynamic Input-Dependent Approximation of CNNs

Maedeh Hemmat, Joshua San Miguel and Azadeh Davoodi
Department of Electrical and Computer Engineering
University of Wisconsin - Madison, Madison, WI, USA
Email: hemmat2@wisc.edu, jsanmiguel@wisc.edu, adavoodi@wisc.edu

Abstract. In this work, we propose AirNN, a novel framework which enables dynamic approximation of an already-trained Convolutional Neural Network (CNN) in hardware during inference. AirNN enables input-dependent approximation of the CNN to achieve energy saving without much degradation in its classification accuracy at run-time. For each input, AirNN uses only a fraction of the CNN's weights based on that input (with the rest remaining 0) to conduct the inference. Consequently, energy saving is possible due to fewer number of fetches from off-chip memory as well as fewer multiplications for majority of the inputs. To achieve per-input approximation, we propose a clustering algorithm which groups similar weights in the CNN based on their importance, and design an iterative framework which decides dynamically how many clusters of weights should be fetched from off-chip memory for each individual input. We also propose new hardware structures to implement our framework on top of a recently-proposed FPGA-based CNN accelerator. In our experiments with popular CNNs, we, on average, show 49% energy saving with less than 3% degradation in classification accuracy due to doing inference with only a fraction of the weights for the majority of the inputs. We also propose a greedy interleaving scheme, implemented in hardware, in order to improve the performance of the iterative procedure and compensate for its latency overhead.

Keywords: Neural Network Accelerators, Early Termination, Input-dependent Approximation, Approximate Computation, Low-energy Design.

I. Introduction

In recent years, deep neural networks, inspired by human brain system, have been impressively successful in performing complicated tasks including object recognition, face detection, and image classification. Due to high accuracy of these networks, they have gained significant attention in different fields of machine learning and computer vision. However, the accuracy boost comes with significant increase in the size of neural networks which mandates their efficient hardware implementation. An important challenge for the efficient implementation is their high demand in terms of memory and computation; state-of-the-art Deep Neural Networks (DNNs) require storage of several Megabytes of parameters and execution of billions of operations for each inference [7], [19].

To reduce computational and memory costs of DNNs, various approaches have been proposed to simplify the network and boost the energy efficiency. Pruning, quantization, low-rank approximation, and approximate computation are among widely used techniques that aim to increase the efficiency of the network by means of reducing the number of parameters to store in memory or decrease the precision of computational operations [3], [6], [18], [24], [13]. As an example, in [24], the authors propose to approximate the network by removing less critical nodes to reduce computation complexity. However, all of the aforementioned techniques are mainly applicable during training and do not provide an opportunity to trade off accuracy and energy dynamically. More importantly, these techniques are oblivious to the variations in the inputs.

Motivation: Need for Input-independent CNN Approximation. A key insight is that not all inputs require the same amount of computation to be correctly classified. While some inputs need to perform all computations in the network, the vast majority are easy to classify. Furthermore, not all weights in the network contribute equally to generate the output. Given these observations, reconfigurable and approximate network designs have been proposed in [17], [23], [21].

The authors in [21], [20] propose SeFAct, a framework for dynamically reducing the energy consumption of the network and enabling network early termination. To achieve this, they propose to adaptively activate a subset of all neurons to do the inference, to be determined by the received input, thus reducing energy. This approach though requires a special learning phase in order to find the appropriate correlation between activations and inputs belonging to different classes.

In [17], a teacher-student scheme is exploited to propose Big/Little DNNs for energy-efficient inference. Here, a Little network with fewer layers/less energy and a Big network with higher complexity and accuracy are trained. Then, during inference, the Little network is executed first and Big network is inferred only if Little can not provide an acceptable result. However, this method can even increase the memory and computational costs if the two networks do not have shared weights or layers. In addition, the user cannot reconfigure the design with respect to the available energy budget.

In [23], the authors propose an incremental learning/inference framework in which the network is trained incrementally at the beginning by changing its architecture through increasing the number of filters in convolutional layers at each iteration. Then, in the inference mode, a portion of the network can be turned off whenever the network can provide acceptable accuracy. The main drawback of the proposed framework though is that it needs an incremental training process and can not be directly applied to an already trained network. In addition, although fully connected layers are known to be redundant and a significant portion of their weights do not play an important role in accuracy, they are not optimized for incremental inference to boost energy efficiency.

Our Solution: AirNN. To solve the aforementioned challenges, we propose AirNN, a featherweight framework for dynamic and input-dependent approximation of CNNs. Our goal is to approximate an already-trained CNN in hardware dynamically, based on the received input in order to trade off accuracy and energy and enable network early termination. *The unique feature of AirNN is that it approximates the CNN for each input.* The approximation is achieved by fetching only a fraction of the total weights from off-chip memory, to be determined based on the specific input, with the rest of the weights remaining as 0. This directly translates into energy saving due to fewer fetches from the energy hungry off-chip memory and fewer computations. Orthogonal to input-dependent approximation, the proposed framework allows the user to control the accuracy and energy consumption in run-time via adjusting the termination condition; As a result, more energy saving can be achieved at the cost of negligible accuracy degradation.

To enable network approximation while maintaining the performance, we propose two algorithms, Clustering and Interleaving, to be implemented in hardware:

1) *Clustering algorithm is proposed to improve network energy-efficiency.* The proposed clustering algorithm divides weights of the pre-trained CNN in terms of their importance into groups. For each input, we propose an iterative process (implemented in hardware) which fetches only the required number of important clusters from memory. We also propose new hardware structures to implement our framework on top of a recently-proposed FPGA-based CNN accelerator to dynamically approximate the CNN at run-time. This is while hardware implementation of the core CNN remains intact. The proposed framework can decrease the network energy consumption for around 15%-90% across different networks with less than 3% degradation in accuracy.
2) *Interleaving algorithm is proposed to improve network performance (execution time).* AirNN may increase the latency and execution time of the network for inputs that need several round of inference to generate acceptable output. To tackle this, we propose an interleaving algorithm, implemented in hardware. The proposed interleaving algorithm exploits the ”extreme” sparsity of the weight matrixes originated from clustering to improve the performance. The proposed interleaving algorithm can improve network execution time by up to 60%.

The rest of the paper is organized as follows. In Section II, we briefly overview related works. Section III describes the proposed framework, AirNN, followed by the details of our clustering algorithm. The hardware design is provided in Section V. Our proposed Interleaving scheme followed by its hardware implementation is discussed in Section VI. Lastly, AirNN performance is evaluated in Section VII.

II. Related Works

Prior works on reducing memory and computational cost of deep neural networks can be divided into two main categories: **input-independent** and **input-dependent** approaches.

Input-independent approaches compress the network model statistically and during training, regardless of the variations in the received inputs. These techniques usually focus on reducing the number of parameters and/or reducing the precision of each individual weight. Quantization, low-rank approximation, and pruning are among widely used techniques in this category. These techniques though are mainly applicable during training and do not compress the network model based on the received input.

It should be noted that although AirNN aims to reduce the number of fetched weights during inference, it is different from pruning for two important reasons. First, in pruning, the number of parameters remains unchanged during inference. Whereas in AirNN, the number of parameters is dynamically changed during inference with respect to the received input. Second, in pruning, there is no opportunity to trade off energy consumption and accuracy by means of using less/more weights once the network is trained. In contrast, AirNN is able to control the number of clusters (which determines the number of weights) and termination condition to adjust energy savings.

Input-dependent techniques, in contrast, take the variations in the received input into consideration, thus dynamically pruning/approximating the network at run-time. The key motivation for the input-dependent techniques are twofold: not all inputs require the same amount of computation to be correctly classified, and not all the weights in the network contribute equally to generate the output. Input-dependent approximation techniques can be divided into three main groups: 1) skipping ineffectual MAC operations based on the received input, 2) skipping non-critical layers of the network, and 3) dynamic weight pruning. Our work falls into the third category.

Skipping ineffectual MAC operations. Some of the computations in the network will eventually become ineffectual due to presence of ReLU function (which clamps the negative neurons to zero) and Pooling function (which downsamples the neurons). With this observation, the authors in [1], [22] propose approaches that predict whether or not a neuron becomes ineffectual as it propagates through the network. If eventually ineffectual, the network will skip the MAC computations for that neuron. Due to the variations in the received inputs, the ineffectual neurons will vary from one input to another. Hence, ineffectual neuron prediction should be done at run-time.

Skipping non-critical layers. Prior works in [16], [15] exploit the concept of ”easy vs. difficult inputs” to approximate the network. Given that some inputs are easier to classify, these works propose to dynamically skip some of the layers if the network reaches an acceptable output. To achieve

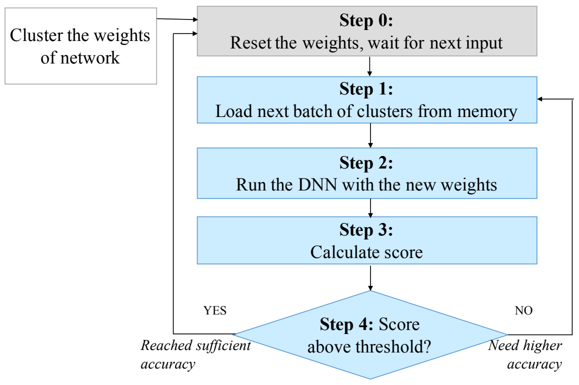
Cluster the weights of network
Step 0: Reset the weights, wait for next input
Step 1: Load next batch of clusters from memory
Step 2: Run the DNN with the new weights
Step 3: Calculate score
Step 4: Score above threshold?
YES
Reached sufficient accuracy
NO
Need higher accuracy

Figure 1: Overview of our approach.

this, for example, the authors in [16] proposed to insert a simple classifier right after each convolutional layer. Then, depending on the confidence of each newly-inserted classifier, the network will decide upon executing or skipping the next convolutional layer. This approach though will increase the network model size and number of parameters because of the inserted classifiers.

