UNIVERSITY OF CALIFORNIA SAN DIEGO

Algorithm-Hardware Optimization of Deep Neural Networks for Edge Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Vahideh Akhlaghi

2020

DEDICATION

To my dearest family and my loving husband

EPIGRAPH

Be the peace you wish to see in the world!

—Martin Luther King, Jr.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the encouragement, guidance and support from many people throughout my academic study.

First and foremost, I would like to express my greatest appreciation to my Ph.D. advisor, Professor Rajesh K. Gupta, for providing me an opportunity to join University of California San Diego and his research group, where I could follow a new path for research and enjoy the beautiful city of San Diego. His great leadership and vision enabled me to choose an exciting research area and explore it enthusiastically during my PhD program. His expertise, guidance and continuous support helped me shape my Ph.D. dissertation as well as my future path.

I am deeply grateful to my co-advisor, Professor Hadi Esmaeilzadeh, for sharing his knowledge in computer architecture and providing opportunities to explore practical aspects of my research area that significantly helped broaden my perspective. I would also like to thank Professor Sicun Gao, one of my committee members, whose expertise in optimized automation notably aided in the introduction of a new way for systematic hardware design. I would like to thank my other committee members, Professor Gert Cauwenberghs and Professor Ryan Kastner, for their feedback and suggestions, without which, I would not have expanded my research from different perspectives. Moreover, I am thankful to Professor Massimo Franceschetti and Professor Hao Su, who kindly accepted to collaborate and shared their valuable ideas and feedback.

In addition to professors at UCSD, I am greatly thankful to Professor Zainalabedin Navabi and Professor Ali Afzali-Kusha, who not only taught me research skills and fundamental concepts in digital systems while studying for my Bachelor's and Master's degrees at University of Tehran, their efforts to instill confidence in their students including me to think big and pursue their own ideas were also commendable.

I was highly fortunate to be surrounded by past and current members of the Microelectronic Embedded Systems Laboratory (MESL) at UCSD, whose guidance, friendship, sense of humor, and support made my Ph.D. program at UCSD enjoyable and unforgettable. I would like to thank my colleagues in the MESL, Abbas Rahimi, Atieh Lotfi, Xun Jiao, Jeng-Hau Lin, Dezhi Hong, Francesco Fraternali, Dhiman Sengupta, Sean Hamilton, Jason Koh, Omid Assare, Manish Gupta, Zhou Fang, Ranak Roy Chowdhury, Muhammad Adnan, Bharathan Balaj and many more who made me feel here home and my path brighter and easier.

Last but not least, I owe my deepest gratitude to my parents, Masoumeh Koosha and Seyfollah Akhlaghi, for their unconditional love, dedication, wisdom, their belief in me and more importantly for respecting my choice to study abroad. I would have not been able to stand at this point in my life without their help, love, patience and support. I am also greatly indebted to my sisters, Saeedeh Akhlaghi and Sepideh Akhlaghi, for always being there in the time of need with love and for encouraging me to move forward. I would also like to extend my deepest gratitude to my husband, Hamed Omidvar, who not only helped strengthen my dissertation through sharing his mathematical insights and fundamental ideas, he has also supported me during challenging times. His trust in me and his love for me have been helping me grow.

The material in this dissertation is based on the following publications.

Chapter 2 contains re-organized reprints of Vahideh Akhlaghi, Abbas Rahimi, and Rajesh Gupta, “Resistive Bloom Filters: from Approximate Membership to Approximate Computing with Bounded Errors”, *In IEEE Design, Automation, and Test in Europe (DATE)*, 2016, of which the dissertation author is the primary author, and Xun Jiao, Vahideh Akhlaghi, Yu Jiang, and Rajesh Gupta, “Energy-Efficient Neural Networks using Approximate Computation Reuse”, *In IEEE Design, Automation, and Test in Europe (DATE)*, 2018, of which the dissertation author is the co-author and primary investigator.

Chapter 3, in full, is a reprint of the material appeared in Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh Gupta, and Hadi Esmaeilzadeh, “Snapea: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks”, *In ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018. This dissertation author is the primary author of the paper.

Chapter 4, partly contains the materials of Vahideh Akhlaghi, Sicun Gao, and Rajesh Gupta, “LEMAX: learning-based Energy Consumption Minimization in Approximate Computing with Quality Guarantee”, *In ACM/IEEE Design Automation Conference (DAC)*, 2018. In addition, this chapter contains the unpublished materials co-authored by Vahideh Akhlaghi, Dezhi Hong, Sicun Gao, Hadi Esmaeilzadeh and Rajesh K. Gupta. This dissertation author is the primary author of the mentioned materials.

Chapter 5 contains the materials of Hamed Omidvar, Vahideh Akhlaghi, Hao Su, Massimo Francescheti, and Rajesh K. Gupta, “Associative Convolutional Layers”, *submitted for publication in International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2021, of which this dissertation author is the co-author and the primary investigator and Vahideh Akhlaghi, Hamed Omidvar, Massimo Francescheti, and Rajesh K. Gupta, “Parameter Approximation of CNNs for Improved Inference on FPGA”, *submitted for publication in Design Automation Conference (DAC)*, 2021, of which this dissertation author is the primary author.

Finally, I would like to thank all my co-authors and collaborators (Abbas Rahimi, Xun Jiao, Yu Jiang, Amir Yazdanbakhsh, Kambiz Samadi, Dezhi Hong, Hamed Omidvar, Massimo Franceschetti, Hao Su, Sicun Gao, Hadi Esmaeilzadeh and Rajesh K. Gupta) who kindly approved the inclusion of the aforementioned publications in my dissertation.

VITA

2007	B. S. in Computer Engineering (Hardware Engineering), University of Tehran
2011	M. S. in Computer Engineering (Computer Architecture), University of Tehran
2020	Ph. D. in Computer Science (Computer Engineering), University of California San Diego

PUBLICATIONS

Vahideh Akhlaghi, Hamed Omidvar, Massimo Franceschetti, and Rajesh K. Gupta, "Parameter Approximation of CNNs for Improved Inference on FPGA", *submitted for publication in Design Automation Conference (DAC)*, 2021.

Hamed Omidvar, Vahideh Akhlaghi, Hao Su, Massimo Franceschetti, and Rajesh K. Gupta, "Associative Convolutional Layers", *submitted for publication in International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2021.

Vahideh Akhlaghi, Sicun Gao, and Rajesh Gupta, "LEMAX: learning-based Energy Consumption Minimization in Approximate Computing with Quality Guarantee", *in Proceedings of ACM/IEEE Design Automation Conference (DAC)*, 2018.

Vahideh Akhlaghi*, Amir Yazdanbakhsh*, Kambiz Samadi, Rajesh Gupta, and Hadi Esmaeilzadeh, "Snapea: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks", *in Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018. (*equal contributions)

Xun Jiao, Vahideh Akhlaghi, Yu Jiang, and Rajesh Gupta, "Energy-Efficient Neural Networks using Approximate Computation Reuse", *in Proceedings of IEEE Design, Automation, and Test in Europe (DATE)*, 2018.

Vahideh Akhlaghi, Abbas Rahimi, and Rajesh Gupta, "Resistive Bloom Filters: from Approximate Membership to Approximate Computing with Bounded Errors", *in Proceedings of IEEE Design, Automation, and Test in Europe (DATE)*, 2016.

ABSTRACT OF THE DISSERTATION

Algorithm-Hardware Optimization of Deep Neural Networks for Edge Applications

by

Vahideh Akhlaghi

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2020

Professor Rajesh K. Gupta, Chair
Professor Hadi Esmaeilzadeh, Co-Chair

Deep Neural Network (DNN) models are now commonly used to automate and optimize complicated tasks in various fields. For improved performance, models increasingly use more processing layers and are frequently over-parameterized. Together these lead to tremendous increases in their compute and memory demands. While these demands can be met in large-scale and accelerated computing environments, they are simply out of reach for the embedded devices seen at the edge of a network and near edge devices such as smart phones and etc. Yet, the demand for moving these (recognition, decision) tasks to edge devices continues to grow for increased localized processing to meet privacy, real-time data processing and decision making needs. Thus, DNNs continue to move towards the edges of the networks at ‘edge’ or ‘near-edge’ devices, even though a limited off-chip storage and on-chip memory and logic on the edge devices prohibit the deployment and efficient computation of large yet highly-accurate models.

Existing solutions to alleviate such issues improve either the underlying algorithm of these models to reduce their size and computational complexity or the underlying computing architectures to provide efficient computing platforms for these algorithms. While these attempts improve computational efficiency of these models, significant reductions are only possible through optimization of both the algorithms and the hardware for DNNs.

In this dissertation, we focus on improving the computation cost of DNN models by taking into account the algorithmic optimization opportunities in the models along with hardware level optimization opportunities and limitations. The techniques proposed in this dissertation lie in two categories: optimal reduction of computation precision and optimal elimination of inessential computation and memory demands. Low precision but low-cost implementation of highly frequent computation through low-cost probabilistic data structures is one of the proposed techniques to reduce the computation cost of DNNs. To eliminate excessive computation that has no more than minimal impact on the accuracy of these models, we propose a software-hardware approach that detects and predicts the outputs of the costly layers with fewer operations. Further, through the design of a machine learning based optimization framework, it has been shown that optimal platform-aware precision reduction at both algorithmic and hardware levels minimizes the computation cost while achieving acceptable accuracy. Finally, inspired by parameter redundancy in over-parameterized models and the limitations of the hardware, reducing the number of parameters of the models through a linear approximation of the parameters from a lower dimensional space is the last approach proposed in this dissertation. We show how a collection of these measures improve deployment of sophisticated DNN models on edge devices.

Chapter 1

Introduction

Deep Neural Networks (DNNs) have expanded applications in various areas, including but not limited to healthcare, education, cybersecurity, and climatology [LLZ$^+$17, HRH$^+$19, AKX$^+$19, OYSO17, SM19, CZZ19, FMJS19, YBU19]. To continuously fuel performance improvements, the size and computation complexities of DNNs are still increasing rapidly. Yet, complex and powerful algorithms are placing high demands on the computing systems in terms of latency and energy consumption. Furthermore, as implementation of these algorithms on the edge and end devices is becoming commonplace due to extreme importance of data privacy and real-time data processing and decision making, similar concerns are also relevant in this area [MARAM18, LBG$^+$15, WBC$^+$19]. Therefore, to fully exploit the benefits of these algorithms and increase their applicability, they are required to become less computationally complex and be implemented efficiently in order to be executed with high speed and low energy consumption on both large and small scale devices while providing high accuracy.

There exist several attempts to improve DNN models execution ,the majority of which can be categorized in two major directions: lowering the computational complexities of these models algorithms and designing efficient hardware architecture. However, in this dissertation, we show that simultaneous consideration of opportunities in both models algorithms and hardware designs provides significant improvement and optimizes the state-of-the-art solutions without unacceptable drop in the models accuracies. This dissertation provides a set of algorithm-hardware optimization techniques to accelerate Convolutional Neural Networks (CNNs), as one of the important DNN models that are extensively used in various applications such as image and video recognition, recommender systems, and etc. These techniques can be applied on various DNN models due to their algorithmic similarity. Before explaining the techniques, in this section, we review CNN algorithms, their computation and memory demands, and the existing solutions to improve the computation costs of these algorithms.

1.1 Background and Problem Definition

CNN algorithms automatically extract features from a given input image through a set of layers, and depending on the application, the extracted features are being processed and used for decision making, classification, recognition, and etc. To extract the features, CNNs consist of various types of layers such as convolutional layers, batch normalization layers, activation layers, pooling layers and fully connected layers. Compared to other types of layers, the computation and memory demands of convolutional layers are relatively high (i.e., it accounts for more than 90% of total computation and memory demand of the whole model [CES16, GPY^{+}17, SPM^{+}16].)

A convolutional layer is composed of a number of filters that are convolved with the input of the layer called input feature maps with several dot-product operations and generate the output called output feature maps. Equation 1.1 shows the mathematical formulation of a singe pixel in the output feature maps Y generated by a a convolutional layer that has F filters and input feature maps of X. The parameters corresponding to filters (i.e., the weights) are shown as a matrix W. In this equation, s is the stride determining the overlap between the regions of the input called receptive fields that each convolution operation is performed on, K the size of kernels in filters, C the number of channels of the input feature maps, and D the number of rows (and columns) of the output feature maps.

$$Y[f,i,j] = \sum_{c=1}^{C} \sum_{k=1}^{K} \sum_{k=1}^{K} X[c][si+k][sj+k] \times W[f][c][k][k] \\ 0 \leq f < F, \quad 0 \leq i,j < D \tag{1.1}$$

Equation 1.1 indicates several points. First, the main operation in the convolutional layers is multiply-and-add (MAC) operations that is performed in a large quantity depending on values of F, D, C, K. As we can see in the equation, total number of operations to generate all the output feature maps in a single convolutional layer of a CNN with F filters is $F \times D^2 \times C \times K^2$, meaning that the current trend of increasing the number of layers and deepening the convolutional layers in CNNs to improve their accuracies results in performing copious number of MAC operations. Second, total number of parameters in a single convolutional layers is $F \times C \times K^2$, which similarly indicates that large number of convolutional layers with a large number of filters in current highly accurate CNN models results in tremendous number of parameters to be stored and accessed on computing devices. According to these points, the computation of today's CNN models mainly due to the large number of MAC operations and large sizes of model parameters pushes the limits of computing devices.

