SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks

Vahideh Akhlaghi[*] Amir Yazdanbakhsh[*†] Kambiz Samadi[‡] Rajesh K. Gupta Hadi Esmaeilzadeh
Alternative **C**omputing **T**echnologies (**ACT**) Lab
[†]Georgia Institute of Technology [‡]Qualcomm Technologies, Inc. University of California, San Diego
vakhlaghi@eng.ucsd.edu a.yazdanbakhsh@gatech.edu ksamadi@qti.qualcomm.com gupta@eng.ucsd.edu hadi@eng.ucsd.edu

Abstract*— Deep Convolutional Neural Networks (CNNs) perform billions of operations for classifying a single input. To reduce these computations, this paper offers a solution that leverages a combination of runtime information and the algorithmic structure of CNNs. Specifically, in numerous modern CNNs, the outputs of compute-heavy convolution operations are fed to activation units that output zero if their input is negative. By exploiting this unique algorithmic property, we propose a predictive early activation technique, dubbed SnaPEA. This technique cuts the computation of convolution operations short if it determines that the output will be negative. SnaPEA can operate in two distinct modes, exact and predictive. In the exact mode, with no loss in classification accuracy, SnaPEA statically re-orders the weights based on their signs and periodically performs a single-bit sign check on the partial sum. Once the partial sum drops below zero, the rest of computations can simply be ignored, since the output value will be zero in any case. In the predictive mode, which trades the classification accuracy for larger savings, SnaPEA speculatively cuts the computation short even earlier than the exact mode. To control the accuracy, we develop a multi-variable optimization algorithm that thresholds the degree of speculation. As such, the proposed algorithm exposes a knob to gracefully navigate the trade-offs between the classification accuracy and computation reduction. Compared to a state-of-the-art CNN accelerator, SnaPEA in the exact mode, yields, on average, 28% speedup and 16% energy reduction in various modern CNNs without affecting their classification accuracy. With 3% loss in classification accuracy, on average, 67.8% of the convolutional layers can operate in the predictive mode. The average speedup and energy saving of these layers are 2.02× and 1.89×, respectively. The benefits grow to a maximum of 3.59× speedup and 3.14× energy reduction. Compared to static pruning approaches, which are complimentary to the dynamic approach of SnaPEA, our proposed technique offers up to 63% speedup and 49% energy reduction across the convolution layers with no loss in classification accuracy.

***Keywords*-Deep Neural Networks; DNN; Convolutional Neural Networks; CNN; Accelerators; Computation Reduction; Predictive Early Activation; Approximate Computing**

I. Introduction

Deep Convolutional Neural Networks (CNNs) are among the most widely used family of machine learning methods that have had a transformative effect on a wide range of applications. CNNs require ample amounts of computation even for a single input query. For instance, assigning a label to a relatively small RGB image (224×224×3) from the ImageNet database [1] requires billions of multiply-and-accumulate operations [2]–[4]. This paper aims to reduce these copious amount of computation by exploiting both their runtime information and algorithmic structure. In convolutional layers of many modern CNNs, each convolution operation is commonly followed by an activation function called a Rectifying Linear Unit (ReLU) that returns zero for negative inputs and yields the input itself for the positive ones. We observe that a large fraction of ReLU outputs are zero, indicating a large number of negative convolution outputs. Figure 1 illustrates this trend among several modern CNNs where ReLU nullifies 42%-68% of inputs. In addition, comparing the outputs of intermediate convolutional layers for different input images shows the zero values vary spatially across the images. Figure 2 illustrates this insight across two images passing through GoogLeNet [5]. The highlighted differences in the output of the intermediate convolutional layer attest to the varying spatial distribution of zeros. Harnessing these insights, we devise SnaPEA, a holistic software-hardware solution, that cuts a large fraction of the computations short by identifying the zero intermediate values earlier during the runtime.

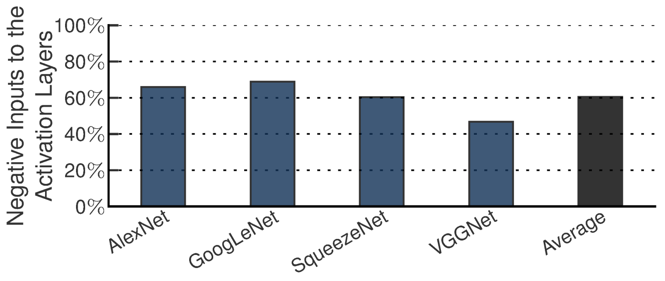


Figure 1: Fraction of activation input values that are negative.

SnaPEA operates in two distinct modes, namely exact and predictive. In the exact mode, in which the classification accuracy remains unchanged, SnaPEA detects the zero values by static re-ordering of weights along with a low-overhead sign-bit monitoring of partial sums. A negative partial sum triggers early termination of convolution operations. SnaPEA, in the predictive mode, trades off the classification accuracy for larger computation savings by predicting the zero values. Predictive mode results in earlier termination of the convolution operations compared to the exact mode, further reducing the amount of computation. Notwithstanding the higher benefits of predictive mode, an undisciplined prediction of zero values leads to significant loss compared to the nominal CNN classification accuracy. To minimize this loss while maximizing the reduction in computation, we propose a co-designed hardware-software solution that (1) statically pre-arranges the weights, (2) determines a threshold for triggering predictive early activation, and (3) uses a low-overhead runtime

[*]Vahideh Akhlaghi and Amir Yazdanbakhsh contributed equally.

SnaPEA: **Sna**ppy **P**redictive **E**arly **A**ctivation

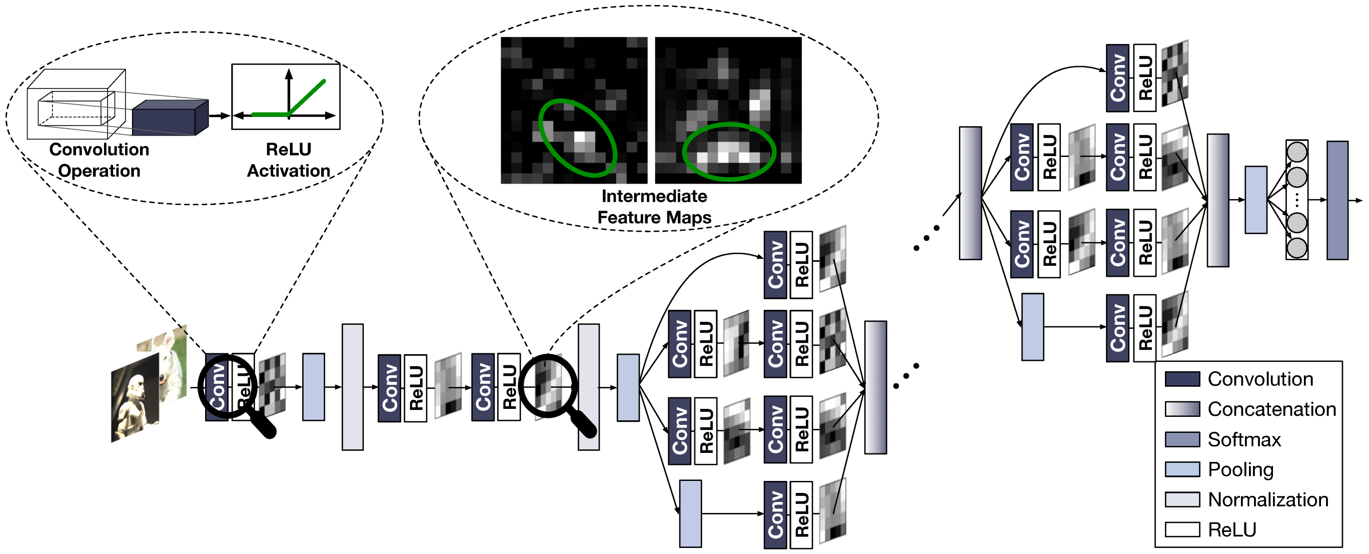


Figure 2: GoogLeNet [5], in which the intermediate feature maps for two input images are magnified. The ellipses on the intermediate feature maps highlight the varying spatial distribution of non-zero values for distinct input images.

monitoring mechanism to apply the early activation. As such, SnaPEA makes the following contributions:

1) **SnaPEA leverages the algorithmic structure of CNNs to to reduce their computation.** This work provides an insight that the amount of computation in CNNs can be significantly reduced by using a combination of runtime information along with the algorithmic structure of CNNs, which feeds many negative inputs to the activation function.
2) **SnaPEA is a runtime technique that cuts the CNN computations short.** Exploiting the aforementioned insight, this paper devises an exact runtime approach that relies on a single-bit sign-check to cut the computation short without losing any accuracy. In addition, SnaPEA comes with a predictive mode that speculates on the outcome of sign-check and terminates the computation even earlier, trading off accuracy for less computation.
3) **SnaPEA provides hardware-software solution to control the accuracy trade-offs.** We develop a multi-variable optimization algorithm that systematically thresholds the degree of speculation based on the sensitivity of the CNN output to each layer. The threshold becomes a knob for controlling the accuracy-computation tradeoff.