Dynamic weight pruning. In contrast to static weight pruning, the works in this group aim to prune the weights dynamically and at run-time, to be determined based on the current input. SeFAct, proposed in [21], [20] and our proposed framework, AirNN, fall into this category. In SeFAct, the authors propose to adaptively activate a subset of all neurons to do the inference, to be determined by the received input, thus pruning non-critical weights. This approach though requires a special learning phase to extract the correlation between the weights and the inputs belonging to different classes. In [20], the authors discuss how optimized thresholds for learning phase can be found. The results show 20%-25% energy saving in LeNet5 and AlexNet.

AirNN is fundamentally different from the prior input-dependent approaches in three major ways. 1) Our work guarantees to maintain accuracy while approximating the network and reducing the energy. It is because AirNN breaks down the inference into several iterations and allows each input to go through as many iterations as required to be correctly classified. 2) AirNN is not increasing the network model size as it is not adding new layers/parameters to the network, thus it can be implemented with the minimal overhead. 3) AirNN introduces the termination condition as a knob that enables users to explore the tradeoff between the classification accuracy and energy consumption.

III. Overview of Our Approach

We are given an already-trained CNN so the weights are known a-priori. Our goal is to approximate the CNN, dynamically and for each input, thus enabling early termination. This input-dependent approximation is performed via using only a fraction of the original (non-zero) weights and without altering the structure and hardware implementation of the core CNN. Using a fraction of the weights translates into energy saving by means of reducing off-chip data accesses from DRAM and elimination of some multiplications. Input-dependent approximation results in degradation in accuracy. Our goal is to achieve the maximum energy saving subject to not falling below a threshold of an accuracy score [12].

AirNN procedure. Figures 1 shows a high-level flow chart of our input-dependent approximation, AirNN. First, as a pre-processing step, we run a clustering algorithm which groups the weights in the CNN into K clusters. The clustering algorithm works at the granularity of a layer. The ordering does not matter because the algorithm is independently applied to each layer. For each layer, the weights are divided into K clusters, by solely looking at the values of the weights in that layer.

Clustering is done only once and offline, and the clusters are stored as separate groups in memory. The clustering algorithm ensures that the weights within one cluster have the most similarity to each other, while the weights in different clusters have the least similarity. This allows viewing the K groups of weights in terms of their "importance" in the sense that the cluster with the highest absolute value of weights contributes the most to the classification accuracy.

As shown in Figure 1, after the one-time clustering, Step 0 begins upon receiving a new input for classification. In this step, all weights in the CNN are reset to 0. Next an iterative procedure starts to approximate the CNN and conduct inference for that input. At Step 1, we load a new batch of weights from memory which are fetched from the clusters with the highest level of "importance". AirNN fetches a new group of weights for each layer from the clusters of that layer. As a result, each layer is approximated more accurately compared to the previous round because of utilizing an additional cluster of fetched weights during inference. The remaining weights in the network remain 0. Then the CNN runs with these weights in Step 2 which allows calculating a score in Step 3. If the score is below the given threshold, a new iteration starts (for that same input) to fetch the next batch of "important" weights. The new weights in the network will be the weights from the iterations so far, for example the first two batches in this case, while the remaining weights remain 0. The iterative algorithm terminates as soon as the accuracy score reaches above the specified threshold indicating that an acceptable classification accuracy has been reached. Upon termination, the procedure goes back to Step 0 and waits to receive the next input. Per-input approximation of the network and its early termination are main sources of energy consumption reduction during inference.

The above procedure approximates the CNN per input with as few non-zero (original) weights as possible. Each iteration of our procedure can be viewed as applying a level of approximation for the considered input, with the first iteration having the most approximation and the final one having the least. A separate inference is done per iteration so the procedure may be viewed as performing incremental inference with more number of (non-zero) weights per iterations.

Each batch of weights that are fetched per iteration consists of two clusters which are the ones with the maximum positive and minimum negative average weights across the clusters. The reason behind this is that some functions in the CNN (particularly ReLU) are sensitive to the sign of the input to generate the accurate output. Therefore, the clusters with the

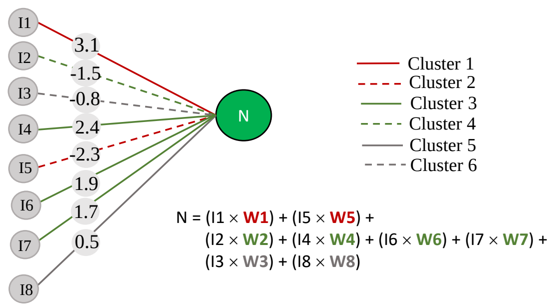

Figure 2: Illustration of the approximate computing performed at a neuron by sequentially fetching clustered weights.

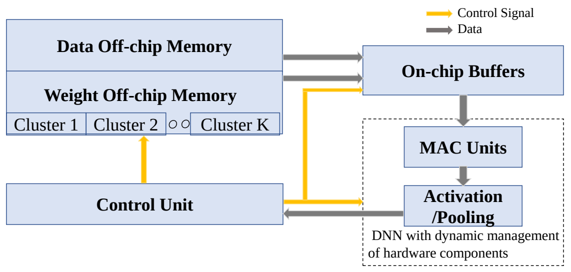

Figure 3: Overview of the underlying hardware.

most positive and most negative average weights should both be given higher priority.

For each input, up to $\lceil \frac{k}{2} \rceil$ iterations may occur to reach a final approximation and produce the final inference result. In the extreme case, all clusters will be fetched so no weights will be zero and there won't be any approximation (which guarantees exceeding the minimum threshold eventually). We show in our experiments with well-known CNNs and datasets that often times, one or two iterations are sufficient to approximate with high accuracy for the majority of inputs.

The termination condition of our procedure controls the accuracy. It is based on calculating a score in Step 3. We use the metric proposed in [17] to calculate the score which is the absolute value of difference between the two largest output neurons of the last layer in the CNN. Due to presence of fully connected layer as a classifier at the last stage of all CNNs, each output neuron represents the probability that the input belongs to the corresponding class (i.e., a number in range (0,1)). Consequently, small values of score translates to higher likelihood of misclassification since it reflects that two output neurons have close probabilities. Therefore, larger score is equivalent to more accurate classification.

In our experiments, we first set the threshold to 0.9. This threshold is based on empirical results reported in [23] which shows this threshold leads to almost no degradation in accuracy. Then, we use the termination condition threshold as a knob to further trade off accuracy and energy at run-time. To achieve energy consumption adjustment, one can vary the termination threshold at the cost of accuracy degradation. Decreasing the threshold will degrade the accuracy given that more samples are prone to misclassification. However, we will show that it can be used for further energy saving with minimal accuracy degradation.

Figure 2 shows an example on how the result of K-means clustering is utilized to approximate the computation at each neuron. Assume that the clustering algorithm groups the weights of the layer which contains this neuron into 6 clusters based on their values. The figure shows how the weights of the incoming edges to this neuron are grouped after clustering.

As can be seen, the weights belonging to the same cluster have the most value similarity. The computation is then performed iteratively, starting from fetching the clusters with the highest absolute value of weights (i.e., clusters 1 and 2). The output neuron is partially computed by fetching weights W_1 and W_5 from memory. In fact this is done for each neuron in each layer before inference starts at that iteration. If the output of the network is then found to not be acceptable, then the weights of clusters 3 and 4 are fetched in the next iteration. The computation at this neuron is more accurate in this iteration because W_2, W_4, W_6 and W_7 are used. The result of the inference is therefore more accurate.

As the above example shows, by fetching the clusters sequentially and from the more important ones, we can approximate the computations per neuron and reduce the number of memory accesses. We also show that for the majority of the inputs, only one or two iterations are needed to reach a desired accuracy. This translates into energy savings because the overall number of fetches from the off-chip memory are significantly reduced.

It should also be added that AirNN will be most beneficial when the weights do not fit on chip which is the case for many complex neural networks and/or when neural networks are deployed on resource-constrained platforms with limited memory and energy budget such as embedded systems or mobile devices.

Abstract Hardware model. An abstract hardware model of AirNN is also shown in Figure 3. From the hardware side, the weights and input data are stored off-chip. Unlike conventional CNN implementations, in our hardware model the weights are stored by their clusters. During an inference, the network receives the weights and data from memory to perform computations and generates an output. We use a control unit to implement input-dependent approximation and dynamic management of hardware resources in the CNN. The control unit receives its input from the output of the implemented CNN and controls the memory and computational units of the neural network to either load more clusters of weights and run the network for more iterations or generate the final output, terminate the execution of network, and reset the on-chip buffer.