Table 1.1 summarizes various CNN models for the classification task on ImageNet-1K [RDS$^+$15] dataset with their corresponding number of parameters, the number of MAC operations that are required to classify a single input image and their Top-1 and Top-5 accuracies. The reported accuracies are obtained from [Tor30]. As we can see in the table, in CNN models with similar architectures increasing the number of parameters and MAC operations leads to increase in their classification accuracies. However, this is not the case across various model architectures. For example, VGG-19 with batch normalization layers (VGG-19-w/BN in the table), although consists of a considerably larger number of parameters and operations compared to ResNet-50, ResNet-101 and ResNeXt architectures, both of its Top-1 and Top-5 accuracies are lower than those models. In addition, comparing ResNeXt-50 (32×4d) and ResNet-50 indicates that only

Table 1.1: Various CNN models for ImageNet-1K dataset and their number of parameters, MAC operations for classifying a single input image, and their Top-1 and Top-5 classification accuracies. The reported accuracies are obtained from [Tor30] except those of ResNeXt-101(64×4d) marked by asterisks which are obtained from [XGD+17]. In networks with similar structures, increasing the number of parameters and operations will improve the accuracies.

CNN Models	#Parameters	#MAC Operations	Top-1 accuracy	Top-5 accuracy
VGG-19-w/BN [SZ14]	144M	19.6B	74.4	91.9
MobileNet-V2 [SHZ+18]	3.4M	300M	71.9%	90.3%
ResNet-18 [HZRS16a]	11.1M	1.8B	69.8%	89.1%
ResNet-50 [HZRS16a]	25.6M	3.8B	76.2%	92.9%
ResNet-101 [HZRS16a]	44.7M	7.6B	77.4%	93.6%
ResNeXt-50(32x4d) [XGD+17]	25.6M	3.8B	77.6%	93.7%
ResNeXt-101(64x4d) [XGD+17]	79.8M	15.2B	79.6%*	94.7%*

re-structuring the original model (ResNet-50) without any additional parameters and operations can result in improving the accuracy. Therefore, in general, Table 1.1 indicates that there are opportunities in the model algorithm itself that can be exploited to improve the model accuracy without the need to add extra parameters and operations.

In addition to opportunities in the model algorithms, another possible direction to improve the model execution cost (i.e., energy consumption and latency) is to optimize hardware design and architecture of underlying computing devices for these algorithms. Existing powerful computing devices with tremendous computing capacities such as Graphics Processing Units (GPUs), although are the main reason behind the advancement of highly accurate DNN models, due to several reasons such as high cost, large energy consumption and area, they are not suitable to be deployed in various range of devices especially in the edge and end devices. Therefore, there is an imperative need to design low energy yet powerful computing devices that can execute the CNN models efficiently without significant degradation of the model accuracy.

According to the mentioned needs and opportunities to improve the computational costs of CNN models without degrading their accuracies to the unacceptable levels, in this dissertation, we introduce a set of techniques, namely algorithm-hardware optimization techniques. These methods optimize the execution of the CNN models and advance the related state-of-the-art methods by considering the optimization opportunities available both in the model algorithms and their hardware implementations.

1.2 Related Works

To improve the computational costs of DNN algorithms, most of the existing techniques mainly focus on optimizing either the computation in DNN models or the underlying computing architectures. Here, we review some of these techniques in each category.

1.2.1 Algorithmic Optimization

To optimize the model computation and size, several techniques have been introduced that can be categorized into four groups [CWZZ18]: network pruning, quantization, low rank factorization, and transferred convolutions. Network pruning approaches reduce the network size and computation by removing the unnecessary connections and operations in the DNN models determined based on various metrics. In quantization based approaches, the computation precision is reduced by lowering the bit-width of operations, weights and feature maps. The methods in low-rank factorization category break down the convolutional layers into several smaller layers by factorizing their weight matrices to low-rank ones, which leads to fewer number of parameters and MAC operations. The last category of the methods to optimize DNN algorithms is transferred convolutions in which a fraction of parameters are eliminated and their corresponding feature maps are generated by a specific transformation of the outputs of the remaining parameters. In the rest of this section, several methods in each category will be explained in more details.

Network Pruning

One of the common techniques to reduce the computation and memory demands of a DNN model is pruning in which unimportant parameters and operations are identified and removed.

One set of approaches to identify unimportant parameters and operations is based on the magnitude of the weights that eliminates the neurons corresponding to the weights with small magnitude. An example of such methods is biased weight decay [HP89] which includes the weight decay term in the weight update process during the backpropagation in order to make the unimportant weights approach zero and be removed automatically. Another method in this category is introduced in [HPTD15b] that prunes the parameters and the related operations of the pre-trained models by removing their small magnitude weights.

Optimal Brain Damage (OBD) [CDS90] and Optimal Brain Surgeon (OBS) [HSW93] are another set of pruning approaches that eliminate the weights according to their impact on the loss function rather than their magnitude. These methods are built upon measuring the changes in the loss function as a result of deleting a weight and removing the weights with small impact on the loss. To measure the impact of weights on the loss function, the Taylor series of the change in the loss function is represented as a function of perturbation of the weights and a set of coefficients that can be obtained by the Hessian matrix at the end of training.

Despite the effectiveness of the mentioned approaches in reducing the model size and computation, due to irregular sparsity which leads to irregular computation patterns, their final impact on the execution costs is not optimal. To avoid this issue, finding structured sparse models in which a set of regular computational blocks in the original model (e.g., a number of filters in a convolutional layer of a CNN model) are removed are another technique in pruning DNN networks proposed in [WWW$^+$16]. The technique learns the structured sparse models by adding a group lasso of a set of weights into the loss function and using the regularization technique to make all the weights in a group zero.

Quantization

Lowering the numerical precision of computation through lowering the bit-width of parameters and feature maps, namely quantization, is another common effective approach to reduce the computation and memory demands of DNN models. In general, the computation in DNN models is performed in 32-bits floating point format (FP-32), which imposes high computational and memory burden on computing devices due to performing high-cost floating point MAC operations and transferring large sizes of data across the memory hierarchy, especially for large models with a large number of operations. To reduce the cost, quantizing the values of parameters and feature maps to a numerical format with lower bit-width is commonplace; however, due to reduced precision the final accuracy of the quantized models may be degraded in some cases. Here, we review some of the quantization methods available in the literature and their impact on the model accuracies.

Representing the values in fixed point format with less than 32 bits during the inference phase is one of the common quantization method, which requires significantly lower cost computation compared to computation in 32-bit floating point format. [GMG16] uses dynamic fixed point values in which the model parameters and feature maps are quantized to different bit-width due to the different ranges of their values. To recover the resulting accuracy degradation due to quantization, the under-studied models are fine-tuned (with the floating point format in back-propagation stage) until the final accuracy drop remains within 1%. Quantizing the values to integer format is another approach that is studied in [JKC+18], which quantizes the network parameters and feature maps to 8-bits integer values during the inference while during training the models, in the backpropagation pass floating point format is used and in the forward pass, the impact of quantization is computed. This method, although reduces the computation overhead, it drops the accuracy significantly (2% accuracy drop for ResNet models on ImageNet-1K are reported.)

Although quantization is more common in the inference phase of DNN algorithms, there exists some work that quantize the computed gradients and the model parameters and feature maps during DNNs' training as well. Binarized Neural Networks (BNNs) [HCS$^+$16] and XNOR-Nets [RORF16] binarize the weights and feature maps, and Quantized Neural Networks (QNNs) [HCS$^+$17] quantizes them to more than 1-bits during both inference and training phases. In addition, in XNOR-Net and QNN, the impact of binarizing/quantizing the gradients on the model accuracy are also studied. The accuracies obtained by all these methods indicates that excessive quantization, especially in training phase, can lead to significant reduction in the model's accuracy. In addition, most of these methods require elaborate modifications to the model and its training and inference phases and hence are not always easy to implement.

Low-Rank Factorization

Low-rank factorization methods aim to reduce the number of operations and parameters of a DNN by breaking down a matrix multiplication in fully connected layers or convolutional layers into several smaller matrix multiplications that have lower rank, thus lower number of parameters and operations.

To be more specific, let's go back to the operations in a convolutional layer discussed in Section 1.1 with a parameter matrix of W with the dimension of $F \times C \times K \times K$ and the input feature maps as a matrix X with the dimension of $C \times D_{in} \times D_{in}$. In general, the convolution of W and X in the convolutional layer can be formulated as follows:

$$Y_f = W_f * X \quad \forall f \in \{1, 2, ..., F\} \tag{1.2}$$

where W_f and Y_f is the parameters of filter f and its the corresponding output feature maps, respectively. Here, the dimension of W_f is $C \times K \times K$.

In low-rank factorization approaches, the parameter matrix of W_f corresponding to filter f is broken down into several low-rank matrices called separable filters [RSLF13] in different ways. One way upon which filter decomposition approaches in [TXZ$^+$15] and [JVZ14] are built is to represent the parameter matrix of a filter f (W_f) as a sum of R matrix multiplications between a set of horizontal and vertical matrices H_f and V with rank R (i.e., $W_f = \sum_{r=1}^{r=R} H_f^r (V_r)^T$). The matrix H_f consists of R horizontal kernels of shape $1 \times K$ and the matrix V consists of R vertical filters of shape $C \times K \times 1$. Therefore, the corresponding convolution in equation 1.2 is broken down into two convolutions according to the following equation:

$$Y_f = W_f * X = \sum_{r=1}^{r=R} H_f^r * (V_r * X) \tag{1.3}$$

Here, in this method the main challenge is to find the horizontal and vertical matrices to represent the parameters of a convolutional layer. In [TXZ+15], such matrices are obtained by computing the Singular Value Decomposition (SVD) of the weight matrix of a layer. To minimize the error of this decomposition, SVD decomposition is added in the forward pass of training a CNN model by breaking down the weights in the forward pass and back-propagating the corresponding error and updating the weights in the backward pass. This method due to computing SVD during training of CNNs results in high training cost; the inference phase, however, due to performing smaller convolutions is more efficient compared to the original model. Another approach to find such breakdown is introduced in [JVZ14], which proposes to learn the H and V matrices by defining a new objective function that reflects the approximation error as a difference between the original weights in a pre-trained model and approximated weights obtained by multiplications of H and V and learns these matrices such that the objective function (i.e., the approximation error) is minimized.

Another way of decomposing filters into the low-rank ones is to break them down into depth-wise and point-wise filters, which is a common method widely used for designing a small network for embedded devices such as MobileNet models [HZC+17]. Depth-wise and point-wise separable filtersare a decomposition of matrix W with a shape of $F \times C \times K \times K$ into C separate $1 \times K \times K$ depth-wise filters, which are convolved with feature maps in each channel of the input feature maps X separately and F point-wise separable filters with the shape of $C \times 1 \times 1$, which are convolved with the results of depth-wise filters.

Transferred Convolutions

Transferred convolutions are one of the interesting methods that compact a network by allowing high degree of weight sharing inspired by the translation symmetry existing in most of the perception tasks such as the tasks in computer vision [CW16]. Translation symmetry is a property that means data labels and distributions are invariant to shifts. By this property, shifting an image and feeding it to a network generate the same results as feeding the image to the network and shifting the outputs. This characteristic is held in all the layers of a CNN as well as in the first layer. Therefore, the output feature maps of a layer can be constructed by feeding a part of input feature maps to a layer and constructing the rest through a transformation function, which can help to share and reuse the weights to a large extent. More specifically, transferred convolutions compact the parameters of a model and reduce the operations by specifying a set of base filters, computing the corresponding feature maps and constructing the rest of the feature maps by spatial repetition of the feature maps obtained by the base filters.

An example of such approach is CReLU [SSAL16], which reduces the computation and parameters by $2\times$ through constructing half of the feature maps in each convolution layer as the opposite of the other half. Another method to transfer convolutions is called G-CNN introduced in [CW16] that reduces the computation and the size of parameters by rotating a fraction of the feature maps obtained by a set of base filters, which in size is a fraction of the number of parameters in the original model to construct all the feature maps. In these methods, although the size of feature maps remains the same as the original models, the number of MAC operations and parameters are reduced. Depending on the compression ratios, the accuracy of the models is degraded by up to 3% of the original models.

1.2.2 Hardware Optimization

The next set of existing solutions to improve the execution cost of DNN algorithms is designing specialized hardware architectures optimized for these algorithms. In these solutions the model computation is not modified and only the hardware is optimized to perform the computation efficiently. Here, in this section, we review some of the main architectures developed in academia and industry with the goal of improving the performance and energy consumption of the DNN models.

Due to high computing capacity of GPUs, especially with the added tensor cores along with CUDA cores, and high degree of parallelization, these computing devices are the main computing platforms for training large neural networks. However, due to their large size, high cost, high energy consumption and inefficient processing of single input, GPUs are a poor fit for inference especially on edge and end devices with limited area, memory and energy budget, where efficient processing of single input matters most. To improve the inference performance and cost, many ASIC or FPGA based specialized architectures are introduced [LCL+15, ZLS+15, XYP+17, AJH+16, CES16, JYP+17, FOP+18].

PuDiannao [LCL+15] is one of the ML accelerators that executes various types of ML algorithms and tasks such as classification, regression and clustering. This ASIC accelerator implements a general ML functional unit that supports various computation types used in ML algorithms such as dot-product, sorting, and etc along with various types of on-chip buffers to factor in various locality properties of ML algorithms and maximize reusability of on-chip data.