To evaluate the effectiveness of the proposed technique, we evaluate it on a number of modern CNNs. In the exact mode, which has no effect on the classification accuracy, SnaPEA, on average, delivers 28% (maximum of 74%) speedup and 16% (maximum of 51%) energy reduction over EYERISS [2], a state-of-the-art CNN accelerators. With 3% loss in classification accuracy, on average, 67.8% of the convolutional layers can operate in the predictive mode. The average speedup and energy saving of the layers in the predictive mode over EYERISS are 2.02× and 1.89×, respectively. GoogLeNet sees the maximum benefit of 3.59× speedup and 3.14× energy reduction. Finally, we evaluate the benefits of SnaPEA along with static pruning techniques using the already pruned SqueezeNet CNN [6]. In the exact mode, SqueezeNet achieves 30% speedup and 15% energy reductions with no loss of accuracy, demonstrating the complimentary nature of SnaPEA's dynamic approach to the static pruning techniques. Overall, these benefits suggests that coalescing runtime information with algorithmic insights can lead to new avenues for reducing the heavy computations of CNNs.

II. SnaPEA Hardware-Software Solution

SnaPEA provides a hardware-software solution to reduce the computation in a given CNN. The software part of SnaPEA, illustrated in Figure 3, is comprised of two distinct passes: one for the exact mode, and the other for the predictive mode. In the latter pass, the solution finds the thresholds for speculation while considering the acceptable loss in accuracy. In both cases, the task is to reorder weights of the convolution kernels, depending on the operating mode. To utilize these transformations, the SnaPEA comes with an accelerator design that can efficiently execute the CNN with reordered convolution weights with support for early termination of convolution. This section overviews the hardware and software components of SnaPEA.

A. *SnaPEA Software Workflow*

Figure 3 depicts the software workflow of SnaPEA which takes a CNN model, an acceptable accuracy loss, and an optimization dataset as its inputs. The CNN goes through the multiple passes of this workflow. The first pass, called Convolution Layer Extraction, elicits the convolution kernels of the CNN. Then, the weights of each kernel are re-ordered through the remaining passes, depending on the operating mode, exact or predictive.

Software workflow in the exact mode. To develop this flow, we leverage the observation that in the CNNs with ReLU activation layers, the inputs to the convolution layers are positive. Consequently, in these layers, the convolution output remains positive by performing Multiply-Accumulate (MAC) operations

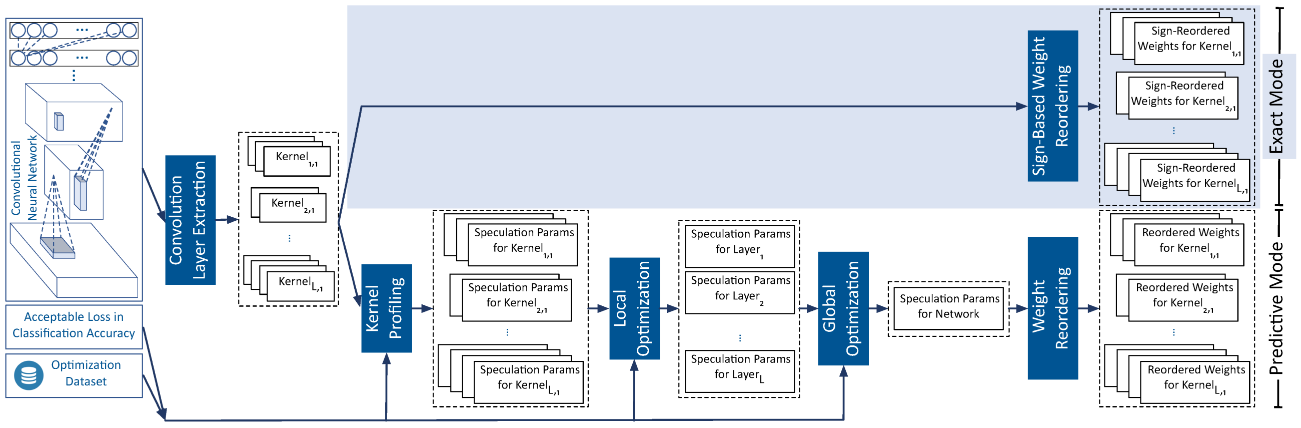

Convolutional Neural Network
Acceptable Loss in Classification Accuracy
Optimization Dataset
Convolution Layer Extraction
Kernel1,1
Kernel2,1
KernelL,1
Kernel Profiling
Speculation Params for Kernel1,1
Speculation Params for Kernel2,1
Speculation Params for KernelL,1
Local Optimization
Speculation Params for Layer1
Speculation Params for Layer2
Speculation Params for LayerL
Global Optimization
Speculation Params for Network
Weight Reordering
Sign-Based Weight Reordering
Sign-Reordered Weights for Kernel1,1
Sign-Reordered Weights for Kernel2,1
Sign-Reordered Weights for KernelL,1
Reordered Weights for Kernel1,1
Reordered Weights for Kernel2,1
Reordered Weights for KernelL,1
Exact Mode
Predictive Mode

Figure 3: Software workflow for SnaPEA.

with the positive subset of the weights. Only performing the remaining MAC operations with the negative subset of the weights can turn the convolution output negative. Given this insight, in the exact mode, Sign-Based Weight Reordering pass reorders the weights of convolution kernels based on their sign such that the positive subset are followed by the negative subset. The reordering enables SnaPEA to first perform MAC with the positive subset and then cut the computation and apply activation function earlier in the case of observing a negative partial output during the computation with negative weights.

Software workflow in the Predictive mode. To reduce the computations further, SnaPEA in the predictive mode, speculates on the sign of the convolution outputs before starting to go through the negative weights. A thresholding mechanisms controls the aggressiveness of the speculation. The intuition is that if the partial output of a convolution after a certain number of MAC operations is less than a threshold, the final convolution output will likely be negative. In this mode, since SnaPEA may misspeculate a positive convolution output as negative, the final classification accuracy may decline. Therefore, to utilize this intuition effectively, the software part of SnaPEA needs to deliberately determine: (1) a threshold value and (2) its associated number of MAC operations, such that the loss in the classification accuracy remains below the acceptable level while the computation reduction is maximized. These two speculation parameters need to be determined for as many layers as possible to maximize the benefits. To determine a proper set of parameters, SnaPEA formulates the problem as a multi-variable constrained optimization problem, and provides a greedy algorithm to solve it (See Section IV for more details). The algorithm is run by the software part on the Optimization Dataset through the following three passes. This triad of passes is to mange the complexity of accounting for the combined effects of the layers without an exponential explosion of the search space. First, the software *statically* runs a characterization pass, named Kernel Profiling, that measures the sensitivity of the accuracy to the imprecision introduced in each kernel in isolation. According to this sensitivity, the Kernel Profiling pass determines a set of speculation parameters for each kernel. Then, the next pass (Local Optimization) consolidates the kernel parameters of each layer and identifies a set of speculation parameters for the layer. This pass also considers the effects of speculation in each layer in isolation. Finally, the Global Optimization pass iteratively adjusts the speculation parameters of all layers such that the cross-layer effect yields an acceptable accuracy with the maximal computation reduction. The optimization algorithm runs *once* offline and does not impose additional runtime overhead during the execution of CNNs.

Based on the obtained speculation parameters for the entire network, the weights of each kernel are reordered by the Weight Reordering pass. This pass reorders the kernel weights by placing the ones determined by the speculation parameters ahead of the others. Then, the remaining weights are reordered based on the same procedure used for the Sign-Based Weight Reordering pass, which puts the negative weights after the positive ones. Finally, these reordered weights determine the execution of the CNN on the SnaPEA hardware.

B. *SnaPEA Hardware Architecture*

The SnaPEA architecture comprises number of identical Processing Engines (PEs), each of which is designed to compute a convolution using the reordered weights. To support computation with the reorderings, each PE is equipped with an index buffer that hold the indices of weights in the original kernel. The PE uses this index buffer to fetch the corresponding input value for each weight. This design is necessary because SnaPEA can reorder the weights but cannot tamper with the order of the inputs or activation. Section V expounds this design. The following provides an overview of the execution flow of a single convolution window in the exact and predictive modes.

Convolution execution flow in the exact mode. The PE first performs the operations of the positive weights. For the negative weights, the PE probes the sign of each partial sum value before proceeding to the next MAC operation. As soon as the partial sum becomes negative, the PE terminates the convolution early and triggers the early activation. Once the early activation is triggered, the PE is free to perform the computations of another convolution window. The sign-bit check merely requires a single AND gate, a low overhead addition to the PE.