We show in our experiments that well-known CNNs can be approximated with a small subset of important weights for majority of the inputs in their testing datasets. This translates into significant energy saving due to significant reduction in number of fetches of weights from off-chip memory. This is also in part because the number of iterations to reach a final approximation per input is typically small (e.g., only one or two iterations for majority of inputs per CNN as shown in our experiments). Next, we discuss more details about our clustering algorithm..

IV. Details of Clustering

In this work, we use the K-means clustering algorithm [2] to cluster the weights of the trained CNN. The objective of K-means clustering is to minimize the Within-Cluster Sum of Squares of the clustered elements (denoted by WCSS) [18]. This in turn maximizes the similarity between the elements of each cluster. To compute WCSS for a cluster, sum of square is measured using the distances of the elements in a cluster from the centroid of that cluster.

To use the K-means clustering algorithm, we need to first determine the number of clusters (value of *K*). Determining the appropriate *K* for each CNN needs a number of considerations. On one hand, *K* controls the cluster size which in turn translates into the number of weights which are fetched from off-chip memory per iteration of our input-dependent approximation procedure. Having a higher number of clusters translates into smaller cluster size which may lead to more number of iterations per input, and thus a higher energy consumption. Having many clusters in theory is better to provide more flexibility in approximation and higher chance that the elements within the same cluster are similar (i.e., have a smaller WCSS). However too many clusters in practice may lead to performance degradation in the CNN due to overhead of the control unit to manage dynamic approximation and early termination of the network.

We use the following procedure to determine a suitable value of *K* to account for the above-described considerations.

Specifically, we use the Elbow algorithm [2] to determine the number of clusters. In the Elbow algorithm, WCSS is reported as a function of *K*. As expected, WCSS decreases with increase in *K* because the elements in each cluster are more likely to be similar if having more number of clusters. Based on our observations, there exists a point where the rate of reduction in WCSS as a function of *K* will be the most, and after that point, increasing the number of clusters does not significantly reduce WCSS. We choose that point to select the number of clusters which achieves a reasonably-small value of WCSS with a reasonably-small number of clusters.

V. Hardware Design

We use a recent FPGA-based CNN accelerator [25] and modify it to incorporate input-dependent approximation of the CNN at run-time. In this section, we assume the CNN is already implemented in hardware and do not alter its implementation. The new hardware component that we propose is composed of two control units to keep track of the number of iterations, load the appropriate weights (for the right number of clusters) at a given iteration, evaluate the termination condition, stop/reset the network upon termination, and wait for the next input. Figure 4 shows the overall architecture of the (modified) accelerator used in this work. The parts highlighted in orange are added by us. We first discuss the FPGA accelerator from [25] which we refer to as the base accelerator followed by design of control units.

A. Overview of the Base CNN Accelerator

As shown in Figure 4, in the base CNN accelerator [25], the weights and input data are stored in off-chip memory and are transferred to on-chip buffers through a DDR3 interface. In addition, each layer of the network is implemented separately and has its own input/output/weight buffers and computational units, as shown in Figure 4. During inference, the weights and input data are read from the main memory and are stored in the on-chip buffers according to the instructions provided by a Read Weight Controller unit. Then, the computational phase starts to perform one inference. The main functions of this phase are multiplication and accumulation operations in each layer of the network in which each weight is multiplied by its corresponding input neuron. Next, the results are accumulated using adder trees and are passed through a linear/non-linear activation function to generate output neurons.

While the accumulation operation needs to wait for the results of previous multiplication, the multiplications are independent from each other. Therefore, to increase the throughput, there are several parallel two-input multipliers to compute each output neuron. The output neurons of each layer will then be stored in appropriate buffers and serve as an input for the next layer. This parallel implementation using two-input multipliers is key to our because it provides maximum flexibility to do inference with a fraction of the total weights, as we require in our approach.

B. Implementation of Convolutional Layers

A convolutional layer in a CNN is a layer with feature maps as its inputs and outputs. Each convolutional layer takes as an input a set of *C* input feature maps and produces *M* output feature maps. Each output feature map is generated by convolving the input feature maps with a kernel of size $C \times K \times K$. To generate *M* output feature maps, the layer has *M* kernels and the size of the weight matrix for the layer is $M \times C \times K \times K$. To generate each feature map, the kernel slides over the input feature map with stride of size *S*. Then, at each position, the kernel's weights are multiplied with the overlapping values of the input feature map and are summed together to generate one single point in the corresponding output feature map. This process is repeated for all of the *M* kernels to generate *M* output feature maps.

Consider the example in Figure 5 for LeNet5. The first convolutional layer is C1 and receives one 28×28 input feature map, so C=1. It maps this input feature map to 20 output feature maps of size 24×24, so M = 20. This is done by defining a kernel of size 5×5 which slides over the input feature map with a stride of 1. Hence, the weight matrix for this layer is $20 \times 1 \times 5 \times 5$. Similarly, the second convolutional layer is C2 and it receives 20 feature maps as its input and generate 50 feature maps as its output. Hence, the weight matrix for the layer is $50 \times 20 \times 5 \times 5$.

To implement the convolutional layer in hardware, we exploit parallelism over input feature maps and process them simultaneously. To do so, at each cycle, a tile of input feature maps of size $C \times K \times K$ as well as *one* kernel with the same size are fetched to the computational units. Each computational unit consists of $K \times K$ multipliers followed by adder trees with C of them in parallel to process C input feature maps. Then, the computation phase starts and the input feature map

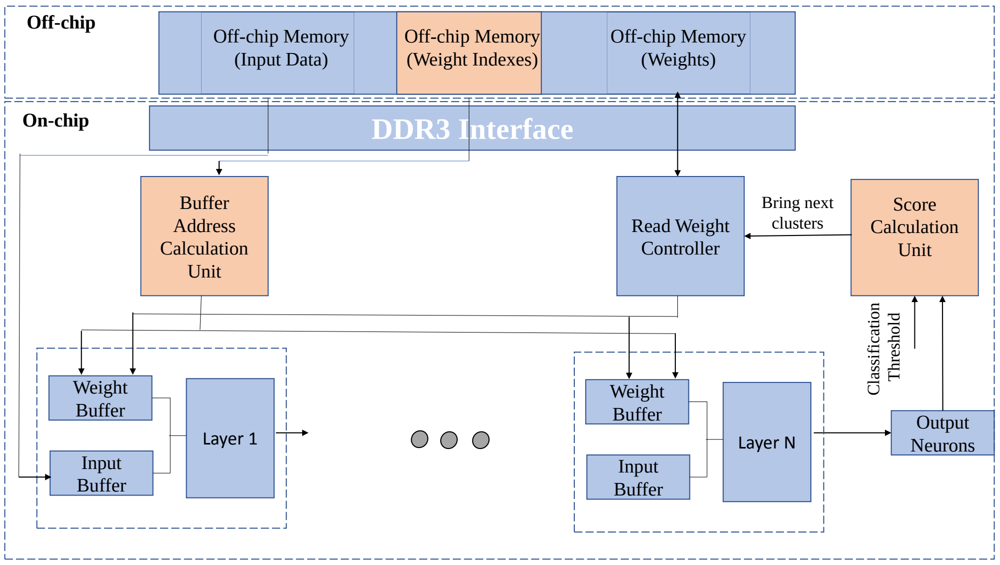

Figure 4: Overall architecture of the modified accelerator. The parts highlighted in orange are added to the base architecture to enable dynamic input-dependent approximation of the CNN.

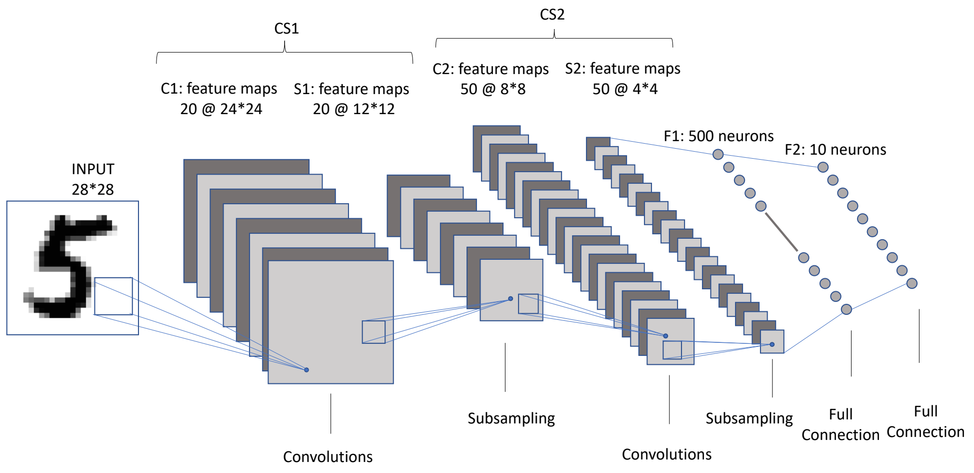

Figure 5: Overview of LeNet5 Architecture.

is mapped to a single position in output feature map. In the next cycle, while the input feature maps remain unchanged, the weights of a new kernel are fetched to the computation units and the second kernel is now processed. Hence, the data flow for the convolution is input-stationary. This process repeats sequentially until all of the kernels are processed and M outputs are generated in the same location for different output feature maps. To further increase throughput and improve data reuse, usually Z tiles of the inputs are processed in parallel to generate Z output neurons. Figure 6 shows the corresponding architecture for C=1. Also, note that to accumulate the results of multiplications, several two-input adders are used, forming a big adder tree.