[ZLS+15] introduces an analytical method to optimize resource and bandwidth utilization on an FPGA platform to maximize the performance. It quantitatively analyzes the computation throughput and memory bandwidth for a CNN design with various optimization methods such as loop tiling and chooses the design with maximum performance and lowest resource requirements. In addition, [XYP+17] focuses on designing a high-performance FPGA accelerator for CNNs using systolic arrays with a number of PEs. To achieve such design, it proposes an automatic design space exploration framework that selects the best mapping of operations onto the systolic array to feed proper data to each PE location, the best PE array shape to maximize DSP efficiency, and a proper tiling size to maximize on-chip data reusability.

Cnvlutin [AJH$^+$16] is another design that eliminates multiplication of the zero operands by grouping the lanes and allowing them to execute operations independently while skipping zero values multiplications. It also introduces a data storage format that encodes the elimination decision and controls the lanes.

Another ASIC accelerator design is called Eyeriss [CES16], which is a spatial architecture consisting of a global buffer and an array of processing elements (PEs). Each PE is made up of several MAC units and local registers and connected to other PEs with a Network on Chip (NoC). The entire operations of a convolutional layer are parallelized on PEs based on a novel dataflow called Row Stationary (RS). RS dataflow maximizes reusability of both feature maps and weights through sharing a set of filters horizontally and a set of feature maps diagonally across the PEs in the PE array, which leads to reduced data movement across various memory units and energy consumption.

In addition, Google in [JYP$^+$17] introduces an ASIC architecture called Tensor Processing Units (TPU), which is hard-wired for parallelizing MAC operations in a matrix-matrix multiplications through a systolic array of 256×256 MAC units. Despite achieving high performance for a batch of dense layers in DNN models, this design suffers from resource under-utilization for processing a single input.

To efficiently serve single requests, Microsoft introduces a new FPGA based architecture called Brainwave [FOP$^+$18], which instead of parallelizing operations in a matrix-matrix multiplication, parallelizes operations in matrix-vector (MV) multiplications and constructs matrix-matrix multiplications through MV multiplications. To parallelize operations in convolutional layers, Brainwave employs a set of tile engines, each performing a native-sized MV multiplications and accumulates the results of all tile engines by a set of accumulators.

In addition to the mentioned 2D architectures, the challenges of designing CNN accelera-

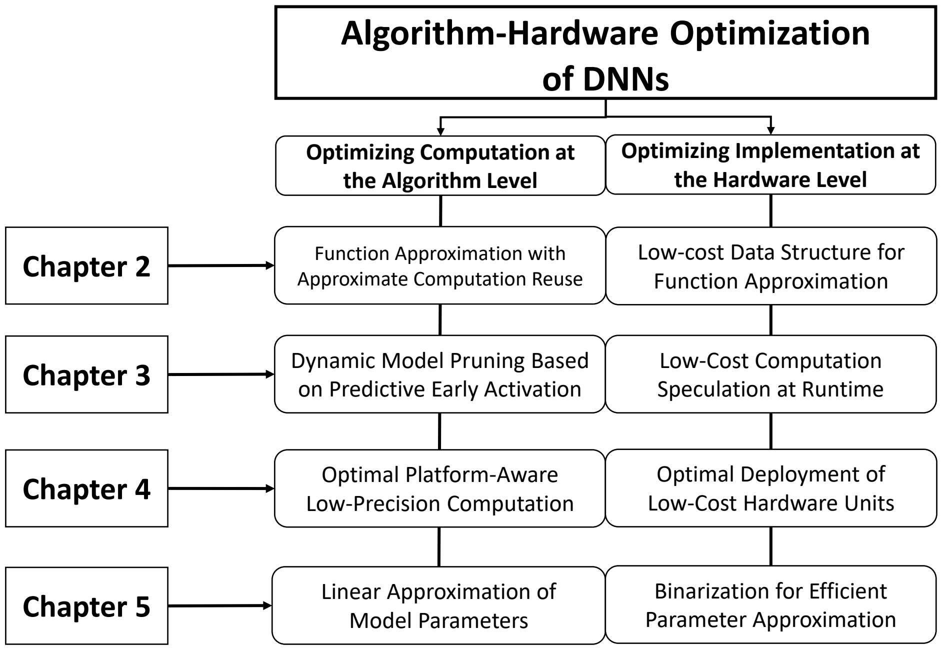

Algorithm-Hardware Optimization of DNNs
Optimizing Computation at the Algorithm Level
Optimizing Implementation at the Hardware Level
Chapter 2
Function Approximation with Approximate Computation Reuse
Low-cost Data Structure for Function Approximation
Chapter 3
Dynamic Model Pruning Based on Predictive Early Activation
Low-Cost Computation Speculation at Runtime
Chapter 4
Optimal Platform-Aware Low-Precision Computation
Optimal Deployment of Low-Cost Hardware Units
Chapter 5
Linear Approximation of Model Parameters
Binarization for Efficient Parameter Approximation

Figure 1.1: The organization of this dissertation

tors in 3D-stacked logic-in-memory computing systems are also studied in the literature and new accelerators such as Neurocube [KKC+16] and TETRISS [GPY+17] are introduced.

1.3 Dissertation Contribution and Organization

As discussed in the previous section, existing solutions to reduce the execution costs of DNN models in the literature mostly focus on optimizing either the underlying algorithms or hardware architectures. Even though the execution cost of these models are improved with the mentioned approaches, sometimes model accuracy is compromised and due to disregarding the other end of the spectrum, maximum benefit is not achieved. Therefore, substantial improvement without unacceptable accuracy degradation is still achievable by considering holistic optimization opportunities. In this dissertation, we explore various approaches for providing such optimizations to CNN accelerators. The organization of this dissertation is summarized in Figure 1.1.

In general, the methods presented in this dissertation are inspired by the fact that due to several reasons such as the algorithmic structures of DNNs, i.e., pooling and activation layers in their algorithms and specific functions at the end of their networks to make final decision based on the underlying task (e.g., softmax for classification), over-parameterization and repeated operations on similar inputs these models are error-tolerant. Therefore, approximate computation of models algorithms with less precise and a decreased number of parameters and operations along with efficient hardware architecture designs to support these approximate computation can lead to lower execution cost while achieving an acceptable accuracy. This dissertation exploits this characteristics and proposes several algorithm-hardware optimization approaches.

First, due to similarity of data and parameters in various CNN models, highly frequent patterns (approximate patterns) can be observed on the inputs (on a part) of inputs of MAC operations in their convolutional layers. Motivated by this observation, we propose an optimization approach at their algorithmic level by which the computed outputs for the matching patterns are reused in order to reduce the number of heavy MAC operations. A proper selection of these patterns (or approximate patterns) can achieve acceptable accuracy while lowering the cost. We optimize the hardware architecture as well for efficient implementation of this computation reuse by exploiting a low-cost data structure to memorize frequent patterns and match approximate patterns. Chapter 2 provides the detailed description of this algorithm-hardware optimization approach.

Second, due to the algorithmic structure of CNN models in which compute-heavy convolutional layers are followed by the activation layers, thus a specific range of their outputs are mapped to a predetermined set of values, a large number of the convolution outputs are not required to be thoroughly computed. Therefore, to reduce the number of MAC operations and memory accesses, we propose to dynamically identify and prune a large number of unnecessary MAC operations by speculating the range of convolutional outputs that are bounded by the activation functions with a fewer number of operations. To support such dynamic pruning at the hardware level, we design a low-cost hardware architecture with a slight modification of the prevailing accelerator architectures. This technique, called SnaPEA, is explained in more detail in Chapter 3.

Third, tolerance of DNN models to low precision computation and hardware units, non-uniform impact of imprecise computation in various layers/filters on the model output and different architecture designs and resource constraints in different underlying computing platforms are the motivation behind our proposed approach in chapter 4, where we propose platform-aware algorithm-hardware approximation of various layers/filters of a CNN model in order to maximize speedup and energy saving. Due to a large design space, we propose a novel machine learning based optimization framework for automatic and quick exploration and selection of proper approximate version of computation in DNN algorithms and proper approximate hardware configurations to implement various layers/filters by taking the resource constraints of the underlying computing platform into consideration.

As the final approach, in chapter 5, to reduce the model size that helps to improve the inference cost by reducing the off-chip memory accesses and the training cost, especially in distributed and federated learning, by reducing the communication overhead (i.e., to transfer gradients of parameters among devices), we provide a plug-and-play solution that generates the model parameters from a lower dimensional space with a linear transformation. For further parameter reduction, this method mainly exploits the associativity between convolutional layers and generates all the parameters through a set of fewer auxiliary parameters that is shared among all layers. Given the limited energy of edge devices, we further optimize this approach in order to provide low energy inference on edge through representation and reconstruction of models parameters with binary parameters and operations.

Finally, chapter 6 concludes this dissertation.

Chapter 2

Hardware Efficient Function Approximation

Current hardware implementations of neural networks exhibit high energy consumption due to the intensive computing workloads. The problem exacerbates, especially, when these models are executed on massively parallel architectures such as GPUs, which bring large-scale computations on a single device at the expense of significant energy consumption. Inspired by highly repetitive patterns on costly operations observed in the computation of these models, we provide an energy efficient implementation of these models through proposing a novel function approximation. We provide an energy efficient yet controllable function approximation using the probabilistic membership provided by Bloom filters (BF). A set of BFs is integrated into the functional units (FU) to store and detect frequent patterns for computational reuse. Depending on the applications, the computation reuse can be expanded through approximate pattern matching that detect frequent patterns on narrow precision data. Our approach has the ability to control the error behavior of a target function, hence the output quality, at the design time with the aid of controllable false positives (FP) available in the structure of the BF.We further lower energy consumption by designing a resistive Bloom filter (ReBF) using memristor array. Our experimental results show that for Convolutional Neural Networks, the BFs enable 47.5% energy saving of multiplication operations, while incurring only 1% accuracy drop. While the actual savings will vary depending upon the extent of approximation and reuse, this work presents a method for reducing computing workloads and improving energy efficiency.

2.1 Introduction

Recent advances in neural networks have achieved impressive performance on various application domains such as medical diagnostics [YJZ$^+$06], image classification [KSH12], speech recognition [HDY$^+$12], and natural language processing [CWB$^+$11]. The continued success of neural networks has led to their implementation on a variety of hardware platforms [CLL$^+$14][HAM07][CDS$^+$14a]. Energy consumption is an important metric for their implementation in increasingly broad range of computing platforms. Arithmetic operations and memory accesses constitute a significant source of energy consumption in deep learning accelerators. We focus here on reducing the computational workloads in neural networks.

In recent literature, computational workloads have been addressed by using approximations in computations thus creating a tradeoff between accuracy and energy [DLC$^+$15][MSS$^+$16]. The approximations can be made both in hardware or in software. For instance, approximate computation units have been shown to have better energy efficiency than the exact ones [JLL$^+$17]. Neural network computations are dominated by additions and multiplications. Due to their cost and latency, multiplications have been a natural target for optimization in hardware. For instance, in [DLC$^+$15], the authors substitute the normal multipliers with inexact multipliers that provide inexact logic but with less hardware cost. Mrazek *et al.* further optimize approximate multiplier design with a uniform structure suitable for hardware implementation [MSS$^+$16]. While the adaptability of neural networks in its applications is naturally suited to use approximation, in practice it also requires *retraining* the network to mitigate accuracy loss caused by logic errors from inexact design. Moreover, once the design has been physically implemented in hardware, it is not possible to reconfigure the design to control the approximation level entirely in hardware.

To overcome above-mentioned limitations, we propose using a reconfigurable and controllable approximation technique in neural networks by exploiting the computation reuse opportunities. Computation reuse has been adopted in various applications where *value locality* and *similarity* are observed [RGLM$^+$14]. To enable computation reuse, we provide tight integration of Bloom filters (BFs) into the computation units in hardware, a data structure that supports approximate set membership queries with a tunable rate of errors to store frequent computation patterns and return the results without actual execution of energy-intensive float point units (FPUs).

To ensure effectiveness of computation reuse using Bloom Filters, we use a set of techniques. First, we perform approximate pattern matching instead of exact pattern matching in neural networks. This is done in the context of arithmetic operations on floating point numbers. We thus explore matching operations under limited precision of operands. This is done via a reconfigurable BF architecture that can do approximate pattern matching with hashing for data items that feature varying bit width. Second, we perform layer-based pattern matching instead of global pattern matching. That is, we detect and store different set of input patterns for each layer separately. The reason is that in neural networks, each layer has its own set of functions thus may experience different input workloads. Accordingly, we configure BFs for each layer separately. Third, we implement the BFs with resistive memory elements to provide energy efficient storage for saving the frequently used patterns. Based on our implementation and evaluation, we make the following contributions:

- We design BFs to generate an approximate function with a guaranteed error bound. Hence, a set of BFs is tightly-coupled to individual functional unit (FU). This set of BFs approximately represents highly frequent computations of the associated FU. Each BF reports no false negatives (i.e., recall rate of 100%), and has a *tunable* parameter to control false positives.

- We explore and use computation reuse opportunities in multiplication operations of neural networks and enhance them with layer-based approximate pattern matching.
- We design a reconfigurable Bloom filter unit that can perform approximate pattern matching, increasing the computation reuse opportunities while leading to a controllable approximation level for neural networks.
- To further lower energy consumption and enable scalability, we utilize low-power memristor array in designing resistive Bloom filter (ReBF).
- We demonstrate the effectiveness of the approximate BFs by reducing 47.5% energy consumption of multiplication operations of LeNet CNN used for MNIST dataset in 45nm technology while incurring only 1% accuracy degradation. To show the possibility of using this approach for other applications and reduce GPUs energy consuption, we integrate ReBF into the Southern Islands GPU and simulate five image processing kernels on it. Caltech 101 computer vision data set [Cal] is used for profiling and finding the error bounds. The experimental results show that five image processing applications save on average, 24.1% energy consumption when we exploit ReBFs to represent their partial functionality along with FUs.