Convolution execution flow in the predictive mode. In the predictive mode, each PE speculates the sign of the convolution output by comparing the partial sums with a threshold value

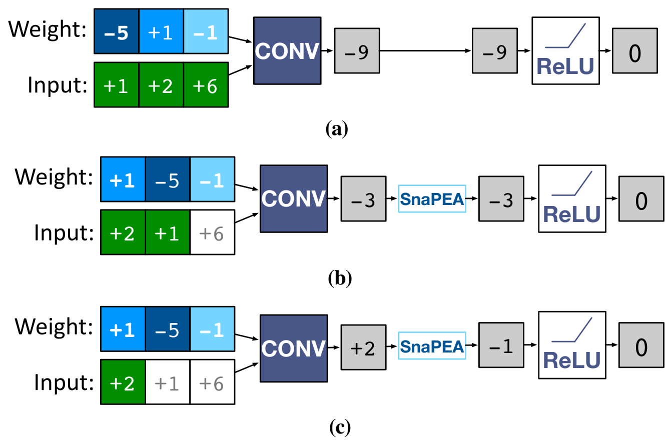


Figure 4: A 1×3 convolution in (a) unaltered (b) exact, and (c) predictive modes. In the latter two, the weights and their corresponding inputs are reordered. The white boxes highlight the operations that are cut.

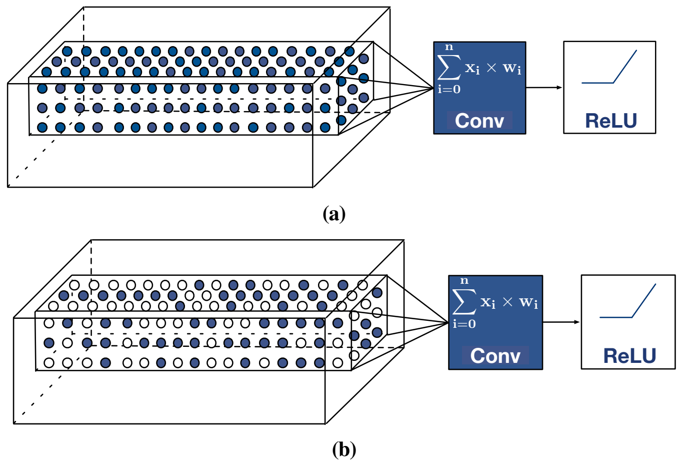

$\sum_{i=0}^{n} x_i \times w_i$
Conv
ReLU
(a)
$\sum_{i=0}^{n} x_i \times w_i$
Conv
ReLU
(b)

Figure 5: (a) The unaltered 3D convolution where all the MAC operations (bubbles) are carried out. (b) The same convolution with SnaPEA, where a significant number of operations are eliminated, delineated by the white bubbles.

after performing a pre-determined number of MAC operations. As mentioned, both the threshold and the number of operation are determined in the SnaPEA software workflow. If the partial result is less that the threshold, PE can speculatively terminate the convolution and compute the activation early. That is, the PE outputs a zero for the current convolution window. To support this speculative execution, each PE is equipped with a unit called Predictive Activation Unit (PAU) (See Section V).

III. Computation Reduction in SnaPEA

Figure 4 demonstrates how SnaPEA reduces the computation by an example of 1×3 convolution. Figure 4a performs the unaltered convolution in which all of the MAC operations are performed and yields "-9" as the output. Figure 4b illustrates convolution in the exact mode. In this mode, SnaPEA reorders the weights based on their sign, and starts the computation with the positive weights. The computation is terminated after performing only two MAC operations as the results is already negative, "-3". The simple sign check stops the computation. Although the partial sum after two MAC operations ("-3") has not reached the final convolution output ("-9"), it will be converted to zero by the following ReLU operation. As such, the results is the same as the unaltered convolution. Therefore, the exact SnaPEA does not change the final output after ReLU and does not lead to accuracy degradation.

Figure 4c illustrates how predictive mode cuts the operations earlier than the exact mode. As shown, after performing the MAC operations on only one weight, SnaPEA predicts that the convolution value will eventually be negative. Even though the corresponding partial sum value is positive ("+2"), SnaPEA speculatively triggers the ReLU function early with a negative value (e.g., "-1") and puts out zero. This speculation reduces the computation from two in the exact mode to one. In real-world CNNs, convolution is most often 3D and requires a relatively large number of MAC operations as depicted in Figure 5a. Using these methods, SnaPEA can forgo a significant number of the MAC operations as illustrated in 5b.

IV. SnaPEA Software Optimization

Significant computation reduction provided by the predictive mode comes at a price of experiencing loss in the classification accuracy due to misspeculating positive outputs as negative ones. To avoid unacceptable loss while maximizing the computation reduction, the predictive pass in the software part of SnaPEA, aims to systematically control the degree of speculation by properly determining the speculation parameters. To determine the parameters, the predictive pass formulates the problem as a constrained optimization problem, and designs a greedy algorithm to solve it. In this section, we first elaborate on the speculation parameters, and then explain the problem formulation and the algorithm to determine the parameters.

A. *Speculation Parameters*

As mentioned in Section II-A, speculation on the sign of a convolution output is performed by comparing the partial result of a set of MAC operations with a threshold value. Therefore, the threshold value and its associated set of operations are the parameters that control the degree of speculation. The threshold is merely a value that is required to be determined by the software for the controlled speculation. However, to determine a proper set of operations, the software requires to select the proper weights. One approach to select the weights would be to sort the weights in descending order of their absolute values, and select those with larger magnitude as a set of operations for performing the speculation. In this approach, although the contributions of both positive and negative weights are taken into account, the classification accuracy drastically declines. The reason is that selecting the weights with the larger magnitude ignores the contributions of input values which are, to a large degree, random and data dependent.

To mitigate the mentioned issue, SnaPEA sorts the weights in ascending order, partitions them into a number of smaller groups, and selects the weight with the largest magnitude from each group. This approach enables even the smallest weights to appear in the set of operations for the speculation; consequently, the smaller weights that may couple with large input values have an opportunity to contribute to the speculation. In this approach, to select a proper set of operations, the software only requires to determine the number of groups. This means that the number of groups can be exploited as an indicator of a set of operations in the speculation parameters. Accordingly, we denote the speculation parameters of all kernels in all layers of a CNN as (Th,N), in which Th is a list of threshold values and N is a list of the number of groups for selecting the corresponding operations.

B. Problem Formulation

The problem of finding the speculation parameters (i.e., (Th,N)) to maximize the computation reduction with an acceptable loss can be formulated as an optimization problem. In order to formulate the problem, we measure the computation reduction by subtracting the number of MAC operations that are performed by SnaPEA from the one performed by an unaltered CNN. However, since the number of MAC operations in the unaltered CNN is constant across various inputs, maximizing the computation reduction becomes equivalent to minimizing the number of MAC operations performed by SnaPEA. Accordingly, we define a function that calculates the number of MAC operations in SnaPEA as follows.

Let $o^d_{l,k}$ be the result of a single convolution window obtained by kernel k in layer l with the speculation parameters Th^k_l and N^k_l for the input image d. The number of MAC operations to compute $o^d_{l,k}$ can be calculated by the function Op shown in (1). Let assume that the reordered weights are stored in a 1D array such that the N^k_l speculation weights are placed at the beginning of the array while the remaining positive weights followed by the remaining negative weights are placed at the end. The function in (1) returns N^k_l if the value of partial sum after performing N^k_l operations (i.e., $\mathrm{PartialSum}_{\mathrm{N}^k_l}$) is less than the threshold value Th^k_l. Otherwise, the number of operations is determined by checking the sign of the partial sum value obtained by performing operations with the negative weights (i.e., $\mathrm{PartialSum}_{w-}$). If a negative partial sum is observed, the function returns the index of the corresponding negative weight in the array (i.e., Idx_{w-}). If none of the above cases occurs (last part in 1), the number of operations is set to the total number of weights in the kernel. Total number of weights of the kernel is $\mathrm{C}_{in,l} \times \mathrm{D}^k_l \times \mathrm{D}^k_l$, in which $\mathrm{C}_{in,l}$ is the number of input channels of the layer l, and D^k_l is the kernel width.

$$\mathrm{Op}(o^d_{l,k},\mathrm{Th}^k_l,\mathrm{N}^k_l) = \begin{cases} \mathrm{N}^k_l, & \text{if } \mathrm{PartialSum}_{\mathrm{N}^k_l} \leq \mathrm{Th}^k_l, \\ \mathrm{Idx}_{w-}, & \text{if } \mathrm{PartialSum}_{\mathrm{N}^k_l} > \mathrm{Th}^k_l \, and \; \mathrm{PartialSum}_{w-} \leq 0, \\ \mathrm{C}_{in,l} \times \mathrm{D}^k_l \times \mathrm{D}^k_l, & \text{otherwise} \end{cases} \tag{1}$$

The amount of computation to produce all the convolution outputs is the sum of the number of MAC operations required to produce each individual output. Based on this definition, the problem is translated into finding the speculation parameters that minimize total number of MAC operations and meet the constraint on the accuracy loss, which can be formulated as the following constrained optimization problem.