C. *Design of the Control Units*

Recall, in our input-dependent approximation scheme, for each input there may be several iterations of incremental inference. At each iteration, the weights corresponding to a new batch of weights (two clusters) are fetched until the termination condition is satisfied. This requires design of control units which should support the following features:

- A control unit is needed to load the weights of two clusters from memory to their corresponding on-chip buffers of the computational units, prior to start of the computation. This should be done for each iteration of incremental inference per input. The challenge is that the weights that are fetched (belonging to the same cluster) may be used in different layers of the CNN so it is not necessary that all weights of a layer are brought in, prior to the subsequent layers. We propose a sparse representation scheme to address this challenge.
- A control unit is also needed to evaluate and keep track of the termination condition based on the current output, and then decide if more iterations are needed at the end of each step of incremental inference.

Moreover, AirNN requires the weights to be stored in the

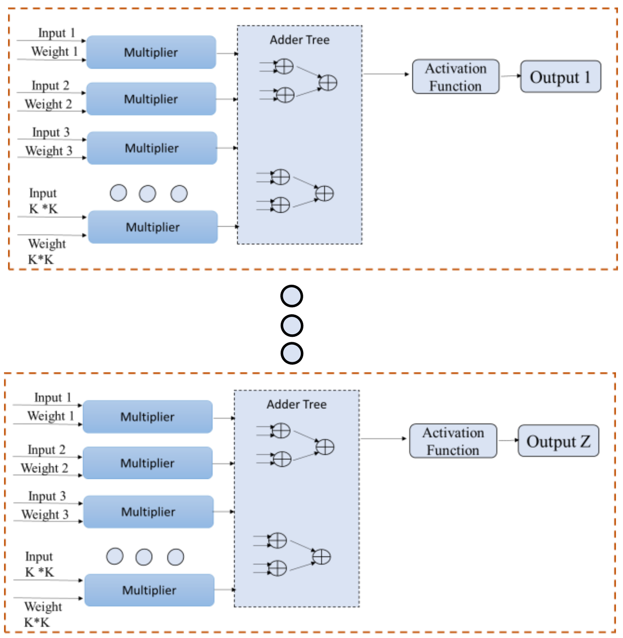

Figure 6: Architecture of a convolutional layer.

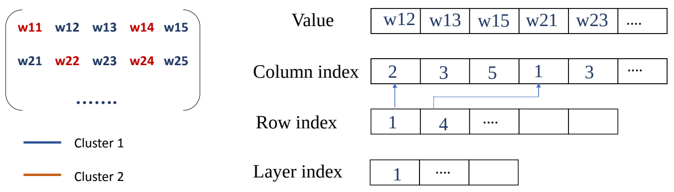

Figure 7: Sparse storage of weights in a cluster with CRS scheme.

order of clusters in the off-chip memory to enable loading a particular cluster each time. This also requires slight modifications to the base accelerator as we discuss shortly.

We note the weight matrix of each layer is sparse initially and gradually gets populated if more clusters are fetched. We therefore store the weights using a sparse matrix representation and in particular discuss how such representation is used when two cluster of weights need to be fetched each time. First, to store a sparse matrix, several schemes including Compressed Row Storage (CRS), Compressed Column Storage (CCS), and Compressed Diagonal Storage (CDS) have been proposed [5]. In this work, we use the CRS scheme because it is applicable to any matrix structure and improves data efficiency.

As shown in Figure 7, in our scheme, we utilize four arrays to store the value and address indexes of the weights in the same cluster. The address is stored in three arrays which keep the row index, column index, and the layer that the weight belongs to. There is a column index entry for each weight value. The row array is shorter and changes whenever the row is changed. The layer array is implemented similar to the row array. The left side of Figure 7 shows the weight matrix in which two clusters are color-coded. The right side shows sparse storage for one cluster as an example. The row index changes to ‘4’ from the second entry to indicate moving to the second row from the 4th element. These indexes are stored in off-chip memory with the weight value, as shown in Figure 4 [12].

In the base accelerator, the Read Weight Controller was responsible to generate the off-chip memory addresses whenever new weights needed to be fetched. Here this unit needs to be modified so it generates the off-chip addresses corresponding to each cluster. We require the weights to be stored in the order of clusters in the off-chip memory. Given this, the modification to the existing Read Weight Controller will be minor so Figure 4 shows this as an existing unit from the base accelerator.

Once the off-chip memory is triggered to read two new clusters of weights, these weights need to be delivered to the corresponding buffers feedings the computational units which may belong to various layers. Here we take advantage of the above-described CRS scheme which already stores an index along with the value of each weight. Specifically, we propose to add a Buffer Address Calculation unit, as shown in Figure 4. It receives as input new weight from the memory which includes value and addresses of its row, column, and layer. This unit then calculates the address of the buffer for each weight using its layer, column, and row indexes.

Once the computations are done and the inference results are generated at the network outputs, the termination condition needs to be evaluated to decide if it is necessary to bring a new batch of weights. Recall, the termination condition required to calculate an accuracy score and determine if it is above a threshold. Here we introduce a score calculation unit which is shown in Figure 4. It receives the outputs of the last layer of the network and generates a control signal which is fed back into the Read Weight Controller unit to trigger fetching a new batch of weights. Recall, the score calculation unit calculates the difference between the two largest output neurons of the network (as defined earlier in Section II) and compares it to the given threshold which can easily be implemented. Then, depending on the generated control signal, the Read Weight Controller unit loads the next batch of clusters or terminates the network execution for the current input , resets the weight buffers and waits for the next input. Per-input approximation and early termination of the network enabled through weight clustering, is the key feature of our proposed framework.

VI. Greedy Interleaving Scheme: Algorithm and Hardware Implementation

To improve the performance of AirNN, we have proposed a greedy interleaving scheme. This scheme exploits the extreme sparsity of weight matrixes, originated from the clustering and mainly targets convolutional layers of the network as they contribute to more than 90% of the computational complexity and power/energy consumption of the network.

In AirNN, many of the weights in the convolutional layer are zero due to clustering, particularly in the earlier stages of the approximation. As a result, when one kernel is processed, the multipliers corresponding to the zero weights are idle (Recall the hardware implementation of convolutional layers discussed in V-B.). The idle multipliers can be utilized efficiently to exploit parallelism across output feature maps in addition to input ones and process more than one kernel. It subsequently improve the performance of the accelerator. To achieve this, we propose a greedy interleaving scheme which interleaves the kernels with respect to the location of their non-zero

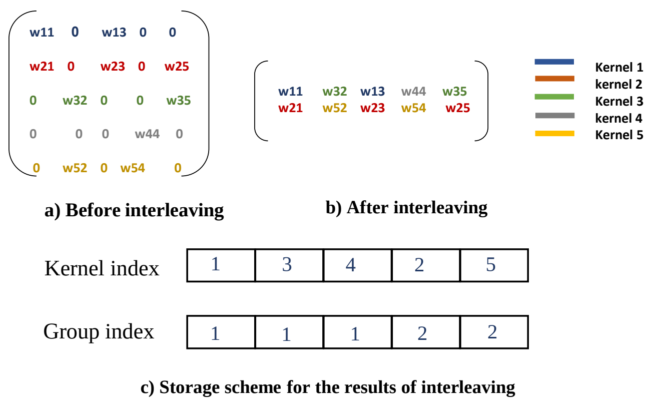

Figure 8: An example of the proposed greedy interleaving scheme.

weights and enables to process several kernels within one single computational unit.

A. Details of the Greedy Interleaving Scheme:

Here, we discuss the proposed interleaving scheme in more details. We first convert the $4D$ weight matrix of size $M \times C \times K \times K$ to a $2D$ matrix of size $M \times N$, where $N = C \times K \times K$. With this representation, each row corresponds to one kernel and our goal is to interleave the rows with respect to their non-zero elements. For this, we start from the first row of the matrix (i.e., the first kernel), called as the reference row. The reference row then should be compared against all the remaining rows to find the rows (kernels) it can interleave with. Starting from the second row, our algorithm traverses through every column of these two rows and check whether they can be interleaved or no. Two kernels can be interleaved if there is no "overlap" in the position of their non-zero element, meaning that they are not using the same multiplier in hardware. If two rows can be interleaved, the algorithm updates the reference row by interleaving it with the second one and go to the third row. This process repeats until the algorithm traverses through all of the rows. At this point and after processing the first row, we have the first group of interleaved kernels which can be processed simultaneously. Note that we need to keep track of the rows that are interleaved and grouped together and do not consider them in next rounds of the algorithm. We then go to the next "non-interleaved" kernel, select it as the reference row, and repeat the above process.

Note that, as we go through different number of iterations, the kernels that are interleaved with each other are changed, given that in different iterations different fraction of weights are included in computations. Therefore, we need to apply the interleaving scheme to each iteration separately. Based on our results, the network is extremely sparse at the first three iterations and becomes considerably populated at the last two or three iterations. Hence, the interleaving scheme can be beneficial only in the first *three* iterations.