2.2 Related Work

For the purpose of improving energy consumption by accepting erroneous computations, several circuit and architecture techniques including voltage overscaling (VOS), and hardware approximation are proposed [RVPR13], [KK12], [GMRR13], [GMP+11]. For instance, [RVPR13] presents a synthesis methodology to allow further VOS beyond the critical points.

Another set of typical approaches are based on the use of inexact designs to replace normal computation units. Approximate adders are introduced by reducing circuit complexity [GMRR13, GMP+11]. Low power imprecise floating point arithmetic units for GPUs are presented in [ZPL14]. For energy efficient implementation of neural networks, Venkataramani *et al.* evaluates the impact of different neurons on neural network accuracy and selectively approximate the less-critical neurons with dynamically configurable accuracies [VRRR14]. A Similar work [ZWT$^+$15] replaces less-critical neurons with approximate ones and skip some neuron operations. These two works focus on finding the opportunities for approximate computing without significantly degrading the accuracy. Du *et al.* proposed an inexact multiplier design using an inexact logic minimization method, and emphasizes the need to approximate multipliers rather than adders. A hardware optimization approach was proposed in [MSS$^+$16] to design multipliers in a uniform way that suits physical VLSI implementation. Although these techniques offer significant energy saving, they cannot control the erroneous behavior for unseen inputs. These techniques typically consider a specific set of input patterns to design approximate hardware/circuit, hence they cannot guarantee error bounds for all data set. To increase the effectiveness of approximate methods in practice, efficient approaches with guarantee on error bounds are necessary.

BFs are among the data structures that can provide compact and efficient representation of a set of values with bounded error rate. BFs are used extensively to keep track of incoming data and states of flows in network applications [BMP$^+$06], [DKSL03]. For instance, data structures using both Bloom filters and hash tables are presented to concurrently track network flows [BMP$^+$06]. Another work provides hardware-based solution to find packets containing predefined signatures [DKSL03]. Multiple BFs with a set of signatures are used to parallelize search operations [DKSL03]. The hardware implementation of these techniques are expensive since they strive to return an exact value by performing one-to-one mapping between inputs and outputs, leading to large BFs.

This work provides an innovative use of fast parallel lookups to significantly enhance the space of pre-computed results table through functional approximations. We employ a set of BFs to approximately represent few output values of a function which occur frequently. Performing such *many-to-few* mapping using BFs imposes errors on the function outputs; however, we have the opportunity to bound the error rates by adjusting the false positives (FP) rate of BFs. To further lower the energy cost, we design resistive BFs exploiting low-power memristor elements.

A common approach for approximate computation reuse provided in the literature is based on using content addressable memory (CAM). Rahimi *et al.* uses content addressable memory (CAM) to perform computation reuse in GPU and perform voltage scaling on CAM to enable approximate pattern matching within a specified hamming distance [RGLM+14][IPK+17]. However, such approximate pattern match is hard to control because the bit mismatches could also occur in the exponent field.

In summary, our work differs from the previous works in two aspects: 1) We propose the first approximate Bloom filters to exploit and enhance the computation reuse opportunities. Such BF design can enable a reconfigurable as well as a controllable approximation by matching patterns with specific bit positions. 2) This approach does not require retraining and the approximation level can be tuned to satisfy the accuracy constraints.

2.3 Function Approximation with Bloom Filters

2.3.1 Bloom Filters (BFs)

BFs provide a compact representation of a set of elements. A BF consists of an m-bit vector which is programmed using k random hash functions. For a given element, k bits of the Bloom vector, selected by the k hash functions, are set in the programming stage. For a look up process, the same hash functions are computed on an input pattern, the membership of which, is queried. If the k bits in the vector determined by k hash values are all set, the input pattern is said to be present in the BF. Since those k values may also be obtained by any of actual members, the presence of an input pattern in the BF may be confirmed erroneously. On the other hand, if at least one of the k bits is not set, the input is certainly not the member of the BF. This explains that a BF has a recall rate of 100% and there is no false negative; however, it allows false positives

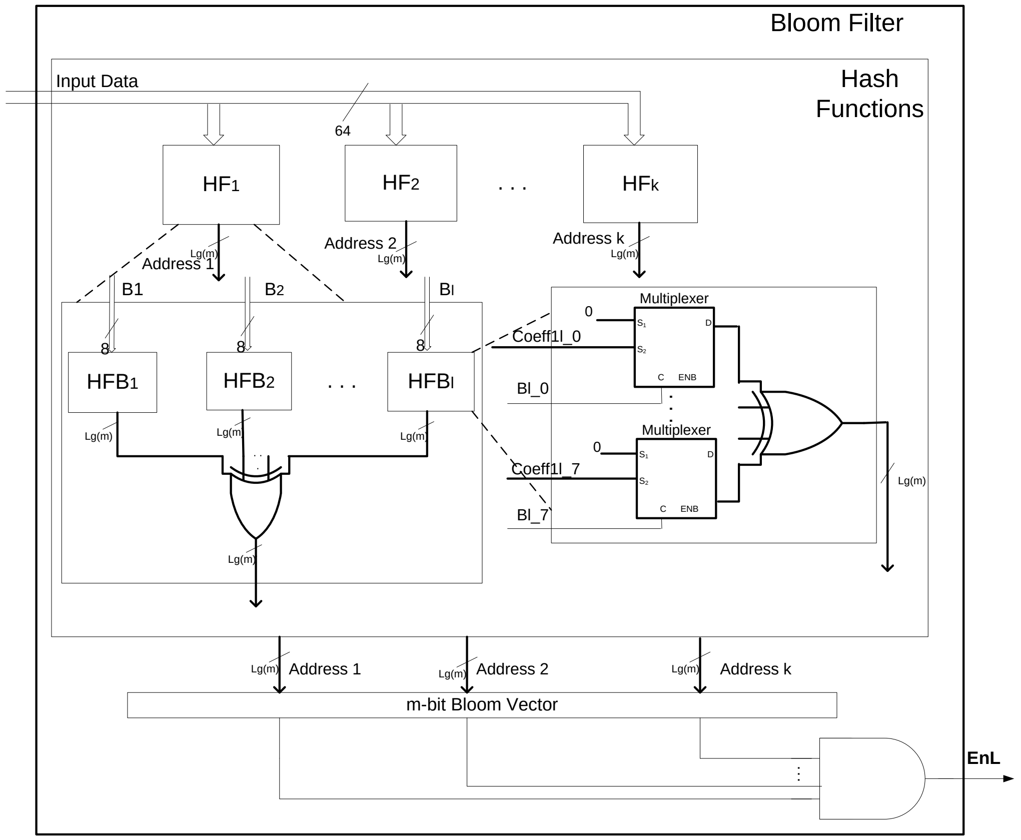


Figure 2.1: The hardware architecture of Bloom filter.

(FP), the rate of which is formulated as:

$$fp = (1 - e^{-\frac{nk}{m}})^k \tag{2.1}$$

where, n is the number of patterns saved in the BF, m is the length of Bloom vector, and k is the number of hash functions. False positive rate of a BF can be controlled by any of the aforementioned parameters. For a given m/n ratio, FP rate can be reduced by increasing the number of hash functions. In addition, the length of the vector (m) should be large enough to ensure that the FP rate is small for a given number of elements saved in the BF.

The hardware implementation of the BF is represented in Figure 2.1. The Bloom vector is shown as an m-bit array. For a look up operation, BF, at first, computes k hash functions (HF)

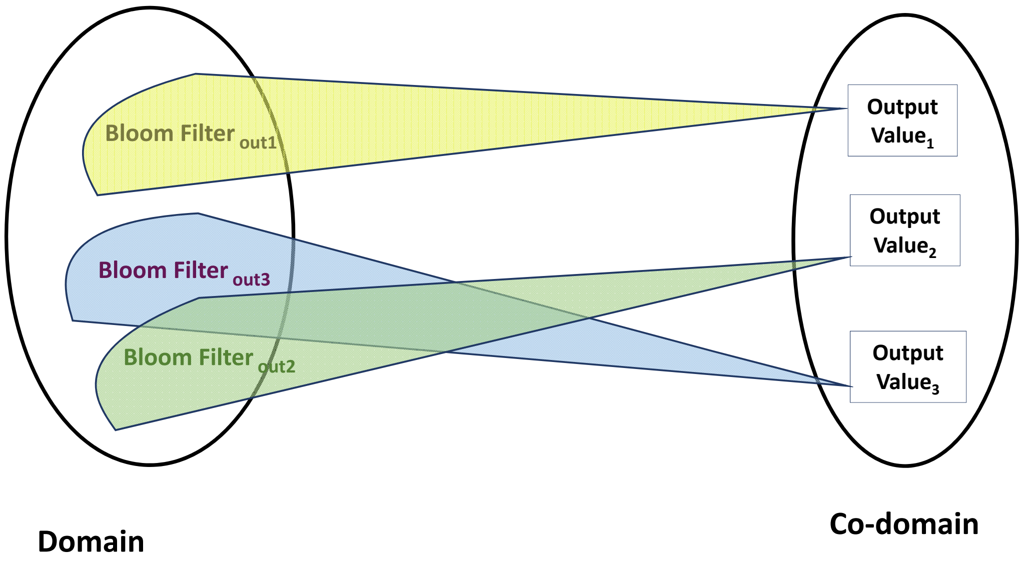


Figure 2.2: Function approximation using Bloom filters

concurrently to produce k addresses. The input data is also split into the number of bytes to parallelize the computations of each hash function. Each part of the computations in a HF is performed on one byte of the data through HFB module. Each HFB is made of XOR gates, and a number of coefficients, each has $log_2(m)$ bits. Coefficients are XORed if the corresponding bit of the input is one. The last XOR gate in each HF produces the final address by XORing outputs of HFBs. If all of the k bits of Bloom vector are one, hit occurs and the EnL signal becomes one.

2.3.2 Utilizing Bloom Filters for Function Approximation

In this section, we describe our proposed method for function approximation using BFs to reduce energy. Our proposed method expands the probabilistic membership query to function approximation. The method exploits value locality and similarity due to the presence of redundancy in the input data and the nature of applications.

To avoid redundant computation overhead due to re-execution of an operation for the same inputs, we identified highly frequent output values in the function co-domain and stored their corresponding input patterns in a set of BFs, as shown in Figure 2.2. For example, input patterns of *output value_1* in function domain are mapped to *output value_1* in function co-domain through *Bloom filter_out1*. To perform a function for a given input, at first, the membership of the input is queried in the BFs associated to the function. If a hit event occurs, the corresponding output value is returned as the final output of the function, and all pipeline stages in the implementation of the functional unit (FU) are clock-gated to save energy. As described in Section 2.3.1, positive responses of the BF to the membership queries are not always correct due to the FP. The source of error in our computation, is a FP event where the BF wrongly reports the *output value* as the function output. Therefore, depending on the FP rate of the BF, the returned output values of the FU for some input patterns are erroneous. Here, the rate of the error corresponds to the FP rate, and can be controlled by tuning the parameters affecting the FP (i.e., *n, m, k*). In case of a miss, the FU continues computing the results for the inputs, the memberships of which, are not confirmed in the BF. Due to absence of the false negative in the structure of BFs, the exact computation is performed for the mismatched input patterns. Based on the aforementioned details, a set of BFs can be exploited to approximate partial outputs of a function with the bounded errors that provide guaranteed quality. The output quality, here, is maintained by adjusting the FP rate of BFs. To approximate the whole functionality of a FU using BFs, all of the inputs of the function are required to be identified and stored in a set of BFs. Accommodating a large number of input patterns in a BF, while meeting the desired error rate, and storing the corresponding pre-computed output values may produce large BFs and registers, thereby, increasing the energy overhead. We address this problem using the two facts. First, because of the value locality and similarity [LWS96, AFL97] especially in data-level parallel applications [SJLS14, RBG13] few number of output values occur frequently at the run-time. The second fact is that the output values in the function co-domain can be classified into a number of clusters. Therefore, a few number of output values (i.e., the centers of clusters) becomes the representative of all output values. This characteristic is useful for approximating a function by returning the value of the cluster center as the function output instead of an exact value in the cluster. The function approximation using the co-domain classification necessitates a probabilistic data structure with the bounded error rate. Hence, BFs that demonstrate controllable error behavior, are applicable to our purpose.

2.3.3 Bounding Bloom Filter Errors at Design Time

In our method, the degree of approximation is entirely controlled by a set of BFs integrated to the FU. Probabilistic membership provided by the FP in BFs results in occasional false hit of the input patterns in the BFs. Returning a wrong output value for an input which is not stored in the BF, while wrongly considered as BF members, is manifested as an error at the output of the function. However, we are able to limit the error rate to meet the acceptable output quality by limiting the FP rate of associated BFs at the design time. As we mentioned earlier, the FP rate of a BF depends on several parameters such as the number of element stored in the BF, the number of hash functions and the length of the Bloom vector. A system, therefore, has the capability of allowing acceptable error by tuning these parameters of BFs attached to the FUs. Our approach to approximate a function can be applied to applications that are amenable to approximation. In the rest of this chapter, we explain how to use BF to perform approximate version of costly operations in image processing applications and in CNNs.