Let L be a set of all the layers in a given CNN, K_l a set of all the kernels in layer l, $\mathscr{D}$ an optimization dataset, ε an acceptable accuracy loss, Th^k_l and N^k_l the speculation parameters of kernel k of layer l, $\mathrm{O}^d_{l,k}$ the outputs of the convolution generated by kernel k in layer l for the input image d from $\mathscr{D}$, and $\mathrm{Accuracy}_{CNN}$ and $\mathrm{Accuracy}_{SnaPEA}$ the classification accuracy of the CNN and the classification accuracy obtained by SnaPEA, respectively. Now, (Th,N) can be determined by solving the following problem:

Algorithm 1 Finding the threshold value and its associated number of operations for all kernels in a CNN

```
Inputs:   CNN: a CNN model, $\mathscr{D}$: an optimization dataset,
          $\varepsilon$: Acceptable loss in classification accuracy
Outputs:  ParamCNN:
          Speculation parameters (Th,N) for the CNN
// Analyze each kernel individually
function KernelProfilingPass(CNN,$\mathscr{D}$,$\varepsilon$)
    Initialize ParamK[l][k]$\rightarrow \emptyset$
    for $\forall$ layer $l$ in CNN do
        for $\forall$ kernel $k$ in layer $l$ do
            for a set of values (th,n) do
                op, err = Simulate(CNN, $\mathscr{D}$, k, th, n)
                if err$\leq \varepsilon$ then
                    ParamK[l][k].append((th,n,op))
            Sort ParamK[l][k] based on op
    return ParamK
// Local Optimizer to find a set of params for each layer individually
function LocalOptimizationPass(CNN,$\mathscr{D}$,$\varepsilon$,ParamK)
    for layer l in CNN do
        for t in range(0,T) do
            for k in layer l do
                param = ParamK[l][k][t]
            op, err = Simulate(CNN,$\mathscr{D}$,$\varepsilon$,param)
            if err $\leq \varepsilon$ then
                ParamL[l].append((param,op,err))
    return ParamL
// Parameter tuning to accommodate for cross-kernel effect
function AdjustParam(CNN,ParamCNN,ParamL)
    for $\forall$ layer $l$ in CNN do
        for $\forall$ t in range(len(ParamL[$l$])) do
            $\mathrm{meritL}[l][t] = \frac{\text{-(ParamL}[l]\text{[2]-ParamCNN}[l]\text{[2])}}{\text{(ParamL}[l]\text{[1]} - \text{ParamCNN}[l]\text{[1])}}$
    l,t = Argmax(meritL)
    return (l,t)
// Global Optimizer to find the parameters for the entire network
function GlobalOptimizationPass(CNN,$\mathscr{D}$,$\varepsilon$,ParamL)
    for $\forall$ layer $l$  in CNN do ParamCNN[l] = ParamL[l][0]
    err = Simulate(CNN,$\mathscr{D}$,ParamCNN)
    while err$>\varepsilon$ do
        l,t=AdjustParam(CNN,ParamCNN,ParamL)
        ParamCNN[l] = ParamL[l][t]
        remove ParamL[$l$][$t$] from ParamL[$l$]
        err = Simulate(CNN,$\mathscr{D}$,$\varepsilon$,ParamCNN)
    return ParamCNN
Initialize ParamCNN[l]$\rightarrow \emptyset$
ParamK$=$ KernelProfilingPass(CNN,$\mathscr{D}$,$\varepsilon$)
ParamL$=$ LocalOptimizationPass(CNN,$\mathscr{D}$,$\varepsilon$,ParamK)
ParamCNN$=$GlobalOptimizationPass(CNN,$\mathscr{D}$,$\varepsilon$,ParamL)
```

$$\min_{\mathrm{Th,N}} \sum_{d\in\mathscr{D}} \sum_{l\in L} \sum_{k\in K_l} \sum_{o\in O^d_{l,k}} \mathrm{Op}(o,\mathrm{Th}^k_l,\mathrm{N}^k_l) \tag{2}$$
$$\text{subject to} \quad \mathrm{Accuracy}_{CNN} - \mathrm{Accuracy}_{SnaPEA} \leq \varepsilon$$

C. Finding the Speculation Parameters

In order to solve the optimization problem formulated as (2), we devise a greedy algorithm (i.e., Algorithm 1). The algorithm takes a CNN, an optimization dataset $\mathscr{D}$, and an acceptable accuracy loss ε and returns a list named ParamCNN that stores the value of the speculation parameters (Th,N). The algorithm first characterizes the sensitivity of the CNN to the speculation performed in each kernel in isolation. Then, it adjusts the speculation parameters for all the kernels through a greedy search such that they cooperatively minimize the computation while keeping the loss less than ε. Accordingly, we break the algorithm into two main stages (i.e., the profiling and the optimization stage) as follows:

Profiling stage. Function KernelProfilingPass in Algorithm 1 profiles the number of operations (op) and the accuracy loss (err) corresponding to various values of $(\mathrm{Th}_l^k,\mathrm{N}_l^k)$ for the kernel k in layer l. The exact mode of each kernel is also included in the profiling results by setting (0,1) as one of the values for its (th,n). The process is repeated for all the kernels in the CNN. The acceptable profiling results in terms of the accuracy loss, are accumulated in a list called ParamK. Each sub-list ParamK[l][k] in the list ParamK is sorted in ascending order based on the value of op.

Optimization stage. The optimization stage evaluates the combined effects of kernels and determines the proper speculation parameters for them. To avoid the complexity of evaluating the combined effects, the optimization stage consists of two functions: LocalOptimizationPass and GlobalOptimizationPass. The function LocalOptimizationPass in Algorithm (1), aims to evaluate the combined effects of kernels in each layer when the speculation is performed in the layer in isolation. Then, the function identifies a set of speculation parameters for each individual layer separately that leads to acceptable accuracy with minimum operations. To do this, the function LocalOptimizationPass generates T configurations for layer l such that in the t-th configuration, the speculation parameters of kernel k is set to t-th profiled parameters from the sorted list ParamK[l][k]. The configurations yielding an acceptable accuracy are selected as the set of configurations for the layer l. The acceptable configurations of all layers are populated in a list called ParamL, and passed to the next function.

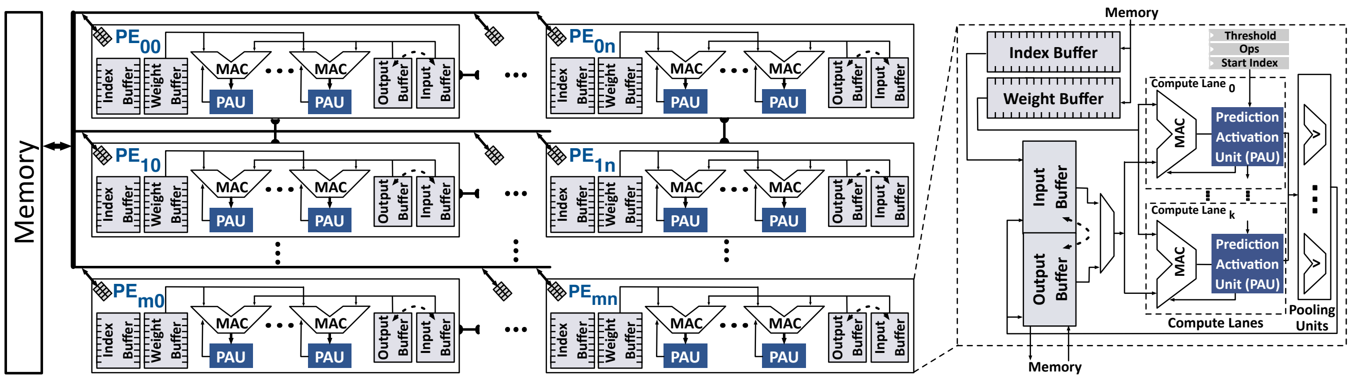
The second function, GlobalOptimizationPass, evaluates the effect of speculation performed in all the layers simultaneously and adjusts their speculation parameters with respect to the cross-layer effect on the classification accuracy and computation reduction. The output of the function is the final speculation parameters for all the kernels in the CNN which is stored in the list ParamCNN. To find the final parameters, the function first initializes the ParamCNN by setting the speculation parameters of each layer l to ParamL[l][0]. This initialization leads to the maximum computation reduction given the configurations stored in ParamL. However, the accuracy loss obtained by the initial setting may not be acceptable. In case of meeting the desired accuracy, the current parameters in ParamCNN is returned. Otherwise, the parameters are adjusted iteratively until the accuracy loss becomes less than ε. For adjusting the parameters, in the next iteration, those parameters are of interest that lead to small increase in the number of operations while large improvement in the classification accuracy. Hence, we define a merit value as $-\Delta_{err}/\Delta_{op}$, where the larger the Δ_{err} and the smaller the Δ_{op} are, the larger the merit is. Accordingly, the function GlobalOptimizationPass selects the configuration with the maximum merit value among all the configuration in ParamL and updates the corresponding speculation parameters in the list ParamCNN.