Figures 8a and 8b show an example on the proposed interleaving scheme for the weight matrix of size 5×5. Different rows are color-coded to distinguish different kernels. As mentioned above, we start from the first row and compare it against all of other 4 remaining kernels. Comparing it against the second row, they can't interleave with each other given that they both have a non-zero weight in their third column. In contrast, it can be interleaved with third and fourth rows. Recall that each time two rows interleave, we will update the reference row. Once the first reference row is processed completely, we pick the next row provided that it is not already interleaved with other rows in previous iterations. In this case, we need to pick the second row and interleave it with the last row. As the figure shows, after applying the interleaving scheme, 5 rows (representing 5 kernels) are divided into 2 groups, one containing rows 1, 3, and 4 and the other containing rows 2 and 5.

B. Hardware Modifications for Implementation of the Interleaving Scheme

Here, we discuss the details of implementing the proposed interleaving scheme which aims to accelerate the execution of convolutional layers in the earlier iterations of AirNN by exploiting the sparsity in the weight matrices observed in the those iterations. More specifically, our design enables processing more than one kernel at a time and thus improves the execution time of computationally-intensive convolutional layers.

To implement the proposed interleaving scheme in hardware, we need to modify the underlying accelerator while considering the following issues:

1) Once the interleaving scheme is applied, one *single* computational unit is processing a group of kernels. As a result, the multipliers' outputs belong to different kernels rather than one kernel. Hence, the computational unit should be modified in accumulation phase to only accumulate the results of multiplications that belong to the same kernel. This is in contrast to the conventional design of computational units where all multiplications' results are added. Furthermore, one computational unit is now generating more than one output neuron, each belonging to a different output feature map. Therefore, the accelerator controller should be able to keep track of the generated output neurons and store them appropriately.
2) Since interleaving the kernels is done once offline, we need to store the results of interleaving appropriately on hardware. More importantly, the Read Weight Controller should be able to read the weights of the interleaved kernels appropriately.

Modification of computational units. To address the first issue, we need to modify the architecture of each computational unit in accumulation phase, so that multiplication results are added up appropriately. This can be done by limiting the maximum number of interleaved kernels and inserting a few multiplexers in accumulation phase.

To be more specific, the modified design is shown in Figure 9 for a kernel of size 5×5, which is the kernel size of LeNet5 and CIFAR10 studied in this work. As can be seen, the base computational unit has 25 multipliers. To enable interleaving in hardware, we limit the "maximum" number of kernels to be interleaved to "four kernels". Accordingly, the base computation unit is divided into four blocks, each block having six multipliers with the last one having seven multipliers. Here,

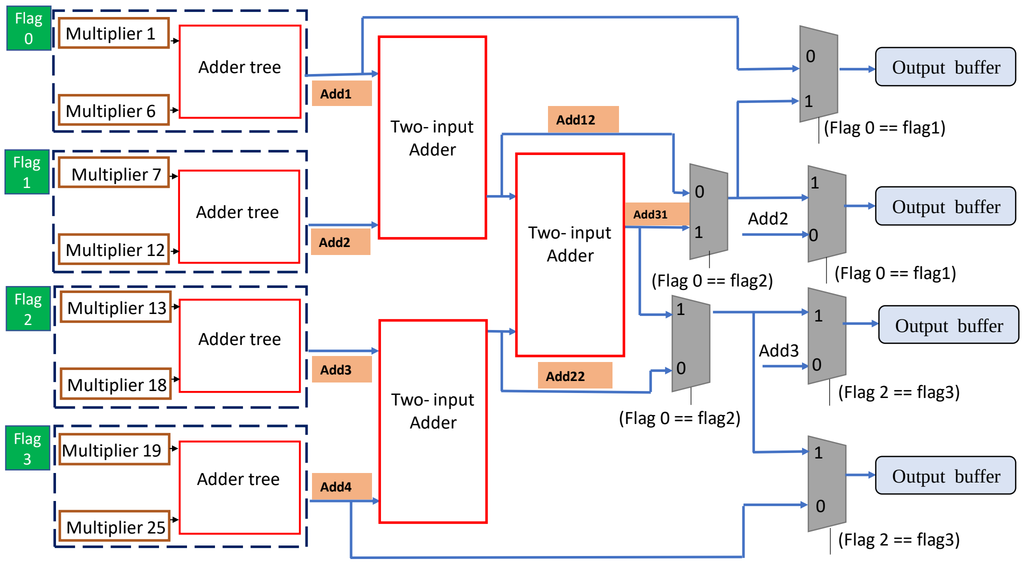

Figure 9: Overview of the modified computational unit.

we add one more constraint to our interleaving scheme: the kernels interleaving with each other should occupy multipliers in the granularity level of blocks; So, a kernel with less than 6 non-zero weights occupies one block, while a kernel with more than 6 and less than 12 non-zero weights occupies two blocks. Following the proposed granularity, each kernel occupies either one, two, or four blocks in the computation unit. Also, each block has a flag which is used to indicate whether or not two consecutive blocks are occupied by the same kernel. If the kernel is changed from one block to another, then the flag bit will be inversed. As a result, each rising/falling edge of the flag bits implies a different block.

The proposed block granularity enables us to accumulate the results of multiplications appropriately. Regardless of how many kernels are interleaved, the results of multiplications within each block should be accumulated and as a result, four outputs are generated. Then, depending on the way the kernels are interleaved, three cases can happen:

Case 1. No kernel occupies more than one block. In this case, we only need to transfer the four generated results, shown as Add1, Add2, Add3, and Add4, to output buffers and no other addition is required. Accordingly, the flag bits are switching from one block to another and the multiplexers enable the architecture to bypass next adders and generate the appropriate outputs.

Case 2. The kernel is occupying four blocks (i.e., no interleaving has happened). In this case, we need to accumulate all the partial sums generated by each block (i.e., Add1, Add2, Add3, and Add4) and transfer Add31 to output buffers as shown in Figure 9. Given that all blocks have the same flag, the multiplexers enable transferring Add31 to output buffers.

Case 3. Atleast one kernel is occupying two blocks (i.e., either blocks one and two or blocks three and four). In this case, we need to have one more addition to accumulate the partial sums and Add21 and Add22 should be transferred to output buffers.

Selecting appropriate number of blocks. Number of blocks (which corresponds to the number of the kernels processed simultaneously) in our design depends on two factors: 1) Size of the kernel which determines the number of available multipliers. 2) Sparsity of the weight matrix in the first initial iterations which determines the distribution of the number of the interleaved kernels at each iteration. Also, to appropriately sum the partial additions with the help of multiplexers, we need to have an even number of partial additions at each level of adder tree in the computational unit (such as Add1, Add2, Add3, and Add4 in the first level and Add12, Add22 in the second level). Hence, the number of blocks should be a power of two (i.e., two, four, eight, etc). Using the above considerations, we can determine the number of blocks for each DNN model.

For instance, assume a convolutional layer in a given network has a kernel size of 4×4. This results in 16 multipliers per processing unit. Also, the number of blocks should be less than or equal to the total number of available multipliers in the processing unit and is constrained to power of two (i.e., two, four, eight, and sixteen stages in this case). To pick the most appropriate value for the number of blocks, we need to look at the distribution of interleaved kernels. The most frequent number of interleaved kernel should be then rounded to the nearest power of two which in turn determines the number of blocks.

In the case of LeNet5 and CIFAR10, we observe in our profiling that interleaving up to four kernels happens most frequently. Hence, we divided each computational unit to 4 blocks to minimize the overhead of the multiplexers, while achieving the desired speedup in execution time of convolutional layers.

Storing interleaving results. To address the second issue, we need to store the interleaving results and enable the Read Weight Controller to fetch appropriate weights considering the interleaved kernels. For this, we first store number of groups per iteration and kernel indexes of the kernels belonging to the same group for each iteration. More specifically, at each iteration, there are two arrays where the first array stores the “kernel index” such that the kernels in one group appear right after each other. The second array stores "group index" which distinguishes the groups and separates one group of interleaved kernels form the next group. The size of each array is the total number of kernels in the underlying layer. Recall the example in SectionVI, Figure 8. There, we had 5 kernels which resulted in two groups after interleaving. You can see the corresponding interleaving arrays in Figure 8c. In the first array, the kernels belonging to same group appear after each other (i.e., 1, 3, 4 and 2, 5) and the second array distinguishes the groups.

Next, we need to enable the the Read Weight Controller to bring the weights of the kernels belonging to the same group. It can be achieved with the help of the Compressed Row Storage scheme which was discussed earlier. It enables us to store the weights by their cluster (see Figure 7). Recall that in this storage scheme, the row index changes whenever we move from one row to another. In addition, as we discussed in Section VI, we can view the 4D weight matrix as a 2D one, where each row represents a kernel. Hence, the change in the row index reflects a change in the kernel in process. On the other hand, we have stored the kernel indexes for the kernels belonging to the same group. Putting all these together, the Read Weight Controller should be slightly changed to read the kernel weights by their groups rather than reading them consecutively. As a result, we are able to fetch weights for several kernels in one group to the computational units to start the inference.