2.4 Function Approximation in Image Processing Applications

In this section, we focus on image processing applications harnessing floating-point FUs in the GPUs. The multimedia applications exhibit tolerance to the error, and offer the well-known notion of output quality with peak signal-to-noise ratio (PSNR). With approximate computing, an application with PSNR of equal or greater than 25 dB can still appear to execute correctly from the user's perspective [Bar06]. In GPUs, floating-point FUs consume higher energy per-instruction than their integer counterparts, and the overall arithmetic operations contribute to more than 70% of the total GPU power consumption in compute-intensive kernels [ZPL14]. FUs in the GPU architecture, used to map the most of arithmetic operations of a kernel, are targeted for integrating BFs. To determine the number of BFs for each FU, we need to configure them to ensure that the output quality will never go below the desired threshold. We require, at first, to identify the maximum tolerable error rate for each FU in a given kernel. We set the error magnitude conservatively to its maximum value for each FP event during the design time analysis. Then, the parameters of BF should be set to yield the FP rate less than or equal to the error rate. To do that, we need to specify the number of inputs to store in the BF through profiling. For the sake of clarity, we describe the process of configuring BFs in more detail, given the fact that four types of operations are identified in GPU architecture: adder (ADD), multiplier (MUL), multiply-accumulator (MAC) and SQRT.

2.4.1 Maximum Tolerable Error Rates

To find maximum tolerable error rates of the FUs used in a kernel, we use the following algorithm, and simulate the kernel for 30 different images using Multi2Sim [Mul]. The desired output quality, here, is assumed to be equal or greater than 25 dB. However, the algorithm has the ability to find the maximum error rate for any given PSNR threshold.

The first step is to find the maximum tolerable error rate for each individual operation. To achieve this, one operation is selected and random error with different rates is injected into it. We start from error rate of 0.1 and decrease it until the average PSNR of final output images becomes acceptable. The obtained error rate is, then, assigned to the operation. The process is repeated for all operations in a kernel (the error rate of 0.001 is obtained for each FUs in the Sobel filter). The next step aims to inject errors to all FUs to see their combined effect on the output quality. To do that, we inject errors simultaneously into all operations with the error rate assigned to them in the previous step. If the average obtained PSNR for 30 images are satisfactory, the algorithm stops. In case of unacceptable output quality, in the third step, we identify the frequency of each operation using the assembly code of the kernel generated by Multi2Sim, and sort them. Then, we decrease the error rate of the most frequent operation (MUL in Sobel) by ten times. If the output quality is not still acceptable, the next FU in the sorted list will be chosen to decrease its error rate. This step is repeated until acceptable PSNR is achieved. The algorithm ends up with maximum tolerable error rate for each operation. For example, in Sobel filter application, error rate of 0.001 for ADD, MAC, and SQRT and error rate of 0.0001 for MUL leads to average PSNR of 28 dB.

Table 2.1: Maximum Acceptable Error Rate and Output PSNR

App	ADD	MUL	MAC	SQRT	PSNR (min, max, avg) (dB)
Sobel Filter	0.001	0.0001	0.001	0.001	(26.4,32.9,28.0)
Sharpen	–	0.01	0.01	0.01	(24.7, 36.8, 28.12)
Roberts	–	0.001	0.001	0.001	(25.4, 32.5, 27.9)
Prewitt	–	0.01	0.001	0.01	(25.2, 33.3, 27.1)
Scharr	–	0.001	0.0001	0.001	(26.1, 31.1, 27.2)

Table 2.1 summarizes the maximum tolerable error rate of each FU for five image processing applications, and minimum, maximum and average PSNR of the output images. To ensure that our proposed approximate architecture generates acceptable output, the associated BFs to each operation must display a FP rate of lower or equal than the pre-determined error rates.

2.4.2 The Number of Input Patterns Saved in BF

We performed profiling on inputs and outputs of every floating point operations in the kernel using 10% of the training samples. In this phase, the frequency of each output value and the number of different input patterns responsible for that value are extracted. We also choose a number of output values to assign a set of BF to them. To increase the frequency of computational reuse (i.e., hit rate), we should increase the number of BFs and their size to save inputs of more output values. If we want to save all inputs of each output value (e.g., 329 inputs for most frequent output value of MAC in Sobel), large size of BF is needed, hence increasing energy overhead. To overcome this problem, we investigate the frequency of each input pattern for a specific output value. We observed that a few number of input patterns for an output constitute most fraction of the hit rate achieved by that output (e.g., one input for the most frequent output value of MAC in Sobel leads to hit rate 6.2% (compared to 7.3% obtained by 329 inputs)). Therefore, for each output value, we saved its most frequent input patterns.

2.4.3 BF Configuration

After specifying the FP rate of BFs, which is determined by the tolerable error of the unit BFs are integrated into and the number of elements to save (i.e., n) in the profiling stage, we change the value of m which indicates the length of BF vector, and k, which indicates the number of hash function to be used, to meet the predetermined error rates (i.e., the error rate of each units which is shown in Table 2.1) and design a BF for each operation to investigate its energy overhead.

2.5 Function Approximation in CNNs

Here, we explain the process of using our proposed technique to improve energy consumption of CNN architectures. Without the loss of generality, we apply the process on LeNet, a CNN used for MNIST dataset.

2.5.1 LeNet Architecture

Fig. 2.3 depicts the architecture of LeNet that consists of six layers, where the first, third, and fifth layer are convolutional, while the second, fourth are pooling layers, and the sixth layer is a fully connected layer. Consisting of a set of learnable filters, the convolutional layer is the core building block of a CNN. It performs the convolution operations on each filter and a portion of input volume, where a large number of costly multiplication operations are used, generating a new output image, namely a feature map. Then the pooling layer is used to reduce the size

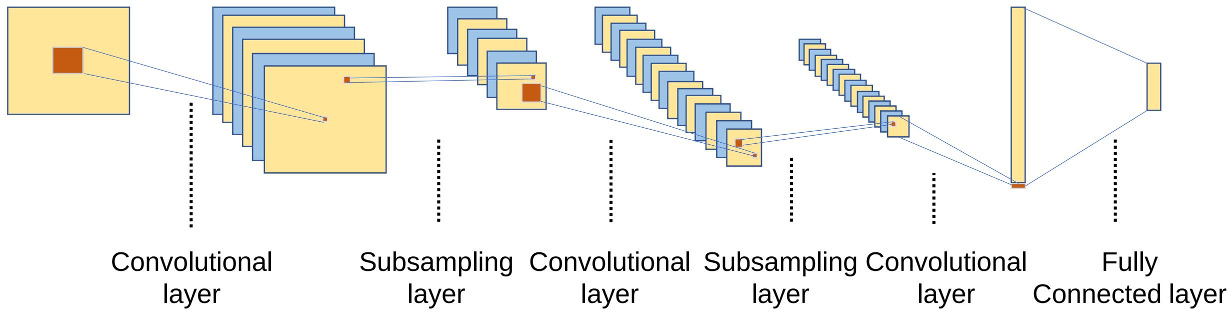


Figure 2.3: An illustration of a convolutional neural network.

of a feature map by averaging various pixel strengths. This process only preserves the most informational features of input by dropping the unnecessary minor information. We integrate BFs in multiplication units used in convolutional and fully connected layers, which account for 98% of multiplications in our experimented neural network [MSS+16].

2.5.2 Layer-Based Pattern Matching

To maximize the energy savings, we need to maximize the computation reuse opportunities. Since we need to store a set of pre-calculated computations, we aim to store most frequent input patterns to maximize the computation reuse opportunities. To do this, we use several steps. First, we profile the input operands of multiplications using some training input. Second, in the profiled input, we look for the most frequent input patterns and calculate their results.

In this process, we use two strategies to look for the most frequent input patterns: *global-based* and *layer-based*. *Global-based* means we look for the most frequent input patterns from all the multiplication operations in neural network inferences, regardless of their locations. *Layer-based* means we look for the frequent input patterns for each layer separately. That is, for each layer, we find the most frequent patterns from the input operands of multiplications profiled from that specific layer. For example, to find the most frequent patterns for the third convolutional layer, we profile all input operands of multiplications in that layer and find the most frequent patterns in this set of input operands.

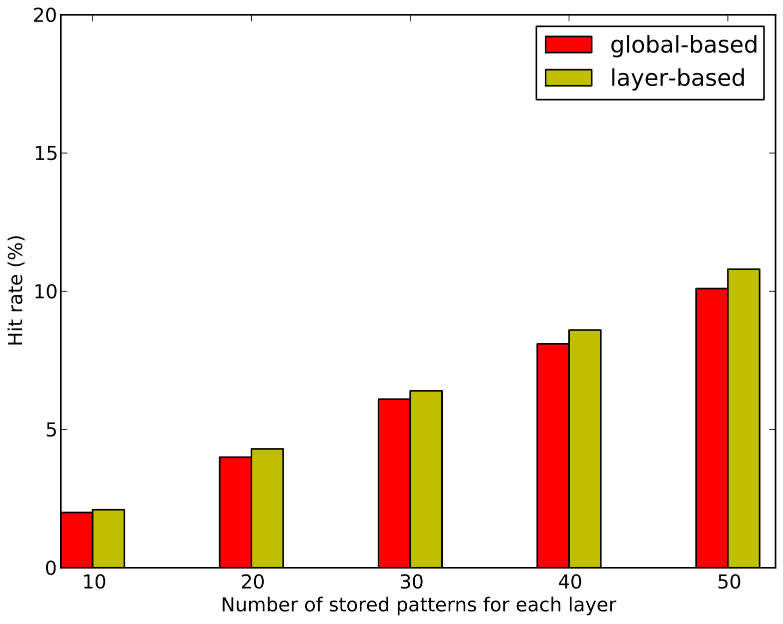


Figure 2.4: The hit rate of exact pattern matching.

Third, we then check the hit rate of the chosen frequent patterns using another set of data. We also vary the number of stored input patterns for each layer. Note that, for the sake of simplicity, we always use the same number of stored patterns for each layer. As shown in Fig. 2.4, we can see that layer-based matching leads to higher hit rate than global-based matching. From now on, we conduct all of our experiments using *layer-based* approach. We also observe that as the number of stored patterns increases, the hit rate also increases. However, the hit rate still remains low, at around 10%, even if we store 50 patterns for each layer. Thus, we improve the hit rate by developing approximation techniques as described in the next part.

2.5.3 Approximate Pattern Matching

As shown in Fig. 2.4, even if we use layer-based pattern matching, the hit rate is still low. Thus, we propose the use of approximate pattern matching for floating point numbers instead of exact matching, i.e., we only match for limited bit width. For example, there are two floating point numbers 0.45 and 0.451, with their IEEE 754 format as 00111110111001100110011001100110

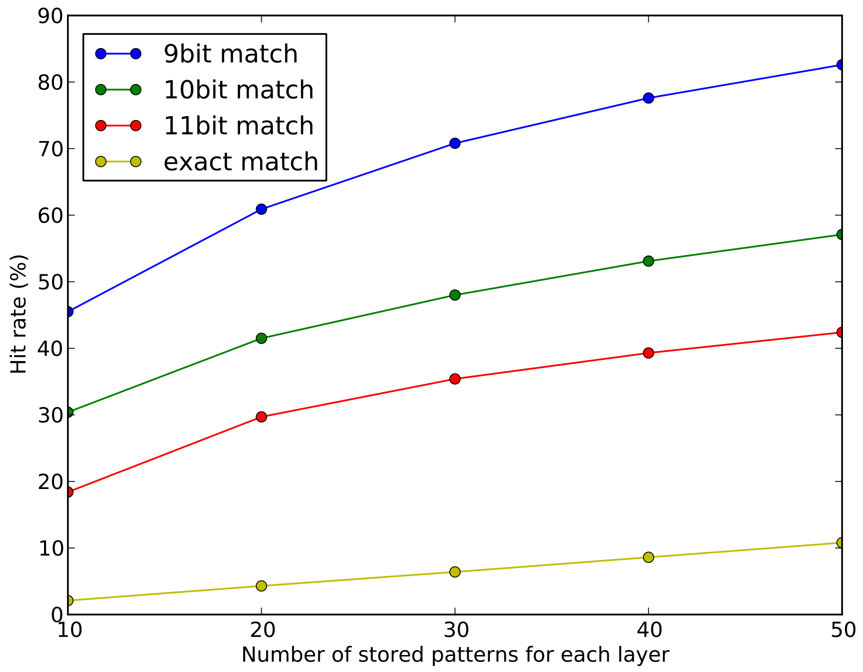


Figure 2.5: The hit rate of approximate pattern matching.

and 00111110111001101110100101111001. If we use exact matching, then 0.451 would not match 0.45. However, if we use 9-bit matching, then 0.451 would match 0.45-because their first 9 bits (sign bit and exponential bits) match. In this case, their first 16 bits (sign bit, exponential bits and 7 mantissa bits) are identical so they will match even under *16-bit matching* mode. We use four different approximate matching modes to measure the hit rate: 9-bit, 10-bit, 11-bit, and exact matching, as illustrated in Fig. 2.5. We can see that as we increase the approximation level, the hit rate also increases significantly even by 1 bit. For example, by storing 50 patterns (for each layer), a 10-bit approximation can have hit rate at 57.1% while 9-bit approximation can have hit rate at 82.6%, which is 56% higher.