V. Architecture Design for SnaPEA

SnaPEA provides an accelerator architecture in order to efficiently execute the CNN with the transformed convolution operations. Modern CNNs consist of several back-to-back layers including convolution, ReLU activation, pooling, and fully-connected. To provide an end-to-end solution, the accelerator architecture consists of several units to execute the computation of all layers in the CNN. In order to efficient execution of CNNs, the architecture, specifically, targets to optimize the hardware of the convolution layers because of the following reasons. The first reason is that the computation of the convolution layers dominates the overall runtime of modern CNNs [2], [3], [7]–[10]. The second reason is to execute the convolutions with the reordered weights and to support the predictive early activation at the hardware level. To perform the computations of the fully-connected layers, the same hardware unit designed for the convolution layers is employed. The fully-connected layers are mainly used to perform the actual classification. CNNs usually have much smaller number (i.e. one or two) of fully-connected layers compared to the convolution layers at the final stage of the network. For example, GoogleNet has 57 convolution layers and only *one* fully-connected layer. On average, the computation of fully-connected layers accounts for $\approx$1% of the total number of computations performed in CNNs [2], [3], [8]. Therefore, using the same hardware unit for the fully-connected layers has virtually no impact on the total runtime of the CNNs. Finally, the SnaPEA architecture consists of dedicated units to support the computations of ReLU activation and pooling layers as well.

Figure 6 (a) illustrates the high-level block diagram of the proposed accelerator architecture. The accelerator consists of a 2D array of identical Processing Engines (PEs). Each PE is equipped with an input and output buffer that communicates with the off-chip memory. The weights of kernels and the inputs—coming from an off-chip memory—are stored in the dedicated buffers within each PE. In the following, we explain each unit of the accelerator architecture in more details.

Processing Engine (PE). Figure 6 (b) depicts the microarchitecture of one PE in the SnaPEA architecture. Each PE comprises multiple compute lanes, a weight and index buffer, an input/output buffer, and multiple Predictive Activation Units. Each compute lane consists of one dedicated Multiply-and-Accumulate (MAC) unit and one Predictive Activation Unit (PAU). The weight, index, and input/output buffers are shared across all the compute lanes within each PE. The computation of a convolution layer in each PE starts upon receiving a block of input features, their corresponding weights, and the weight indices from the off-chip memory. In every cycle, the PE controller reads one weight value from the weight buffer and broadcasts it to all the compute (MAC units) lanes. The PE controller also reads one weight index from the index buffer and sends the fetched index to the input buffer. Upon receiving the index, the input buffer reads a set of values (one value per



(a) Block Diagram of SnaPEA Architecture (b) PE Microarchitecture

Figure 6: (a) The overall structure of the SnaPEA architecture and its multilevel memory hierarchy, containing an off-chip memory and a distributed on-chip buffer for input and outputs. (b) The microarchitecture of each PE. The weights are shared across the compute lanes.

each MAC unit) and sends them to the MAC unit for processing. Each compute lane is dedicated to perform all the computations of *one* convolution window. That is, each MAC unit performs the multiplication of one input and weight for each convolution window and sends the results to the accumulation register. The accumulation register accumulates the partial sums for each convolution window. At the same time, the Predictive Activation Unit (PAU) checks the values of the partial sums to determine whether further computations for each convolution window is required. If the PAU determines that no further computations for a convolution window is required, it data gates the corresponding multiplier and accumulator to save energy. This process continues until either all the computations for the current convolution window are performed or the PAU determines to apply the activation early.

Weight and index buffers. The weight buffer contains the weight values of the convolution kernels in the pre-determined order (See Section IV). The weights are ordered offline and loaded into the memory with the proper ordering. Since the ordering of the weights are changed, we also need to add an index buffer to properly index the input buffer. This index is used to load a value from the index buffer. In every cycle, the controller fetches one weight from the weight buffer and broadcasts it to all the compute lanes. Simultaneously, the controller reads an index and sends it to the input buffer to read the corresponding input value. The input buffer delivers the inputs to each compute lane to perform one multiplication for adjacent convolution windows.

Input/Output Buffers. The input buffer holds a portion of input data for each convolution layer. Upon completion of all the computations, the results are written into the output buffer. We use one physical buffer for inputs and outputs. However, the buffer is logically divided into two sub-buffers for holding the input and output data of each layer. The logical partitioning allows us to use each of the sub-buffers as an input or an output buffer. The results of a layer l stored in the output buffer may be used by the next layer $l+1$ in . In this case, the data of each sub-buffers are logically swapped without wasting additional cycles for data transfers.

Predictive Activation Unit (PAU). Figure 7 illustrates the microarchitecture of the Predictive Activation Unit (PAU). One PAU unit is added to each compute lane to support the convolution operations in the exact and predictive mode. Performing the convolution operations in the exact mode only requires to check the sign of the partial sum value during the MAC operations with the negative weights. Accordingly, in the exact mode, the signal Predict is set to zero which allows the sign-bit of the partial sum stored in the register Acc Reg to determine the termination of the convolution operations. Once the sign-bit becomes one, the signal terminate is asserted and notifies the controller to terminate the rest of computations for the underlying convolution window.

In the predictive mode, the sign of the convolution output is speculated through the threshold value (th) and its associated number of operations (n) which are statically determined through the software part (See Algorithm 1). To perform speculation, PAU first checks the partial sum value, coming from the accumulator register, with a threshold value after a pre-determined number of MAC operations. At this time, the controller sets the signal Predict to one. If the partial sum value is less than the pre-determined threshold value, PAU predicts that the final value of this convolution window will eventually become negative. In this case, the PAU performs the following tasks: (1) notifies the controller that no further computations are required for this convolution window and (2) performs the early ReLU activation and sends zero to the output buffer. If the partial sum value is larger than the pre-determined threshold, the compute lane continues the computations for the convolution window normally until it reaches the negative weights. The next check on the partial sum starts upon starting the MAC operations with the negative weights. Here, the signal Predict is de-asserted, and PAU periodically performs a simple one-bit sign check on the partial sum values after each MAC operations, similar to the process mentioned in the exact mode. Once the sign-bit becomes one, the PAU terminates the convolution operations of the current window and sends a zero value to the output.

The mechanism of dynamically checking the partial sum values might lead to idle computation lanes. These computation lanes remain idle until the rest of the lanes finish the computations of their assigned convolution window. Accordingly, increasing the computation lanes may result in making more lanes idle despite providing higher parallelism between the convolution windows. In Section VI, we evaluate the effect of increasing computation

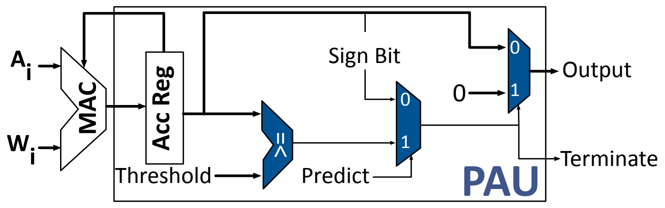


Figure 7: Prediction Activation Unit (PAU). The Predict signal determines the PAU operation mode (exact or predictive). The Terminate signal, once asserted, terminates the computation early.

lanes on the idle cycles and their effects on the performance and energy savings.

Pooling unit. Once the computations of a group of convolution windows complete, the PE performs the pooling operation on the results. Once done, the PE writes the results back into the output buffer. These results are either used in the computations of the next layers of CNNs or written back to the off-chip memory, if no further computations is required.

Organization of PEs. As shown in Figure 12, the SnaPEA architecture contains multiple identical PEs organized in a 2D array. The PEs are logically grouped both *vertically* and *horizontally*. The input data are partitioned between the horizontal PEs and the kernels are partitioned between the vertical PEs. The PEs in the same horizontal and vertical groups work on the same portion of the input data and kernels, respectively. Before the computation starts, a portion of input data are broadcasted to all the PEs within the same horizontal group. Similarly, one or more kernels are broadcasted to the PEs within the same vertical group. After the input and kernel data distribution, the PEs start and proceed their computations independent from other PEs. Once the computations for all the PEs within the same horizontal group end, the on-chip buffer delivers the next portion of input data. In this partitioning, some of the PEs may finish their computations earlier than other PEs within the same horizontal group. These PEs remain idle until all the other PEs complete their computations for all the assigned kernels and input data portion. This synchronization mechanism reduces the cost of multiple data broadcasting among the PEs while having a small impact on the performance. We evaluate the impact of this synchronization mechanism in Section VI-B by analyzing the sensitivity of performance to the number of compute lanes per each PE.