VII. Simulations Results

In our experiments we used five well-known CNNs namely, LeNet300-100 and LeNet5 (running on MNIST dataset), CIFAR10 (running on CIFAR10 dataset), VGG-16 (running on ImageNet dataset), and MobileNet-V2 (running on ImageNet dataset). These networks are first implemented and trained with Neupy [9] and TensorFlow. Then, we quantized the trained floating point weights to 32 bits and imported each network in Matlab to verify post-quantization accuracy. This framework was used to compute accuracy for different clustering scenarios as determined by our approach.

To measure energy consumption of the network during inference, we first adapted the energy consumption of the components in the network from [14] and [19]. We used Synopsys Design Compiler to measure energy consumption of our proposed control units including the score margin calculation unit and the address calculation unit. The main memory of the network is DDR3 DRAM in 65nm technology [25]. We used NVSIM simulator [4] to measure energy consumption of DRAM accesses. The results are summarized in Table I.

TABLE I: Energy consumption of different components.

Component	Energy (pJ)
Two-input multiplier	1.00
Two-input adder	0.40
ReLU	0.90
Max-pool	1.20
Address calculation unit	0.35
Score margin calculation unit	0.27
DRAM memory access	1950.00

As can be seen, the majority of energy is consumed by DRAM accesses. In our experiments, we show significant decrease in total number of DRAM accesses using input-dependent approximation scheme. This is despite the fact that multiple inference iterations may be needed to approximate per input.

A. Number of Clusters in K-means Clustering

The first step for AirNN is clustering the weights of a pre-trained CNN in which the weights are grouped into K clusters based on their similarities (i.e., their values and signs). As explained before, to determine the value of K for a CNN, we used the Elbow method to measure the With-in Cluster Sum of Squares (WCSS) for each CNN layer and then calculated the average WCSS over all the layers as a function of number of clusters. Figure 10 plots the average WCSS as a function of K for four CNNs studied in this work. As can be seen, a small number of clusters can be found by looking at the point where the rate of decrease in WCSS is maximum. This corresponds to the value of K for LeNet300-100 to be 8 while it is 10 for LeNet5, CIFAR10, VGG-16 and MobileNet-V2 from the figure. We note, for MobileNet-V2, clustering was applied to all the layers except the 19 point-wise convolutional layers with 1×1 kernels because these layers only had a single weight so the clustering algorithm was not applicable to them.

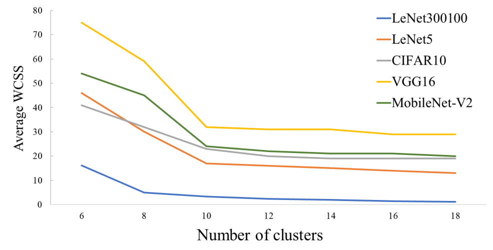

Figure 10: Within-Cluster Sum of Squares (WCSS) of weights as a function of number of clusters (K).

The results obtained form the Elbow method guarantee that the weights within each cluster have the most similarity to each other and hence the clusters can be fetched with respect to their importance (i.e., their contribution to generate the output result). Also, in the proposed iterative approach, the accuracy will improve as more clusters are included. However, in addition to having impact on performance, the number of clusters will determine the energy saving of the iterative framework as it determines the number of iterations required per input. Thus, we need to gain more insight on the relation between number of clusters and network energy consumption. On one hand, more clusters may be desirable as they provide more opportunities for the network to be terminated if the current output is acceptable. Moreover, with more clusters, the number of weights within each cluster will reduce as the total number of weights are constant. As a result, an acceptable output may be achieved with lower energy consumption. On the other hand, more clusters and subsequently more iterations can degrade the performance of the network due to latency overhead incurred by the iterative framework.

To consider this tradeoff while determining the number of clusters, we have conducted further experiments where number

TABLE II: Accuracy versus fraction of weights used per iteration for LeNet300-100 for different number of clusters for LeNet300-100.

Number of Iterations	Fraction of Weights Included	Accuracy
8 clusters		
Base architecture	1	97.41
iteration 1	0.01	45.90
iteration 2	0.10	93.80
iteration 3	0.31	97.16
iteration 4	1	97.41
10 clusters		
Base architecture	1	97.41
iteration 1	0.01	33.10
iteration 2	0.05	95.90
iteration 3	0.16	97.29
iteration 4	0.37	97.39
iteration 5	1	97.41
12 clusters		
Base architecture	1	97.41
iteration 1	0.01	28.90
iteration 2	0.02	75.23
iteration 3	0.09	96.50
iteration 4	0.20	97.00
iteration 5	0.40	97.39
iteration 5	1	97.41

of clusters are varied for a particular network. (Note that the minimum number of clusters is the one obtained by the Elbow method). Then, for different number of clusters, the fraction of weights at each iteration and their corresponding accuracy are measured. As an example, Table II reports the accuracy versus fraction of weights used per iteration for LeNet300-100 under various number of clusters. Then, we compared them and chose the one that results in almost the same accuracy compared to the others using smaller fraction of weights. In the case of LeNet300-100, we need at least 30% of weights to achieve an acceptable accuracy with 8 clusters (i.e., less than 1% degradation compared to the base case), while it is around 16% with 10 clusters and 20% for 12 clusters. Hence, 10 clusters is chosen for LeNet300-100.Using the same approach, we have chosen 12 clusters for LeNet5 and 10 clusters for CIFAR10, VGG-16, and MobileNet-V2.

Note that the number of *fetched* clusters is determined by the termination condition and is input-dependent and equal to twice the number of iterations. It is at most K per input. In fact we show in our experiments it is often much smaller than K.

B. Accuracy vs Energy Tradeoff during Incremental Inference

In this experiment, we show the tradeoff between various parameters during incremental inference using AirNN.

Figure 11 shows classification accuracy, cumulative energy consumption (since iteration 1), and fraction of used weights, as a function of number of iterations for the five CNN models. The orange/blue bars show the accuracy and cumulative energy when averaged across all the inputs for which AirNN has not yet been terminated. (The majority of the inputs will terminate in less than 4 iterations, as we will report in another experiment.) More specifically, the reported energy number in each iteration is cumulative, meaning it includes the energy that is consumed in all the previous iterations, and is normalized to the energy consumption of the base model. Note that if AirNN is terminated with fewer iterations for an input, then that input is not counted towards energy consumption in future iterations. To compute the classification accuracy at a particular iteration, we only considered the inputs for which AirNN terminated up to that iteration. The classification accuracy for the rest of the inputs was set to 0 because AirNN was not yet terminated for them. The red curve shows the fraction of used weights in an iteration which is independent of the input (but can vary from one network to another).

As can be seen, executing more iterations increases energy consumption due to fetching more weights but in turn the accuracy is improved. The fraction of fetched weights directly translates to the number of DRAM accesses which is the dominant source of energy consumption, as was previously reported in Table I.

Next, in Table III, we report the number of required iterations (until termination), cumulative energy consumption, and fraction of the fetched weights to reach within 3% accuracy of the base model for the five CNN models. The classification accuracy, before and after applying AirNN, is also reported in the last two columns of the table. Note, the numbers for each network is an average, when running AirNN across all the test inputs except fraction of weights which is independent of the input.

As the results in Table III show, AirNN is able to significantly reduce the energy consumption of the network without any significant loss in the accuracy. As an example, in case of LeNet5, we can reach within 2% accuracy of the base model after 3 iterations with using at most 45% of the weights and consuming only 49% of the energy, compared to the base model. A similar trend can be seen for the other CNNs.

Figure 13 shows the distribution of percentage of input samples with respect to the number of required iterations for AirNN to terminate, for each CNN. As can be seen, for the majority of the inputs, AirNN terminates with only two or three iterations. This means, for the majority of the inputs each CNN can be approximated with significantly fewer non-zero weights, translating into a significant energy saving.

C. Adjustment of the Score Threshold for Accuracy versus Energy Tradeoff

In this section, we have run some experiments to investigate the impact of changing the score threshold on energy consumption and accuracy. To achieve this goal, we have decreased the score threshold to values less than 0.9 and measured the accuracy and normalized energy consumption corresponding to each score threshold. The results are shown in Figure 12 for four networks. As expected, reducing the score threshold will decrease the accuracy of the network given that with smaller score threshold, more samples are prone to misclassification. In the meantime, it will also decrease the energy consumption of the network, because many inputs require fewer number of iterations to be classified. As the figure shows, by decreasing the score threshold from 0.9 to 0.8, the network accuracy is degraded around 5%. Also, AirNN achieves more energy saving for the inference. As an example, in the case of CIFAR10, the normalized energy consumption is reduced from 0.47 to 0.38 of the base network.