But note that the increased hit rate does come with a cost. Rather than returning the exact computation result, the approximate pattern matching will return an inexact result. And as we increase the approximation level, the extent of inaccuracy will also increase. We explore several different approximate matching modes in the experimental results discussed later by varying the matching bit width and number of stored patterns to maximize the energy savings while keeping

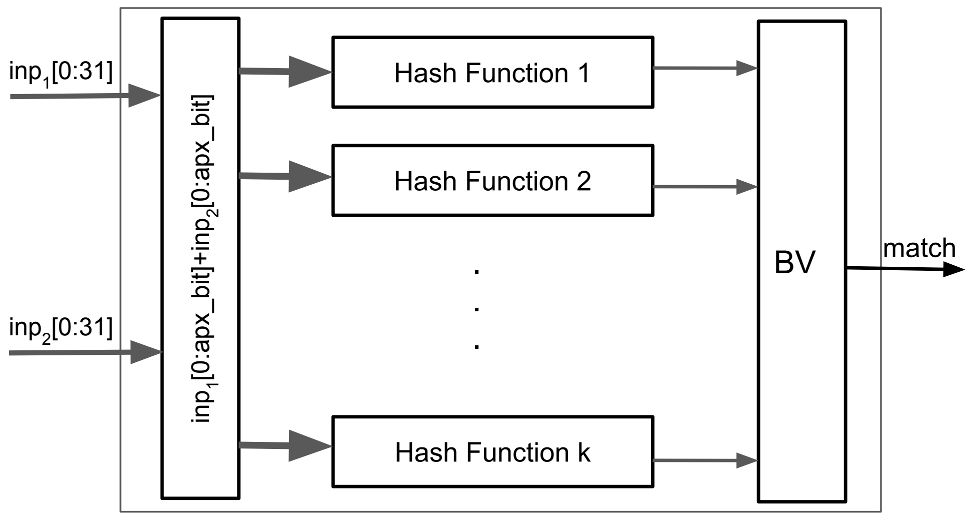


Figure 2.6: The implementation of approximate pattern matching.

the accuracy loss minimal.

2.5.4 Approximate Pattern Matching with Bloom Filters

To implement approximate pattern matching at the hardware level, we employ BFs. The detected frequent patterns are stored in a set of BFs, and the BFs are integrated to the multiplier. The number of BFs equals to the number of distinct output values generated by the frequent patterns. Each BF stores the patterns corresponding to its assigned output value.

The overall architecture of using BF for approximate pattern matching is shown in figure 2.6. In order to enable the approximate pattern matching, we store approximated input patterns in the BF. We resize each input of the multiplier to *apx_bit* bits by selecting its *apx_bit* most significant bits and concatenate them into a single vector. The obtained bit vector forms an input to the hash functions which determine bits to be set in the BF vectors. Similarly, to investigate the approximate pattern matching of incoming inputs to the multiplier, the inputs are re-sized and concatenated before going to the hash functions. Then, the bits in the BV specified by the hash functions determine whether the incoming pattern of the multiplier is matched or not. In case of matching, the multiplier is clock-gated to avoid the re-execution and the output in the register corresponding to the BF is returned as the output of the multiplier.

To control the drop in the classification accuracy of the LeNet that may occur due to FP rate of the BFs, the FP rate will be tuned by properly setting the parameters of the BFs (n, m and k). Since most of today's applications such as neural networks demonstrate tolerance to the controlled imprecision in computations, in this work, BFs with controllable error rate are adapted to implement approximate pattern matching and recall the computations in neural networks.

To further improve the energy consumption of the computations, we employ resistive memory elements to implement Bloom vectors, which exhibit significant energy savings than its CMOS counterparts [ARG16]. Moreover, resistive memory consumes little area overhead as it can be implemented on top of the chip [IPR16].

2.6 ReBF: Resistive Bloom Filter

Before introducing architecture of a resistive Bloom filter (ReBF), let examine energy efficacy of CMOS implementation of BFs. BF allows accommodating more patterns, hence, increasing the frequency of computational reuse. The more pre-computed results the BFs use, the more energy the entire system will save due to clock-gating of costly FUs. We accordingly implement BFs configured for five image processing applications using Verilog, and synthesized them with 45 nm standard CMOS library. Each individual output value is stored in a 32-bit register, connected to the corresponding BF.

Figure 2.7 illustrates the energy of the proposed architecture using CMOS BFs compared to the conventional one, which purely uses FUs. As shown, computational reuse with the aid of CMOS BFs offers negligible or no benefit due to the high energy overhead of implementing BFs using CMOS cells. For instance, Sobel incurs 23.8% energy penalty and on overage the overhead is 4.4%. This energy overhead is mainly due to the implementations of m-bit vectors that are turned into CMOS storage elements (i.e., about 70% of the power is consumed by the Bloom vec-

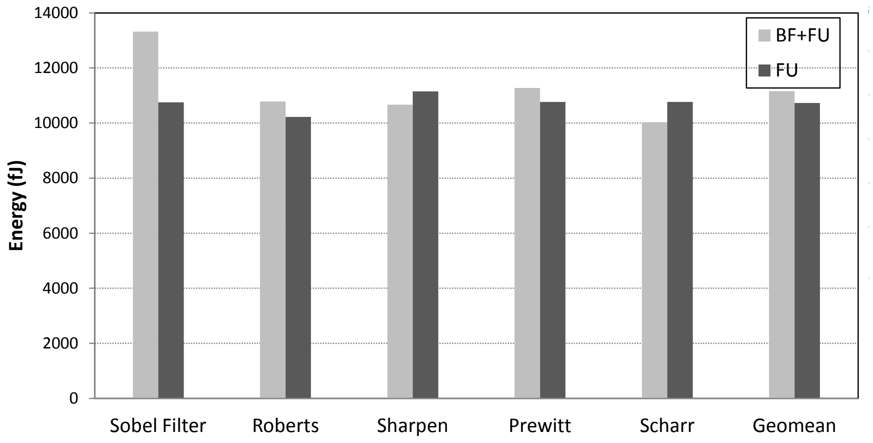


Figure 2.7: Energy consumption comparison of the proposed architecture using CMOS BFs and conventional FU.

tor). To improve the energy efficiency of our proposed architecture, we use 1-transistor/1-resistive (1T-1R) cells, a promising low-power technique, to implement the m-bit vector used in the BF.

2.6.1 ReBF Architecture

In this section, we describe the implementation of ReBF using the memristive elements. The architecture of ReBF is shown in Figure 2.8. Each bit of the resistive Bloom vector is implemented by a 1T-1R cell. The cells are connected to each other through a match line (ML). The stored value in each cell is represented by the value of the resistor element. High value of the resistor is considered as digital one, and low value as digital zero. The transistor in the cell is controlled by a select line (SL). During the programming stage, *k* hash values for each member are obtained using the combinational CMOS circuit, and, then, the resistor of the corresponding cell in ReBF is set to the high value. For the look up/search process, the ML is, at first, pre-charged to Vdd via the pre-charge circuit. Then, the same *k* hash functions are computed for the incoming input using CMOS technology, and *k* addresses are generated. These addresses will be decoded in parallel through *k* CMOS decoders. The outputs of decoders determine the cells in the ReBF to be verified. The select lines of the corresponding cells are, then, activated. This turns on the NMOS transistor in the cell. If the stored value in any of the *k*cells is zero (i.e., the resistor value

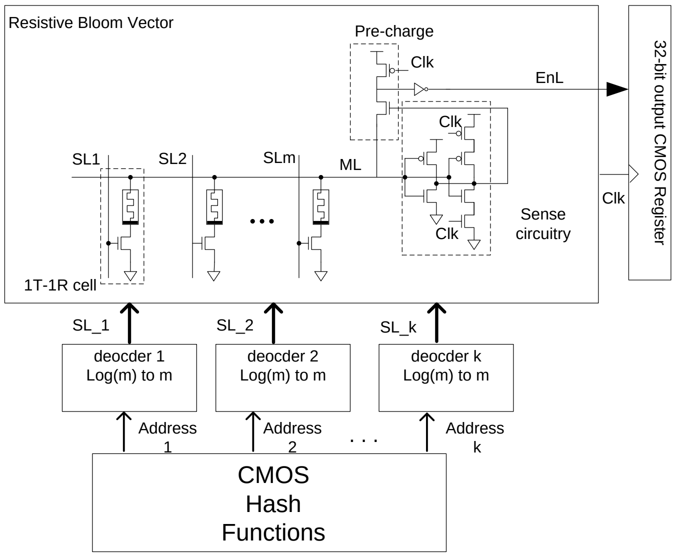


Figure 2.8: 1T-1R implementation of ReBF.

is low), a path between Vdd and ground will be established and the ML will be discharged. The voltage drop on the ML will be amplified by the sense circuit; therefore, a full swing will be observed on the EnL signal. This means that the incoming input is not saved in the ReBF. In this case, EnL prevents the output register to be read. The delay of the circuit is determined by the time it takes to pre-charge ML to Vdd, and the time it takes to discharge the line and observe the full swing on EnL. The worst-case delay happens when only one of the cells is not matched, and tries to discharge the ML. If more than one cell are not matched, the ML will be discharged rapidly, thereby, the delay is lower than the previous case. However, if all *k* cells are set to one, the resistor values are all high, and prevent the ML from discharging. This means that hit occurs and EnL signal becomes high, which leads to returning pre-calculated value stored in a CMOS register as the final output of the FU that uses the ReBF.

2.6.2 Scalability of ReBF

The low-power capability of ReBF allows it to save more input patterns using larger Bloom vectors compared to the CMOS BF. This means that further portion of a FU functionality

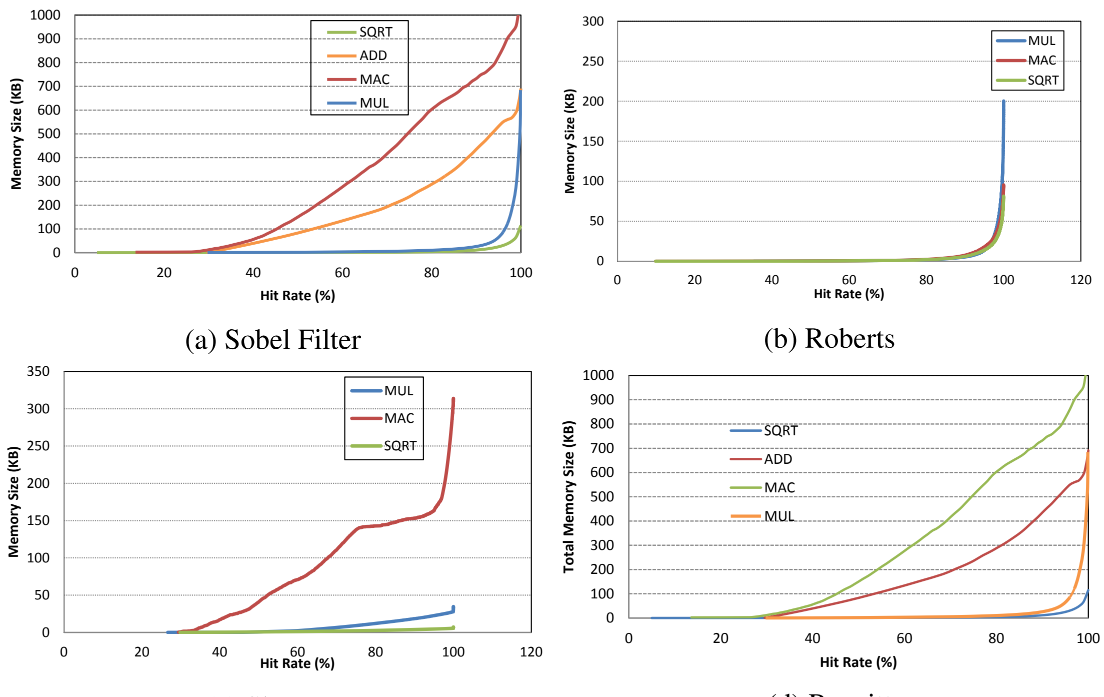


Figure 2.9: Total memory size vs frequency of computational reuse (hit rate)

can be represented by a set of ReBFs. To see the relation between the required amount of memory and the degree of representing a function using Bloom filters, we computed total memory size required for each operation in four image processing applications. The degree of function representation is equivalent to the hit rate in BF. To obtain the memory size, we set the FP rate of BFs for each operation to their pre-determined maximum error rate which guarantees acquiring the acceptable output images. Given three hash functions to map the input patterns to BF, total size of memory for different hit rates in four applications is shown in Figure 2.9. This memory size counts for the m-bit vectors and the output registers. As shown, the number of bits required to represent more functionality of an application increases rapidly, specifically in MAC operation of Sobel filter and Sharpen because of increased number of inputs compared to other operations with the same hit rate. This makes ReBF a feasible replacement for the corresponding function.

2.7 Experimental Results

2.7.1 Evaluation of Function Approximation on Image Processing Applications

To assess the efficiency of the function approximation using ReBF, we choose five image processing applications adopted from AMD APP SDK v2.5 [AMD]: Sobel, Robert, Prewitt, Scharr and Sharpen. The openCL code of these applications are simulated by Multi2Sim [Mul] to perform profiling and finding the error rate bound on four operations (i.e., ADD, MUL, MAC, and SQRT). We generates the VHDL code of these FUs as the six-stage pipeline unit, commensurate with the AMD Southern Islands GPUs [Mul], using FloPoCo [Flo]. The hardware realization of different size of BFs with different number of hash functions are implemented using Verilog. To compare the energy consumption of these hardware, the implemented FUs and BFs are synthesized using Synopsys Design Compiler, with 45 nm standard CMOS library. The operating voltage is set to 1.0V and the clock period is 1.5 ns. To estimate the power and delay of ReBF, transistor-level design of the Bloom vector of different sizes is performed using HSPICE with the same technology. We consider R_{ON} as 10K Ω and R_{OFF} as 1M Ω. We set the other parameters of the cell such as the capacitance of the line based on the one presented in [ZJ13].