VI. Evaluation

A. Methodology

Workloads. We use several popular medium to large scale dense CNN workloads. We also include SqueezeNet [6] that maintains AlexNet-level accuracy with 50× fewer parameters through a static pruning approach. The fewer parameters in SqueezeNet are attained using an iterative pruning and re-training of the convolution weights. Table I summarizes the evaluated networks and some of the most pertinent parameters such as model size, number of convolution layers (Conv.), number of fully-connected layers (FC), and the baseline classification accuracy. In all of the evaluations, we use ILSVRC-2012 [1] validation dataset.

System setup. We use Caffe v1.0 [11] to run the pre-trained networks on a GPU. We compile Caffe using NVCC v8.0.62 and

Table I: Workloads, their released year, model size, number of convolution (Conv.) and fully-connected (FC) layers, and baseline classification accuracy. The model size shows the size of weights in Megabytes.

Network	Year	Model Size (MB)	# of Layers		Classification Accuracy
			Conv.	FC	
AlexNet	2012	224	5	3	72.6%
GoogLeNet	2015	54	57	1	84.4%
SqueezeNet	2016	6	26	1	74.1%
VGGNet	2014	554	13	3	83.0%

Table II: SnaPEA and Eyeriss [2] design parameters and area breakdown.

		SnaPEA		Eyeriss	
		Size	Area (mm^2)	Size	Area (mm^2)
PE	# Compute Lanes / PE	4	0.012	1	0.003
	Partial Sum Register	N/A	0	48 B	0.002
	Input Register	N/A	0	24 B	0.001
	Weight Buffer	0.5 KB	0.014	0.5 KB	0.014
	Index Buffer	0.5 KB	0.007	N/A	0
	Input / Output RAM	20 KB	0.250	N/A	0
	Predictive Activation Units	4	0.008	N/A	0
Accl.	Number of PEs	64	18.62	256	4.94
	Global Buffer	N/A	0	1.25 MB	12.9
Total Area		18.6 mm^2		17.8 mm^2	

GCC v4.8.4 with maximum architecture-specific and compiler optimizations enabled. We configure Caffe to use Nvidia cuDNN v6.0, a highly tuned GPU-accelerated deep neural network library.

Training/testing datasets. To learn the threshold values and their associated set of operations for each kernel, we implement Algorithm 1 through updating the data of convolutional layers in Caffe v1.0. We uniformly sample a subset of images from each of the 1,000 classes in ImageNet [1] to obtain the training and testing datasets for the proposed algorithm. The uniform sampling among all the classes enables us to cover images from distinct classes during the training and testing phases of Algorithm 1.

Architecture design and synthesis. We implement the microarchitectural units of the proposed architecture including the controllers, PEs, predictive activation unit (PAU), and registers in Verilog. We use Synopsys Design Compiler (L-2016.03-SP5) and a TSMC 45-nm standard-cell library to synthesize the proposed architecture and obtain the area, delay, and energy numbers of the logic hardware units.

SnaPEA and baseline architecture configurations. In this paper, we explore an 8×8 array of PEs in SnaPEA, each with four compute lanes, with a total of 256 MAC units. However, the SnaPEA architecture can be scaled up to larger numbers of PEs. Table II lists the major architectural parameters of the SnaPEA design. We add a weight buffer and an index buffer, each 0.5 KB per each PE. Both weight and index buffers are shared across all the compute lanes within each PE. Each PE is also equipped with a 20 KB buffer, that is evenly divided between input and output. The total capacity of the buffers therefore is 1.25 MB. Similar to the weight and index buffers, both input and output buffers are shared across all the compute lanes within a PE. Sharing the on-chip memories across multiple PEs enables us to reduce the overhead of index buffers. We size the input and output buffer so that the

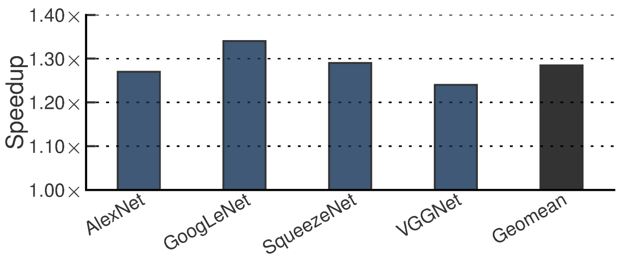
Table III: Absolute and relative energy comparison for different components of SnaPEA architecture along with off-chip memory access energy cost. PE energy includes the cost of Predictive Activation Unit (PAU).

Operation	Energy (pJ/Bit)	Relative Cost
Register File Access	0.20	1.0
16-bit Fixed Point PE	0.30	1.5
Inter-PE Communication	0.40	2.0
Global Buffer Access	1.20	6.0
DDR4 Memory Access	15.00	75.0

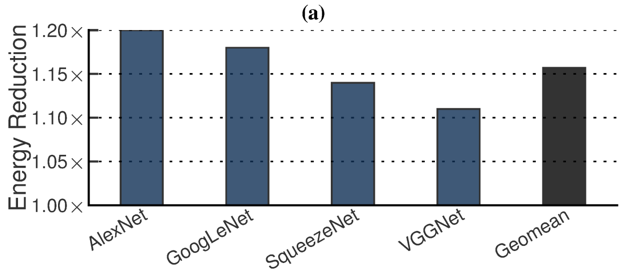
activations of all the CNN models, except VGGNet, fit within these on-chip buffer. This sizing eliminates the need of draining and filling the on-chip buffers during the execution. For VGGNet, which has deeper and larger layers, however, SnaPEA has to spill the activations to memory during the accelerations. We consider the overhead of spilling the data to the off-chip memory in our experiments. For the baseline architecture, we use the EYERISS [2] accelerator. Table II shows the major architectural components for EYERISS. To have the same peak throughput in both accelerators, we configure EYERISS to have the same number of MAC units (256) as ours. In addition, we allocate the same on-chip memory size (1.25 MB) to both accelerators. The frequency of both accelerators are fixed to 500 MHz. Table II summarizes the area of the major microarchitectural components in SnaPEA and EYERISS. Overall, the SnaPEA accelerator needs ≈4.5% more area compared to the EYERISS architecture with the specified configurations (Table II). This increase in the area is mainly attributed to the added predictive activation units (PAUs) in the PEs and the controllers.

Energy measurements. Table III lists the energy consumption of SnaPEA microarchitectural units. For hardware units, we use the synthesis results with TSMC 45-nm and reported numbers in TETRIS [8], which uses the same technology node and has a similar PE architecture as EYERISS. We include the energy overhead of the predictive activation unit in the energy cost of PE (second row in Table III). However, for the baseline architecture (EYERISS), we exclude the energy consumption of the predictive activation unit and use a relative cost of 1.0 in the evaluations. We use the publicly available Micron's DDR4 system power calculator [12] to estimate the energy cost of accesses to the off-chip memory.

Cycle-level microarchitecture simulation. We develop a cycle-level microarchitectural simulator that closely model the architecture of EYERISS and SnaPEA hardware to measure the performance and energy savings of both hardware. We integrate the microarchitectural components explained in Section V into the simulator in a cycle-level manner. To measure the energy savings, we use the synthesis results and the reported energy numbers from some of the recent works [2], [8], [13]. Furthermore, we use CACTI-P [14] to calculate the area and power of the register files and on-chip buffers. In the case of any inconsistency in terms of technology node, we properly scaled the area, delay, and energy numbers to make them consistent with our synthesis flow. We integrate the delay and energy numbers collected from the aforementioned sources into our cycle-level simulator. The simulator takes the configuration of a CNN architecture as input and generates an event log for each hardware component. Finally,



(a)



(b)

Figure 8: Overall (a) speedup and (b) energy reduction with exact mode.

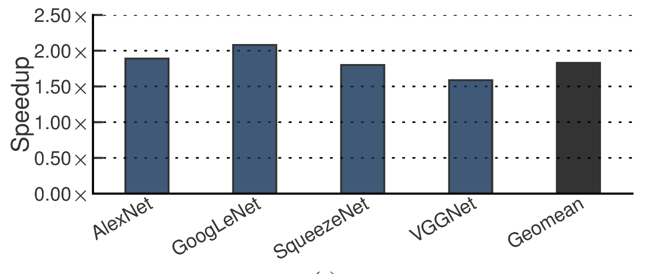
using the generated event log along the integrated delay and energy numbers, the simulator reports the number of cycles and energy numbers for the whole network.