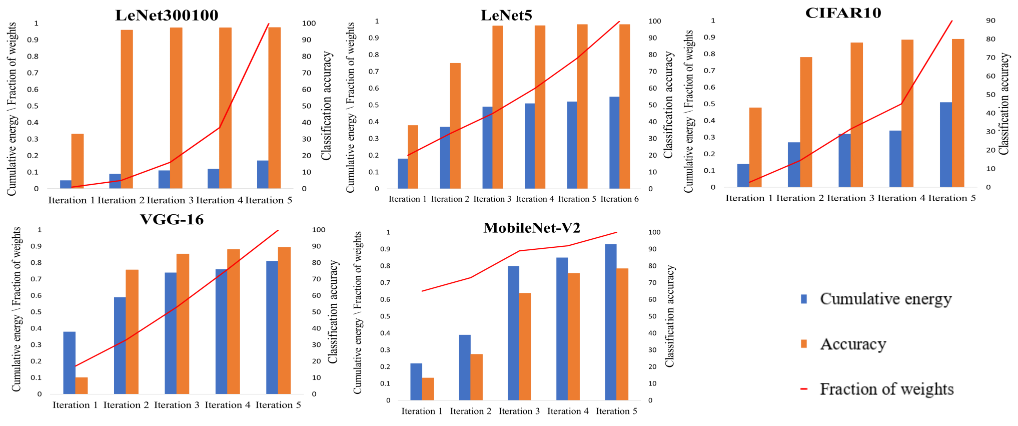

Figure 11: Accuracy, cumulative energy consumption, and fraction of used weights, as a function of number of iterations for various CNN models. The numbers shown for each iteration are averaged across all inputs for which AirNN has not yet been terminated by that iteration. (The majority of the inputs only require a couple of iterations, as will be shown later.) As can be seen, executing more iterations increases energy consumption due to fetching more weights but the accuracy improves in return.

TABLE III: Results of applying AirNN to various neural network models. The numbers for each network is an average, when running AirNN across all the tested inputs.

Network	#Required iterations	Fraction of weights	Normalized cumulative energy	Accuracy (Before AirNN)	Accuracy (After AirNN)
LeNet300-100	3	0.16	0.11	97.41	97.29
LeNet5	3	0.45	0.49	98.40	97.11
CIFAR10	3	0.35	0.32	80.10	78.10
VGG-16	4	0.76	0.76	89.44	88.01
MobileNet-V2	4	0.92	0.85	78.30	75.60

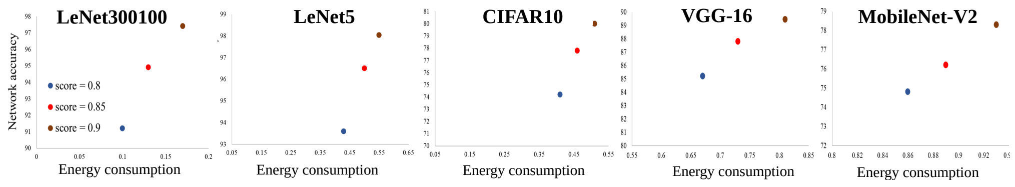

Figure 12: Accuracy and energy consumption as a function of score.

TABLE IV: Results of applying AirNN to two already-pruned networks to achieve additional energy savings.

Network	#Required iterations	Fraction of weights	Normalized cumulative energy	Accuracy (Before AirNN)	Accuracy (After AirNN)
VGG-16 (pruned with [8])+AirNN	4	0.78	0.78	88.90	87.30
VGG-16 (pruned with [11])+AirNN	4	0.84	0.81	87.90	86.60

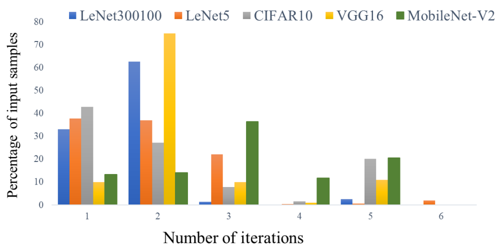

Figure 13: Distribution of number of inputs based on the required number of iterations of our algorithm for different CNNs.

D. Assessment of AirNN on Already-Pruned DNNs:

As mentioned earlier, AirNN may be combined with existing model pruning techniques to further reduce the energy consumption of the network by pruning a subset of weights dynamically and at run-time based on its received input.

To show this, we first considered two different pruned models of VGG-16 (pruned using two recent works [8] and [11])[1]. Next, we applied AirNN to each pruned model to cluster the remaining, post-pruned weights in each case. The clustering led to 10 groups for the pruned model of [8] and 12 groups for the pruned model of [11]. Then, we ran the networks for different number of iterations and measured the energy consumption and classification accuracy. The results are summarized in Table IV. Note that cumulative energy and accuracy are averaged across all test inputs. As can be seen, after 4 iterations, both pruned models reach within 2%

[1]We directly used the pruned and retrained models which were publicly available.

accuracy of the base models by consuming only 0.78 (in [8]+AirNN) and 0.81 (in [11]+AirNN) of energy, compared to the corresponding base models.

E. Results of the Proposed Interleaving scheme

In this section, we provide the results of applying the interleaving scheme to the convolutional layers of LeNet5, CIFAR10 and VGG-16. To apply the interleaving scheme, we first extracted the weight distribution of the kernels at different iterations for each network. Then, we have applied the interleaving scheme to non-zero kernels while limiting the maximum number of interleaved kernels to 4.

Results of LeNet5 and CIFAR10. The results are shown in Tables V and VI for LeNet5 and CIFAR10, respectively. The first column of the table reports the number of iterations. The second column reports the number of non-zero kernels for each layer per iteration. Note that due to clustering, it is possible to have a kernel with all zero weights for a specific iteration. These kernels are not considered when applying the interleaving scheme, given that they are not executed in the base iterative framework. As can be seen, for the first convolutional layer of LeNet5, there are 3 and 16 non-zero kernels in the first and second iterations, respectively (while the original number of kernels in non-iterative framework is 20). The third column reports the number of interleaved kernel achieved once we apply interleaving. This number reflects the total number of kernels which should run *sequentially* while each of these interleaved kernels consists of up to 4 individual kernels, running in parallel. The last column reports the speedup achieved per layer after interleaving the kernels.

As the results show, in the first iteration up to 3.5x and 2.9x speedup can be achieved across different convolutional layers of LeNet5 and CIFAR10. The performance speedup per layer decreases in the second iteration and it gets saturated in the third iteration. However,as the majority of inputs go through one or two iterations, the proposed interleaving scheme will be beneficial to improve network performance.

Results of VGG-16. Unlike LeNet5 and CIFAR10, the proposed interleaving scheme is not effective at speeding up the inference for VGG-16. It is because the sparsity of convolutional layers is not significant in VGG-16 as we reach the second iteration. Low sparsity of weight matrices in convolutional layers originates from the distribution of the weights in these layers. Unlike LeNet5 and CIFAR10, in VGG-16, a significant fraction of the weights have relatively high absolute values. Given that our clustering algorithm is value-based, these weights are fetched to the computational units in the very first iterations. This in turn reduces the sparsity which is the key to our interleaving scheme.

F. Performance Evaluation

Latency measurement setup. In this section, we measure the latency of AirNN before and after applying the interleaving scheme for LeNet5, CIFAR10 and MobileNet-V2. To measure the latency of the network during inference, we first adapted the latency numbers of each individual computation unit including Pooling and ReLU from [19], [14]. We used

TABLE V: Results of applying the interleaving scheme to LeNet5.

Iteration	# of non-zero kernels (Before interleaving)	# of grouped kernels (After interleaving)	Speedup (Per layer)
Convolutional layer 1			
iteration 1	3	1	3.0x
iteration 2	16	5	3.2x
iteration 3	20	17	1.2x
Convolutional layer 2			
iteration 1	50	14	3.6x
iteration 2	50	23	2.1x
iteration 3	50	48	1.0x

TABLE VI: Results of applying the interleaving scheme to CIFAR10.

Iteration	# of non-zero kernels (Before interleaving)	# of grouped kernels (After interleaving)	Speedup (Per layer)
Convolutional layer 1			
iteration 1	11	5	2.2x
iteration 2	28	17	1.6x
iteration 3	32	29	1.1x
Convolutional layer 2			
iteration 1	32	11	2.9x
iteration 2	32	23	1.4x
iteration 3	32	32	1.0x
Convolutional layer 3			
iteration 1	64	24	2.6x
iteration 2	64	51	1.2x
iteration 3	64	60	1.0x

Synopsys Design Compiler to measure energy consumption of our proposed control units. We have also used NVSIM [4] to measure latency of DRAM accesses. Table VII summarizes the latency of arithmetic operations in 45nm technology and memory accesses in 65nm technology [25]. Then, according to the architecture of the underlying layers in the network such as number of input channels, number of output channels, and size of the convolutional kernels, we build our analytical model to map the network into the hardware. The underlying hardware is modeled according to Google's Tensor Processing Unit (TPU) design [10] which is also shown in Figure 3. Since the mapping is done based on the network architecture, our analytical model accounts for the available hardware resources. Lastly, once the model is built, we measure the network latency as the total number of clock cycles that is required to finish the inference for a given input. We note that the number of iterations per inference varies from one input to another.

In the case without interleaving, the latency is increased with increase in the number of iterations given that each input needs to repeat the inference phase for several times. The performance of the iterative framework can be alleviated by the proposed interleaving scheme; Because it allows the hardware to utilize idle multipliers and exploits parallelism over both input and output feature maps.

AirNN execution time. Table VIII summarizes the cumulative execution time (since iteration 1), as a function of number of iterations. The runtimes are normalized to those of the corresponding base models (which are non-iterative). Execution time at each iteration is measured by calculating the number of required clock cycles to finish the inference up to that iteration, when averaged over all the test inputs. The execution time of

TABLE VII: Latency of various arithmetic operations and memory accesses.