Energy Saving

Energy consumption of an individual resistive Bloom vector (RBV) for different length of 2048, 1024, 512, 256, 128, and 64 bits is summarized in Table 2.2. The delay of the vector with different size is fixed at 1.4 ns. This is achieved by adaptively adjusting the operating voltage shown in the table. Energy consumption of CMOS Bloom vector of different sizes is also shown in the table. As we can see, implementing Bloom vector using memristor cells presents extremely low energy consumption even if the larger vector is used (i.e.,19.3 fJ compared to 1649.55 fJ for 2048 bit).

Table 2.2: Energy consumption Comparison of resistive Bloom vector in ReBF and CMOS Bloom vector

Size (bits)	**2048**	**1024**	**512**	**256**	**128**	**64**
Vdd (V)	1.0	0.71	0.6	0.54	0.51	0.47
RBV (fJ)	19.3	17.12	8.04	5.10	3.77	1.62
CMOS BV (fJ)	12188.4	6662.4	3992.4	2603.1	1837.35	1649.55

Table 2.3: Optimum ReBF configuration for different applications

FU	**Sobel**					**Roberts**					**Sharpen**					**Prewitt**					**Scharr**				
	#Out	#In	HR%	BV	#Fn	#Out	#In	HR%	BV	#Fn	#Out	#In	HR%	BV	#Fn	#Out	#In	HR%	BV	#Fn	#Out	#In	HR%	BV	#Fn
ADD	2	12	29.46	256	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
MUL	2	16	29.61	1024	3	6	26	41.61	512	4	6	15	42.18	256	2	8	30	59.4	512	2	4	12	27.7	256	4
MAC	4	12	25.44	256	4	10	12	29.5	256	4	6	15	30.2	256	2	6	15	32.6	1024	2	8	18	37	512	4
SQRT	16	16	20.8	512	3	10	10	29.5	256	4	-	-	-	-	-	14	14	82	256	2	18	18	10	512	3

To assess the efficiency of our approach, we profile five image processing kernels using 10% of Caltech 101 computer vision dataset [Cal], and find the maximum tolerable error rate for each FUs in the kernels as described in Section 2.4.1 using 30% of the dataset. For each kernels, we select various hit rates obtained from profiling stage. Then, we configure BFs based on the number of elements that depends on the hit rate, and the maximum error rate obtained for each FU. We evaluate energy consumption of our approach for each configuration to find the maximum energy improvement compared to the conventional FU without ReBF. For each application, we summarize the optimum configuration of the Bloom filters (i.e. the number of bits required for Bloom vector (BV) , and the number of hash functions (Fn)) integrated into each FU in Table 2.3. Table 2.3 also shows hit rates (HR) provided by the selected frequent patterns, the number of pre-computed outputs, and the number of stored inputs for each FUs used in the application.

For each FU in five applications, Table 2.4 contains the total energy consumption of hash functions, decoder, and output registers which are implemented using CMOS cells. Comparing to the small energy consumption of resistive vector shown in Table 2.2, CMOS modules constitute most fraction of energy consumption of the proposed architecture (e.g., for MUL in Roberts, 5.10 fJ is used for 256-bit ReBF vector compared to 1637.8 fJ for CMOS modules). This means that

Table 2.4: Total energy consumption (fJ) of CMOS components in ReBF, Hash functions, output registers, and decoder for FUs in different applications

FU	Sobel	Roberts	Sharpen	Prewitt	Scharr
ADD	918.94	-	-	-	-
MUL	882.63	1302.74	796.95	1179.72	996.76
MAC	1347.97	1637.83	879.07	1076.94	1815.01
SQRT	873.78	882.91	-	2329.27	644.58

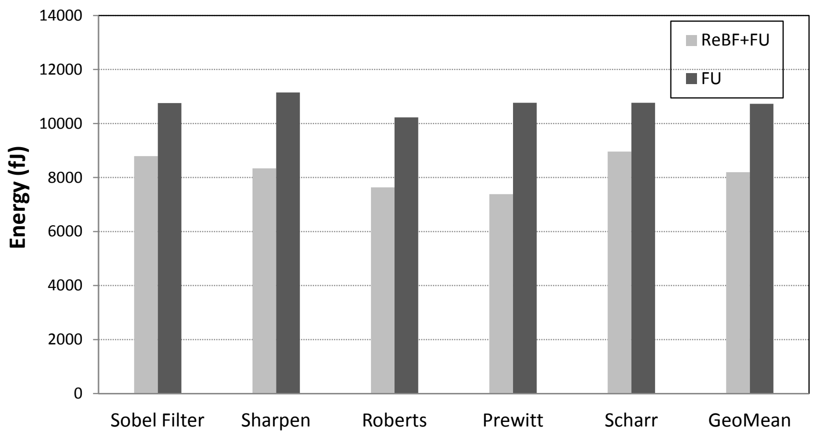


Figure 2.10: Energy comparison of the proposed architecture using ReBFs and conventional FUs.

the main dominant factor that inhibits us from further computational reuse, and energy saving is high energy consumption of modules implemented with CMOS technology. Comparing the energy consumption of resistive vector and CMOS modules shows that the main dominant factor that inhibits us from further computational reuse, and energy saving is high energy consumption of modules implemented with CMOS technology (e.g., for MUL in Roberts, 5.10 fJ is used for 256-bit ReBF vector compared to 1637.8 fJ for CMOS modules). Figure 2.10 illustrates the total energy of using ReBF (including resistive and CMOS parts) along with FUs and energy of conventional FUs without ReBF. We use the optimum configurations for ReBF shown in Table 2.3. For Sobel filter application, 21.7% energy is saved using the 256-bit Bloom vector with five hash functions for each ADD and MAC, and 1024 (512)-bit vector with three hash functions for MUL (SQRT). The bloom filters are used to store 12 input patterns for ADD and MAC, and 16 input patterns for MUL and SQRT.

ReBF also demonstrates 25.2%, 25.3%, 31.4%, and 16.8% energy improvement for Sharpen, Roberts, Prewitt, and Scharr, respectively. For Sharpen, maximum improvement is achieved by using ReBFs for MUL and MAC, to store 15 patterns for each module. To meet the maximum error rates, we employ bloom vector of size 256 bits, and two hash functions for each operation. The provided hit rates are 42.18% and 30.2% for MUL and MAC, respectively. In this case, we do not use ReBFs for SQRT. In Prewitt, since for each FUs higher hit rate is achieved compared to other kernels with nearly the same number of input patterns (e.g., 14 (10) inputs provides 82% (29.5%) hit rate in SQRT module of Prewitt (Roberts)), this application exhibits the most energy saving.

2.7.2 Evaluation of Function Approximation on CNNs

In this work, we use tiny-dnn [tin], a header only, dependency free deep learning library written in C++, as our evaluation platform. For CNN, we use LeNet-like architecture as illustrated in Fig. 2.3. We use MNIST (Mixed National Institute of Standards and Technology) database of handwritten numbers [LCB98] as our dataset to evaluate the accuracy. The dataset is split into a training set and a test set with 60,000 and 10,000 28×28 images. We randomly select 5% of the training input data to profile the frequent input operands. To estimate the energy consumption of the proposed design, we implement the hash functions using Verilog, and we extract the Verilog implementation of a six-stage pipelined floating point multiplier using FloPoCo [DDP11]. Then, the implementations are synthesized using Synopsys Design Compiler, with 45 nm standard CMOS library. The operating voltage is set to 1.0V and the clock period is 1.5 ns. In addition, Bloom vectors (BV) are designed with resistive 1T1R cells using HSPICE, where RON is set to 1K Ω and ROFF to 1M Ω [ZJ13]. Bloom filters can be used in different hardware platforms, including CPU [FSTN16], FPGA [DKSL03], and GPU [ARG16].

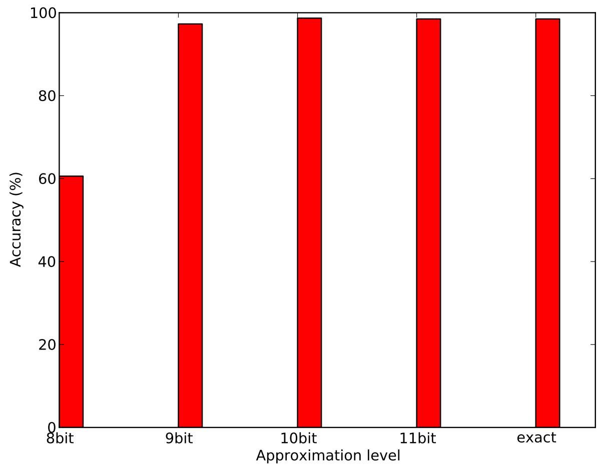


Figure 2.11: Neural network accuracy loss due to approximate pattern matching.

Accuracy Loss

As described in Section 2.5.3, the BF will return an inexact result due to approximate pattern matching. Thus, we investigate here on how the approximation level impacts the neural network accuracy. We vary the approximate pattern matching mode from 8-bit matching to 11-bit matching and store 10 most frequent patterns for each of the approximation modes with a FP rate of 0.001. Fig. 2.11 shows the accuracy under each configuration. The baseline accuracy is 98.5% without any approximations. The 8-bit matching introduces aggressive approximation because it does not cover the last bit in the exponent bits, which leads to only 60.6% accuracy. Starting from 9-bit matching, the accuracy loss is insignificant. Note that 9-bit matching covers the sign bit and exponent bits for floating point numbers. This indicates the high error-tolerance of neural networks to data imprecision.

According to Fig. 2.5, 9-bit matching gives us the highest hit rate among the approximation modes which introduces little drop on neural network accuracy. Thus, using 9-bit matching as our approximation mode, we then investigate how the accuracy will vary with the number of stored frequent patterns. As shown in Table 2.5, various number of stored patterns under 9-bit matching have little impact on neural network accuracy. The lowest accuracy is 97.2% under the (9, 50) configuration, meaning that we use 9-bit approximate pattern matching and store 50 input patterns.

Table 2.5: Energy savings and neural network accuracy across different BF settings.

Matching Mode	(BV size, #Hash_Fn, #inp_BF)	Hit Rate	NN Accuracy	E_{save}
(8, 10)	(64, 2, 1)	66.9%	60.6%	58.9%
(9, 5)	(64, 2, 1)	28.9%	97.5%	24.9%
(9, 10)	(64, 2, 1)	45.7%	97.4%	37.9%
(9, 20)	(64, 2, 1)	60.7%	97.9%	45.4%
(9, 30)	(64, 2, 1)	70.6%	97.4%	47.5%
(9, 30)	(32, 3, 1)	70.6%	97.4%	41.6%
(9, 40)	(64, 2, 1)	77.3%	97.3%	47.3%
(9, 50)	(64, 2, 1)	82.3%	97.2%	44.7%
(10, 10)	(64, 2, 1)	30.4%	98.3%	22.3%

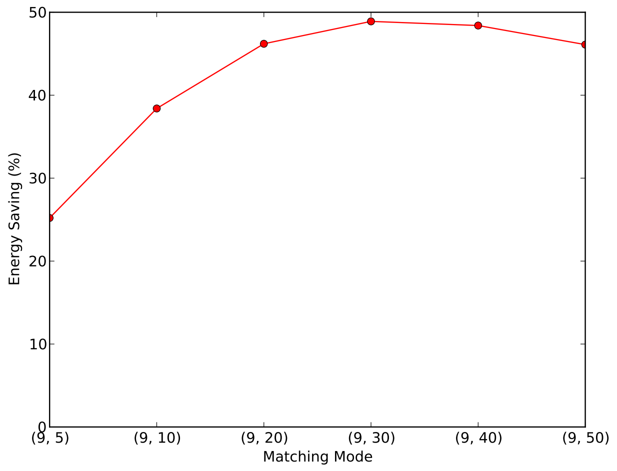


Figure 2.12: Energy Savings under different matching mode.

Energy Savings

We use several different matching modes and BF configuration to compute the energy savings and the resulting neural network accuracy as shown in Table 2.5. The matching mode ($appx\_bit$,$\#inp$) refers to how many bits we use for approximate pattern matching and the number of patterns we store. The BF setting (BV size, #hash_Fn, #inp_BF) refers to BV size in bit length (m), number of hash functions (k) and number of input patterns stored in each BF (n). For example, the BF setting at (64, 2, 1) means that we set the BV size as 64 bits, use 2 hash functions and store 1 input pattern for each BF. To satisfy the FP rate which can lead to acceptable accuracy, we carefully select BF configurations.

Table 2.5 exhibits several important facts. First, 9-bit matching is the optimal matching mode here. By comparing with 8-bit matching and 10-bit matching, we find that 8-bit matching achieves the most energy saving at 58.9%, but its resulted neural network accuracy is only 60.6%, a significant accuracy drop over baseline accuracy of 98.5%. 10-bit matching achieves higher accuracy than 9-bit because it introduces smaller approximation errors into the neural network than 9-bit matching but its resulting energy saving is only at 22.3%, which is less than the one obtained with 9-bit matching mode. Thus, 9-bit matching achieves the better balance between neural network accuracy and energy savings.

Second, after we fix 9-bit matching mode, we then look for the optimal number of patterns to store. We vary the number of stored patterns from 5 to 50. Note that all 9-bit matching modes, regardless of the number of stored patterns, achieve accuracy close to the baseline. Thus, we focus on locating the best energy saving setting. As shown in Fig. 2.12, we found that the energy saving increases as the number of stored patterns increases from 5 to 30 (we call it the first stage), however the energy saving starts to decrease as the number of stored patterns increases from 30 to 50 (second stage). This is because in the first stage, the hit rate increases as the number of stored patterns increases, which will reduce the use of multipliers. In the second stage, although the hit rate still increases, the energy consumption of BFs increases as the number of stored patterns increases, which dominates the energy consumption. Thus, we find that the optimal matching mode is (9, 30).