B. Experimental Results

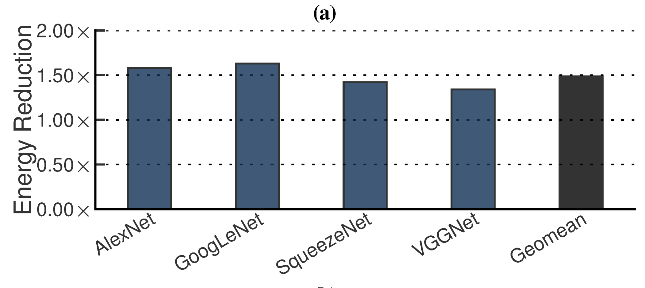
Overall benefits in the exact mode. Figure 8 illustrates the speedup and energy reductions when the predictive activation is disabled (i.e. exact mode). In this approach, SnaPEA hardware only applies the early activation when the value of partial sum drops below zero (See Section V). As there is no prediction, the CNN classification accuracy will *not* be deteriorated. In this setting, SnaPEA, on average, delivers 1.3× speedup and 1.16× energy reductions over EYERISS, respectively. Even for SqueezeNet [6]—a statically pruned convolutional neural network—SnaPEA yields 1.3× and 1.14×. These savings for SqueezeNet show that static pruning techniques are complimentary to the dynamic approach of SnaPEA. Overall, the results in the exact mode show the practicality of SnaPEA in delivering speedup and energy reductions even in the pure exact mode, in which the CNN classification accuracy remains untampered (Table I).

Overall benefits in predictive mode. Figure 9a illustrates the overall performance improvement of SnaPEA over EYERISS in the predictive mode while maintaining the classification accuracy within 3% range of its baseline value (See Table I). In this configuration, the predictive activation units (PAUs) might mis-predict a positive activation value as negative, hence degrading the classification accuracy. The injected error in the convolutional layers may lead to a drop in the final classification accuracy. The highest speedup (2.08×) is observed in GoogLeNet, in which a large fraction of the features are negative, and hence the saving is larger.

Figure 9b illustrates the energy reduction with SnaPEA in predictive mode over EYERISS [2]. Similar to the simulation settings for speedup, the degradation in classification accuracy is maintained within 3%. Among all the CNN models, GoogLeNet enjoys the highest energy reductions (1.63×). Also, in SqueezeNet [6], a statically pruned CNN model, our



(a)



(b)

Figure 9: Overall (a) speedup and (b) energy reduction with SnaPEA over EYERISS [2] in the predictive mode. The acceptable classification accuracy drop is maintained within ≤3% range of its baseline value.

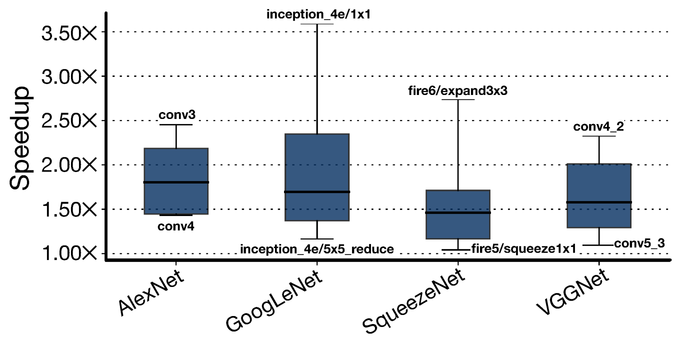


Figure 10: Speedup of convolutional layers in each network for the predictive mode when the degradation in classification accuracy is set to ≤ 3%.

technique yields 1.80× and 1.42× speedup and energy reductions, respectively. This result endorses the effectiveness of SnaPEA, even compared to static pruning techniques [6], in exploiting the runtime information to provide significant savings.

Figure 10 illustrates the speedup of convolutional layers in different networks when accuracy drop is set to 3%. The maximum range of speedup is observed in GoogLeNet, in which the maximum speedup is 3.59× achieved by convolution layer inception_4e/1x1, and the minimum speedup is 17% achieved by the layer inception_4e/5x5_reduce.

Moreover, in the predictive mode, to achieve acceptable accuracy drop, a fraction of the convolutional layers can operate in the predictive mode, which are specified by the software part. Table IV summarizes the percentage of convolutional layers that operate in the predictive mode in each network when the accuracy drop is set to 3%. The average speedup and energy saving across those layers are also brought in the table. The results show that, on average, 67.8% of the convolutional layers operate in the predictive mode, and the average speedup and energy saving across these layers are 2.02× and 1.89×, respectively.

Prediction accuracy. We study how effective the predictive mode is in predicting the negative values. Table V shows the average true negative and false negative rate across all the convolutional layers in the studied CNN models. The true negative rate measures the proportion of negative values that are correctly identified as negative. Applying early activation on these values does not have any effect on final classification accuracy. The false negative rate measures the proportion of the positive values that are mis-predicted as negative and squashed to zero; hence, *might* lead to degradation in the final classification accuracy. On average, the true (false) negative rate of our proposed prediction mechanism is 56.26% (20.41%). Due to our optimization technique (See Algorithm 1), on average, more than 86% of the error occurs on the small positive values. The small positive values in the activations generally have slight effect on the final classification accuracy. The main reason for this is attributed to the fact that each convolutional layer is commonly accompanied by a max-pooling layer, in which the small values are filtered out. The high true negative rate enables us to apply the activation on the negative values early and significantly reduce the ineffectual operations. Furthermore, the high true negative rate along the modest false negative rate exhibits the capability of SnaPEA in utilizing the runtime information to predict the negative values while meticulously injecting errors mainly on small positive values.

Table IV: The percentage of convolution layers that operates in the predictive mode, when classification accuracy drop is set to ≤ 3%. The second and third column illustrates the average speedup and energy reduction across these convolution layers.

Network	% of Convolution Layers	Average Speedup	Average Energy Reduction
AlexNet	60.0%	2.11×	1.97×
GoogLeNet	84.21%	2.17×	2.04×
SqueezeNet	65.38%	1.94×	1.84×
VGGNet	61.50%	1.87×	1.73×

Sensitivity to the degree of speculation. To study the effect of our proposed predictive early activation technique, Figure 11 illustrates the speedup with SnaPEA over EYERISS [2] when the classification accuracy loss varies from 0% to 3%. The 0% classification accuracy loss is when we do *not* use any prediction mechanism (exact mode). The remaining classification accuracy loss levels (e.g., 1.0%, 2.0%, 3.0%) is when we use the predictive early activation mechanism (predictive mode). In fact, supporting distinct levels of loss in the classification accuracy is one of the contributions of our work. The proposed predictive early activations technique exposes a knob for the user to gracefully navigate the trade-offs between CNN classification accuracy and performance and efficiency gains. On average, SnaPEA delivers 1.28×, 1.38×, 1.63×, and 1.9× speedup when we relax the constraint on the acceptable degradation of classification accuracy to 0.0%, 1.0%, 2.0%, and 3.0%, respectively. As we increase the acceptable degradation in the classification accuracy all the evaluated CNNs enjoy a boost in the speedup and energy reductions.

Sensitivity to the number of compute lanes. Figure 12 illustrates the impact of varying the number of compute lanes within each PE on speedup with SnaPEA over EYERISS. We present the results for the predictive mode when the maximum loss in the CNN classification accuracy is set to 3%. The second bar (Default)

Table V: True negative and false negative rate in predictive mode when classification accuracy drop is set to $\leq 3\%$.

Network	True Negative Rate	False Negative Rate
AlexNet	61.84%	21.39%
GoogLeNet	66.36%	28.37%
SqueezeNet	49.32%	16.69%
VGGNet	47.54%	15.21%

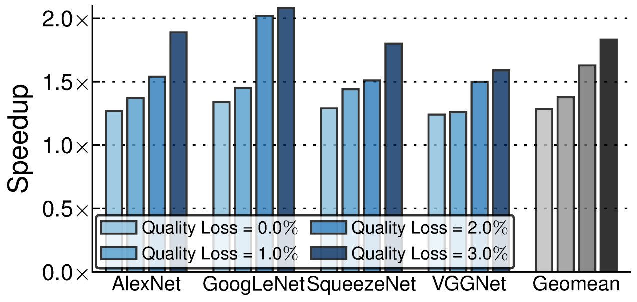


Figure 11: Speedup vs. loss in the CNN classification accuracy. Each bar indicates the speedup when the acceptable degradation in the classification accuracy is 0% (pure exact mode), 1% (predictive mode), 2.0% (predictive mode), and 3.0% (predictive mode), respectively.

shows the speedup in the baseline SnaPEA system (i.e., four compute lanes) over EYERISS with the same number of compute elements. The rest of the bars (first, third, and fourth bar) show the speedup of SnaPEA when the number of compute lanes per each PE is altered uniformly across all the PEs by a factor of half, two, and four, respectively. Increasing the number of compute lanes potentially increases the parallelization level between different convolutional windows. However, due to the synchronization overhead between the compute lanes per each PE (See Section V, Organization of PEs), the improvements diminish. The results show that increasing the number of lanes two times and four times hurts the performance by $\approx 36\%$ and $\approx 45\%$, respectively. Also, if we reduce the number of lanes by $0.5\times$, the performance decreases by $\approx 26\%$. The reason for this behavior is mostly because of an uneven amount of computations performed by each compute lane. In contrast to EYERISS [2], in SnaPEA the number of operations in each convolution window varies due to its runtime early activation. Therefore, increasing the number of arithmetic units reduces the utilization of the compute lanes and diminishes the benefit of higher parallelization.