Component	Latency (clock cycle)
Two-input multiplier	1
Two-input adder	1
ReLU	1
Max-pool	2
Address calculation unit	2
Score margin calculation unit	1
DRAM	120

TABLE VIII: Cumulative execution time of iterative CNNs after interleaving, normalized to the runtime of the base model (which is non-iterative).

Normalized execution time			
Number of iterations	**LeNet5**	**CIFAR10**	**MobileNet-V2**
Iteration 1	0.28	0.37	0.52
Iteration 2	0.34	0.50	0.59
Iteration 3	0.43	0.58	0.64
Iteration 4	0.52	0.65	0.72
Iteration 5	0.58	0.70	0.83
Iteration 6	0.63	-	-

the iterative framework after interleaving is normalized to that of the iterative framework before interleaving. Note that if AirNN is terminated with fewer iterations for an input, then that input is not counter towards the execution time in future iterations.

As the results show, the proposed interleaving scheme is successfully able to reduce the total execution time of the network over iterations. More importantly, although the interleaving scheme is only applied to the first three iterations of the inference, we still have performance improvement at later iterations. This is because a significant speedup can be achieved in the earlier iterations due to extreme sparsity of weight matrices.

Lastly, we have measured the total execution time of LeNet5, CIFAR10 and MobileNet-V2 when running on the test dataset. The test dataset consists of 10,000 images for each of these networks. Similar to the setup for energy measurement, the execution time is measured with respect to the number of inputs at each iteration. Hence, the inputs which are classified correctly at a particular iteration are not considered in the next iterations anymore. We have measured the execution time under three different configurations: 1) non-iterative CNNs where each input needs only one round of inference, 2) AirNN without interleaving, and 3) AirNN with interleaving. Figure 14 shows the normalized execution time for these networks. Note that execution time is normalized to non-iterative framework. As the results show, AirNN increases the total execution time of LeNet5, CIFAR10, and MobileNet-V2 by 1.86, 1.6, and 1.9, respectively. It in turn can degrade the performance of network and decrease energy efficiency. This is while when the interleaving scheme is applied, the total execution time of AirNN (i.e., the total number of required clock cycles) is reduced to 0.37, 0.47, and 0.71 of the non-iterative framework for LeNet5, CIFAR10, and MobileNet-V2, respectively.

With this observation, *the proposed input-dependent framework with kernel interleaving is successfully able to reduce energy consumption and latency of CNNs compared to the base accelerator without significant degradation in accuracy.*

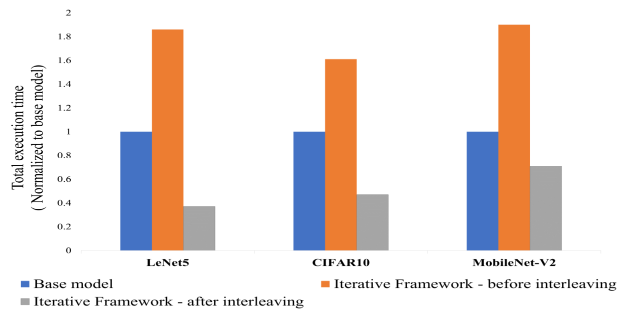

Figure 14: Normalized execution time of CNNs.

G. Energy Overhead of the Proposed Interleaving Scheme

As described in section VI-B, the proposed interleaving scheme can be implemented in hardware by slightly changing the architecture of each computation unit and at the cost of adding six 2-1 multiplexers. As a result, the energy overhead of implementing the interleaving scheme is negligible and the energy consumption of AirNN will almost remain the same.

VIII. Conclusions

This work is the first to propose an input-dependent approximation of CNN in hardware, without actually altering the CNN implementation and by means of few additional control units. Using our novel iterative framework called AirNN, we showed in our experiments that the majority of inputs in popular CNNs can be inferred with a fraction of the weights. With less than 3% loss in accuracy, AirNN is able to achieve around 49% energy saving. In addition, we proposed a greedy interleaving scheme which is able to improve the performance of AirNN by exploiting the extreme sparsity of weight matrices and by utilizing the idle multipliers. As a result, the execution time per input is not increased despite the fact that each input may require more than one round of inference.

Acknowledgments

This research was supported by Award Number 1812600 from National Science Foundation and by Semiconductor Research Corporation GRC Task Number 2845.001.

References

[1] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R.K. Gupta, and H. Esmaeilzadeh. SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks. In *International Symposium on Computer Architecture*, pages 662–673, 2018.
[2] P. Bholowalia and A. Kumar. A clustering technique based on Elbow method and k-means in WSN. *International Journal of Computer Applications*, 105, 2014.
[3] R. Ding, Z. Liu, R. D. Blanton, and D. Marculescu. Quantized deep neural networks for energy efficient hardware-based inference. In *IEEE Asia and South Pacific Design Automation Conference*, pages 1–8, 2018.
[4] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. NVSIM: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012.
[5] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk. Accelerating SpMV on FPGAs by compressing nonzero values. In *International Symposium on Field Programmable Gate Arrays*, pages 64–67, 2015.
[6] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135 –1143, 2015.

[7] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Design, Automation, and Test in Europe Conference*, pages 1474–1479, 2017.

[8] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision*, pages 1389 – 1397, 2017.

[9] http://neupy.com/pages/home.html.

[10] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, pages 1 – 12, 2017.

[11] J. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *International Conference on Computer Vision*, pages 5058 – 5066, 2017.

[12] M.Hemmat and A. Davoodi. Dynamic reconfiguration of CNNs for input-dependent approximation. In *International Symposium on Quality Electronic Design*, pages 176–182, 2018.

[13] M.Hemmat and A. Davoodi. Power-efficient ReRAM-aware CNN model generation. In *International Conference on Computer Design*, 2018.

[14] M. Nazemi, G. Pasandi, and M. Pedram. Energy-efficient, low-latency realization of neural networks through boolean logic minimization. In *IEEE Asia and South Pacific Design Automation Conference*, pages 274 – 279, 2019.

[15] K. Neshatpour, F. Behnia, H. Homayoun, and A. Sasan. ICNN: An iterative implementation of convolutional neural networks to enable energy and computational complexity aware dynamic approximation. In *Design, Automation, and Test in Europe Conference*, pages 551 – 556, 2018.

[16] P. Panda, A. Sengupta, and K. Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *Design, Automation, and Test in Europe Conference*, pages 475 – 480, 2016.

[17] E. Park, D. Kim, S. Kim, Y. Kim, G. Kim, S. Yoon, and S. Yoo. Big/little deep neural network for ultra low power inference. In *CODES +ISSS*, pages 1624–132, 2015.

[18] M.S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing. LookNN: Neural network with no multiplication. In *Design, Automation, and Test in Europe Conference*, pages 1779 –1784, 2017.

[19] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *International Symposium on Computer Architecture*, pages 14–26, 2016.

[20] F. S. Snigdha, S. D. Manasi, J. Hu, and S. S. Sapatnekar. SeFAct2: Selective feature activation for energy-efficient CNNs using optimized thresholds. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[21] F.S Snigdha, I. Ahmed, S.D. Manasi, M.G. Mankalale, J. Hu, and S.S Sapatnekar. SeFAct: selective feature activation and early classification for CNNs. In *IEEE Asia and South Pacific Design Automation Conference*, pages 487–492, 2019.

[22] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li. Prediction based execution on deep neural networks. In *International Symposium on Computer Architecture*, pages 752 – 763, 2018.

[23] H. Tann, S. Hashemi, R. I. Bahar, and S. Reda. Runtime configurable deep neural networks for energy-accuracy trade-off. In *CODES + ISSS*, pages 34:1–34:10, 2016.

[24] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. AxNN: energy-efficient neuromorphic systems using approximate computing. In *International Symposium on Low Power Electronics and Design*, pages 27–32, 2014.

[25] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *International Symposium on Field Programmable Gate Arrays*, pages 161–170, 2015.

Maedeh Hemmat received the M.S. degree in electrical engineering from University of Tehran in 2016. She is currently pursuing the Ph.D. degree in Computer Engineering with the University of Wisconsin-Madison. Her research interests include power-efficient realization of Deep Neural Networks (DNNs) on resource-constrained platforms, implementation of neural networks using new emerging technologies, and design space exploration of DNNs for reconfigurable and approximate implementation. She was a recipient of the Electrical and Computer Engineering Chancellor's Opportunity Fellowship, the CRA-W Grad Cohort for Women workshop, and the A. Richard Newton Young Student Fellowships.

Joshua San Miguel is an Assistant Professor at the University of Wisconsin-Madison. His research interests include approximate computing, intermittent computing and interconnection networks. San Miguel received a PhD in electrical and computer engineering from the University of Toronto.

Azadeh Davoodi (SM'13) is Professor of Electrical and Computer Engineering at the University of Wisconsin–Madison. Her primary research interests are in Electronic Design Automation and debug of Integrated Circuits, hardware security, and in Design Automation of Things, in general. Azadeh is recipient of a 2011 NSF CAREER award. Her work with collaborators received the best paper awards of the 2015 ACM Transactions on Design Automation of Electronic Systems and Best Paper nomination at the 2010 Design Automation Conference.