VII. RELATED WORK

SnaPEA is fundamentally different from the prior studies in three major ways: (1) we exploit the inherent algorithmic structure of CNNs and runtime information to judiciously perform early activation and save ineffectual computations , (2) we expose a knob that enables the user to gracefully navigate the trade-offs between the classification accuracy, performance, and energy efficiency , and (3) we study the rich and unexplored area of task skipping in the domain of deep convolutional neural networks and conjoin these two disjoint lines of research in SnaPEA. Below, we discuss the most related works.

CNN accelerators. Several accelerators for convolutional neural networks has been proposed [2], [7]–[10], [15]–[23]. In some

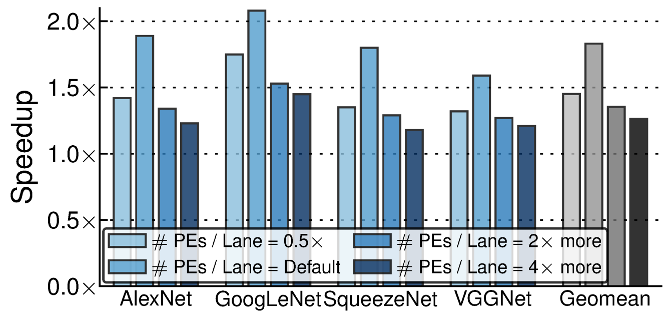


Figure 12: Sensitivity of speedup with SnaPEA over EYERISS to the number of compute lanes per each PEs. Each bar indicates the speedup when the number of compute lanes per each PEs is altered by different factors (acceptable classification accuracy drop $\leq 3\%$).

of the most recent works [2], [20], [23], 2D spatial architectures have been proposed to match with the convolution dataflow and maximize the data reuse. TETRIS [8] and Neurocube [15] have almost the same compute engines as the previous CNN accelerators. However, these works studied the challenges and opportunities for designing efficient CNN accelerators in a 3D-stacked memory setting. Neither of these accelerators evaluated the benefits of performing early activation in the convolution operation.

Pruning techniques. A handful of research [4], [6], [24]–[26] proposed various static pruning techniques to reduce the overhead of computation in deep convolutional neural networks. These static pruning techniques are agnostic to the dynamically-generated zeros whose locations in the activation layer vary from one image to another. As our results show, SnaPEA is complementary to these techniques and further improve the benefits over the static pruning techniques. Furthermore, several architectures also have been proposed [7], [9], [17]–[19] for exploiting the sparsity in the input activations and/or weights to improve the efficiency of the accelerator. In one of the most recent work, SCNN [7] designs an accelerator that exploits the sparsity in both the activations and weights. The proposed novel dataflow in SCNN maximizes the data reuse in the sparse activations and weights. This work is orthogonal to the previous efforts that focused on exploiting the sparsity in CNN accelerators. SnaPEA takes on a distinct approach than prior designs by judiciously re-ordering the MAC operations in a sliding window and performing the early activation in convolutional windows.

Task skipping. A handful of research efforts [27]–[34] have looked into task skipping in various domains. In one of the most recent efforts [29], Sidiroglou et al. proposed loop perforation in which the accuracy is traded in return for improvement in performance. In their proposal, they algorithmically transform the critical loops in the program and *only* execute a subset of their iterations. PredictiveNet [34] proposes a skipping mechanism for CNNs. They first perform the computations on the most-significant bits and then speculatively decide whether to perform the computation on the least-significant bits. However, SnaPEA completely skips the computations of the significant fraction of the operations. As such, SnaPEA not only reduces the computation cost, but also reduces the number of accesses to the on-chip buffers. Although SnaPEA takes inspiration from the prior proposals in task skipping, it uniquely applies the task skipping mechanism in the domain of deep convolutional neural networks in order to effectively eliminate the ineffectual data transfers and computations.

VIII. Conclusion

Traditionally, layers of deep neural networks have been thought to work in separation while handing each other their results. However, our work took a disparate approach in considering the most common sequence of layers in emerging deep networks to reduce the amount of computation. As such, SnaPEA has devised a predictive early activation that operates in two distinct modes, namely exact and predictive mode. In the exact mode, in which the nominal classification accuracy remains untampered, SnaPEA uses a combination of static re-ordering of the weights and low-overhead sign check to determine when to terminate the computation. SnaPEA further improves the performance and efficiency of convolution operations in the predictive mode by speculatively cutting the computation of convolution operations if it predicts its output is negative, immediately applying activation. Compared to a recent CNN accelerator, SnaPEA in the exact mode yields 28% speedup (maximum of 74%) and 16% (maximum of 51%) energy reductions across various modern CNNs without affecting their classification accuracy. With 3% loss in classification accuracy, on average, 67.8% of the convolutional layers operate in the predictive mode, and the average speedup and energy saving across these layers are 2.02× and 1.89×, respectively. The significant gains due to the computation and memory access reduction across several modern CNNs show the effectiveness of our approach that conjoins runtime information and algorithmic insights into a unified accelerator.

IX. Acknowledgments

We thank our shepherd, Philip Stanley-Marbell, and the member of Alternative Computing Technologies (ACT) Lab, Michael Brzozowski, Jake Sacks, Hardik Sharma, Divya Mahajan, and Jongse Park for insightful comments and discussions. Amir Yazdanbakhsh was partly supported by a Microsoft Research PhD Fellowship. This work was in part supported by NSF awards CNS#1703812, ECCS#1609823, CCF#1553192, CCF#1029783, Air Force Office of Scientific Research (AFOSR) Young Investigator Program (YIP) award #FA9550-17-1-0274, and gifts from Google, Microsoft, and Qualcomm.

References

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, 2015.

[2] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.

[3] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs," in *MICRO*, 2016.

[4] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding," in *ICLR*, 2016.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *CVPR*, 2015.

[6] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and¡0.5 MB Model Size," *arXiv preprint arXiv:1602.07360*, 2016.

[7] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *ISCA*, 2017.

[8] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

[9] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *ISCA*, 2016.

[10] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *ASPLOS*, 2014.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[12] "DDR4 Spec - Micron Technology, Inc." https://goo.gl/9Xo51F.

[13] S. Galal, *Energy Efficient Floating-Point Unit Design*. PhD thesis, The Department of Electrical Engineering of Stanford University, 2012.

[14] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques," in *ICCAD*, 2011.

[15] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.

[16] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing," in *MICRO*, 2016.

[17] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernndez-Lobato, G. Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *ISCA*, 2016.

[18] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An Accelerator for Sparse Neural Networks," in *MICRO*, 2016.

[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016.

[20] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *ISCA*, 2015.

[21] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: A Polyvalent Machine Learning Accelerator," in *ASPLOS*, 2015.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[23] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision," in *CVPR Workshops*, 2011.

[24] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the Regularity of Sparse Structure in Convolutional Neural Networks," *CoRR*, 2017.

[25] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary Neural Networks for Resource-efficient AI Applications," in *IJCNN*, 2017.

[26] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," *arXiv preprint arXiv:1707.06168*, 2017.

[27] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-Free Value Prediction with Safe to Approximate Loads," in *TACO*, 2015.

[28] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically Accurate Program Transformations," in *SAS*, 2011.

[29] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing Performance vs. Accuracy Trade-offs with Loop Perforation," in *FSE*, 2011.

[30] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of Service Profiling," in *ICSE*, 2010.

[31] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," Tech. Rep. MIT-CSAIL-TR-2009-042, MIT, 2009.

[32] M. C. Rinard, "Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points," in *OOPSLA*, 2007.

[33] M. Rinard, "Probabilistic Accuracy Bounds for Fault-tolerant Computations that Discard Tasks," in *ICS*, 2006.

[34] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "PredictiveNet: An Energy-efficient Convolutional Neural Network via Zero Prediction," in *ISCAS*, 2017.