Third, we also try different settings of BV size and hash functions. To satisfy the FP error rate of 0.001, we use two realistic BF settings, (64, 2, 1) and (32, 3, 1). The BF setting at (32, 3, 1) consumes more energy than that of (64, 2, 1) because it uses 3 hash functions, which is the main source of energy consumption of BFs. In summary, the optimal configuration is (9, 30) as matching mode and (64, 2, 1) as BF setting. This leads to a neural network accuracy of 97.4% and energy saving of 47.5%.

2.8 Summary of the Chapter

Many algorithms including machine learning and image processing algorithms offer massive parallelism and significant degrees of tolerance to approximate computing. This paper aims to address the following challenge: *how to increase approximate computational reuse through non-volatile resistive storages in GPUs with bounded errors?* Combining computational reuse and approximate computing, we propose a resistive Bloom filter (ReBF) that provides an approximate representation of a function by integrating Bloom filters to the hardware functional units. BeBFs are used to store highly frequent patterns to avoid re-executions. Computation reuse capability can be enhanced by performing approximate pattern matching. This methodology has the ability to control the degree of function approximation by adjusting the false positive rate of the ReBFs. To reduce energy consumption of the proposed architecture, low-power memristive arrays are exploited to perform search operations at extremely low energy. Our experimental results show 47.5% energy reductions of multiplication is obtained with classification accuracy degradation at 1% for convolutional neural networks. Experimental results show that for image processing applications, our approach represents on average 38.42% of the functionality of FUs in five different kernels running on GPUs, while guaranteeing the acceptable outputs with PSNR of greater than 27 dB. This leads to on average 24.1% energy reduction compared to conventional architectures without ReBF.

Chapter 2 contains re-organized reprints of Vahideh Akhlaghi, Abbas Rahimi, and Rajesh Gupta, "Resistive Bloom Filters: from Approximate Membership to Approximate Computing with Bounded Errors", *In IEEE Design, Automation, and Test in Europe (DATE)*, 2016, of which this dissertation author is the primary author, and Xun Jiao, Vahideh Akhlaghi, Yu Jiang, and Rajesh Gupta, "Energy-Efficient Neural Networks using Approximate Computation Reuse", *In IEEE Design, Automation, and Test in Europe (DATE)*, 2018, of which this dissertation author is the primary investigator.

Chapter 3

Dynamic Network Pruning

Deep Convolutional Neural Networks (CNNs) perform billions of operations for classifying a single input. To reduce these computations, this chapter 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 Ṫhis 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.

3.1 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 [RDS+15] requires billions of multiply-and-accumulate operations [CES16, SPM+16, HMD16]. This work 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 3.1 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 3.2 illustrates this insight across two images passing through GoogLeNet [SLJ+15]. The highlighted differences in the output of the intermediate convolutional layer attest to the varying spatial

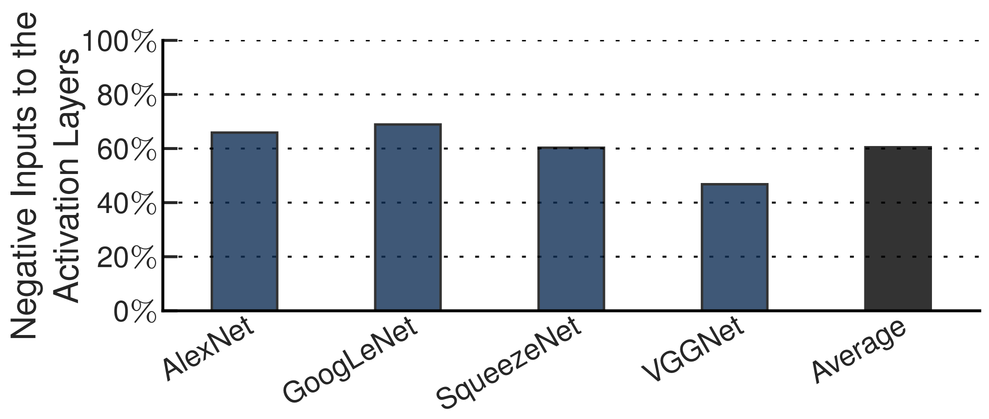


Figure 3.1: Fraction of activation input values that are negative.

distribution of zeros. Harnessing these insights, we devise SnaPEA[1], a holistic software-hardware solution, that cuts a large fraction of the computations short by identifying the zero intermediate values earlier during the runtime.

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

- 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

[1]**SnaPEA**: **Sna**ppy **P**redictive **E**arly **A**ctivation

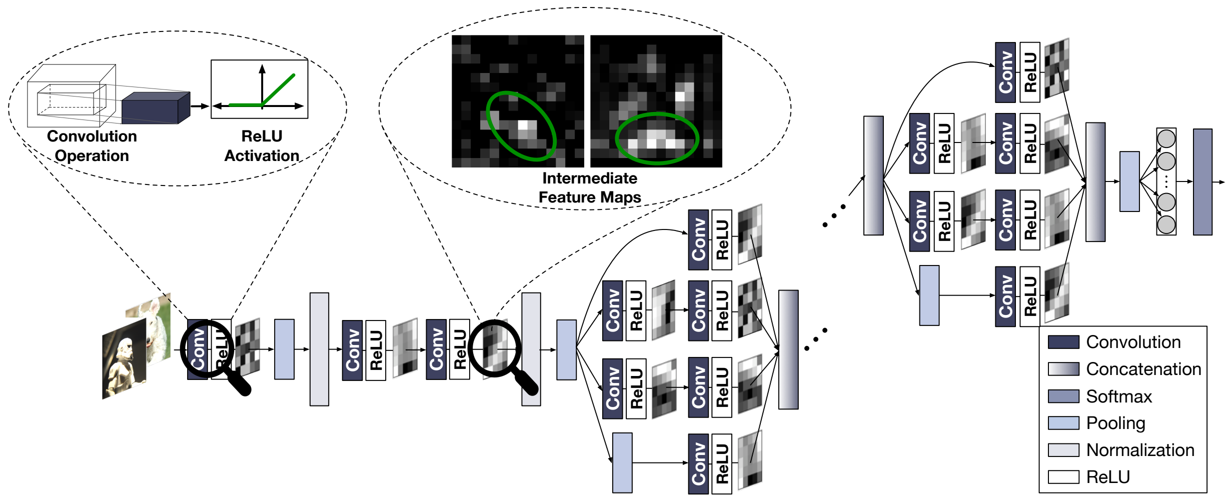


Figure 3.2: GoogLeNet [SLJ+15], in which the intermediate feature maps for two input images are magnified. The ellipses on the intermediate feature maps highlight the varying spatial distribution of non-zero values for distinct input images.

by using a combination of runtime information along with the algorithmic structure of CNNs, which feeds many negative inputs to the activation function.

- SnaPEA is a runtime technique that cuts the CNN computations short. Exploiting the aforementioned insight, this chapter 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.

- 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 [CES16], 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 [IHM+16]. 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.

3.2 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 CNNs has been proposed [GPY+17, PRM+17, CES16, JAH+16, RWA+16, ZDZ+16, HLM+16, AJH+16, DFC+15, LCL+15, ZLS+15, CDS+14b, FMC+11]. In some of the most recent works [CES16, DFC+15, FMC+11], 2D spatial architectures have been proposed to match with the convolution dataflow and maximize the data reuse. TETRIS [GPY+17] and Neurocube [KKC+16] 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 [MHP$^+$17, ALPBP17, HZS17, HMD16, IHM$^+$16] proposed various static pruning techniques to reduce the overhead of computation in deep convolutional neural networks. These static pruning techniques are agnostic to the dynamically-generated zeros whose locations in the activation layer vary from one image to another. As our results show, SnaPEA is complementary to these techniques and further improve the benefits over the static pruning techniques. Furthermore, several architectures also have been proposed [PRM$^+$17, AJH$^+$16, RWA$^+$16, ZDZ$^+$16, HLM$^+$16] for exploiting the sparsity in the input activations and/or weights to improve the efficiency of the accelerator. In one of the most recent work, SCNN [PRM$^+$17] 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

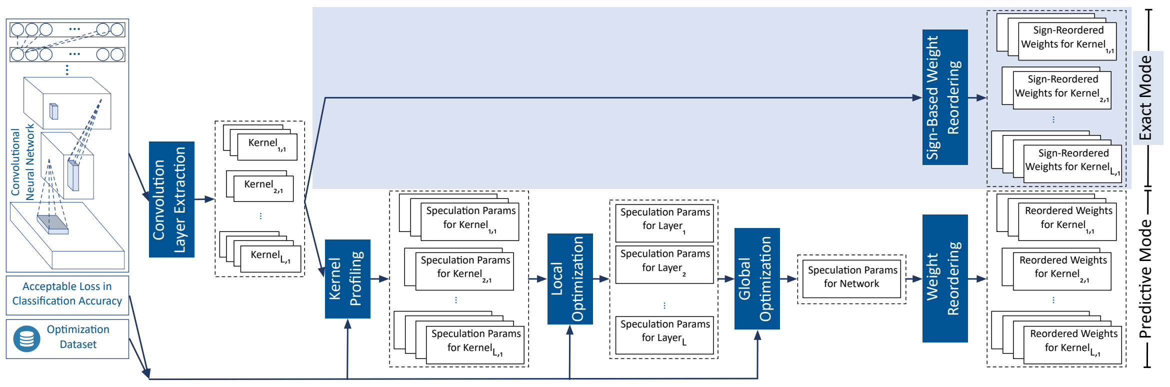
A handful of research efforts [YPT$^+$15, MRR11, SDMHR11, MSHR10, HMS$^+$09, Rin07, Rin06, LSKS17] have looked into task skipping in various domains. In one of the most recent efforts [SDMHR11], Sidiroglou et al. proposed loop perforation in which the accuracy is traded in return for improvement in performance. In their proposal, they algorithmically transform the critical loops in the program and *only* execute a subset of their iterations. PredictiveNet [LSKS17] 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.

3.3 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 3.3, 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.

3.3.1 SnaPEA Software Workflow

Figure 3.3 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.



Convolutional Neural Network
Acceptable Loss in Classification Accuracy
Optimization Dataset
Convolution Layer Extraction
Kernel 1,1
Kernel 2,1
Kernel L,1
Kernel Profiling
Speculation Params for Kernel 1,1
Speculation Params for Kernel 2,1
Speculation Params for Kernel L,1
Local Optimization
Speculation Params for Layer 1
Speculation Params for Layer 2
Speculation Params for Layer L
Global Optimization
Speculation Params for Network
Weight Reordering
Reordered Weights for Kernel 1,1
Reordered Weights for Kernel 2,1
Reordered Weights for Kernel L,1
Sign-Based Weight Reordering
Sign-Reordered Weights for Kernel 1,1
Sign-Reordered Weights for Kernel 2,1
Sign-Reordered Weights for Kernel L,1
Exact Mode
Predictive Mode

Figure 3.3: Software workflow for SnaPEA.

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 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 3.5 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.

3.3.2 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 3.6 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 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 3.6).

3.4 Computation Reduction in SnaPEA

Figure 3.5 demonstrates how SnaPEA reduces the computation by an example of 1×3 convolution. Figure 3.4a performs the unaltered convolution in which all of the MAC operations are performed and yields “-9” as the output. Figure 3.4b 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.

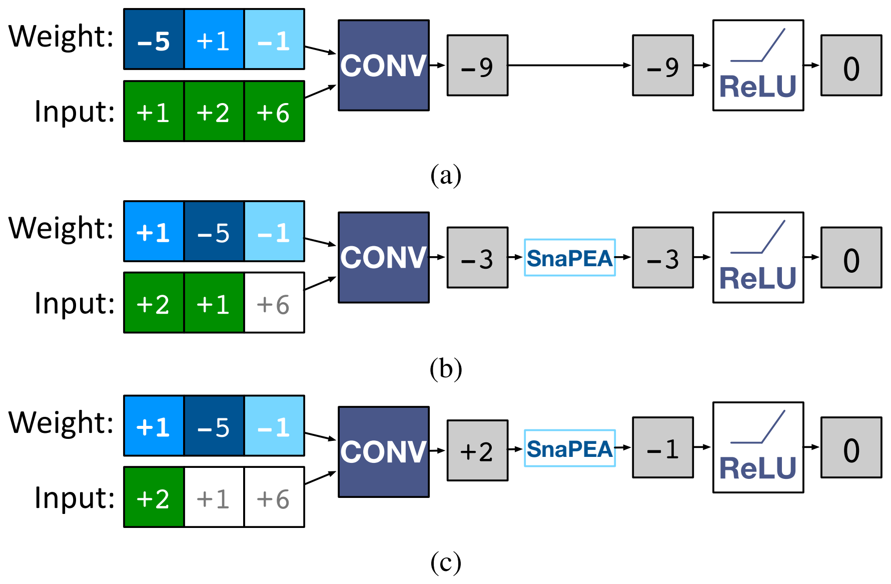


Figure 3.4: A 1×3 convolution in (a) unaltered (b) exact, and (c) predictive modes. In the latter two, the weights and their corresponding inputs are reordered. The white boxes highlight the operations that are cut.

Figure 3.4c 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 3.5a. Using these methods, SnaPEA can forgo a significant number of the MAC operations as illustrated in 3.5b.

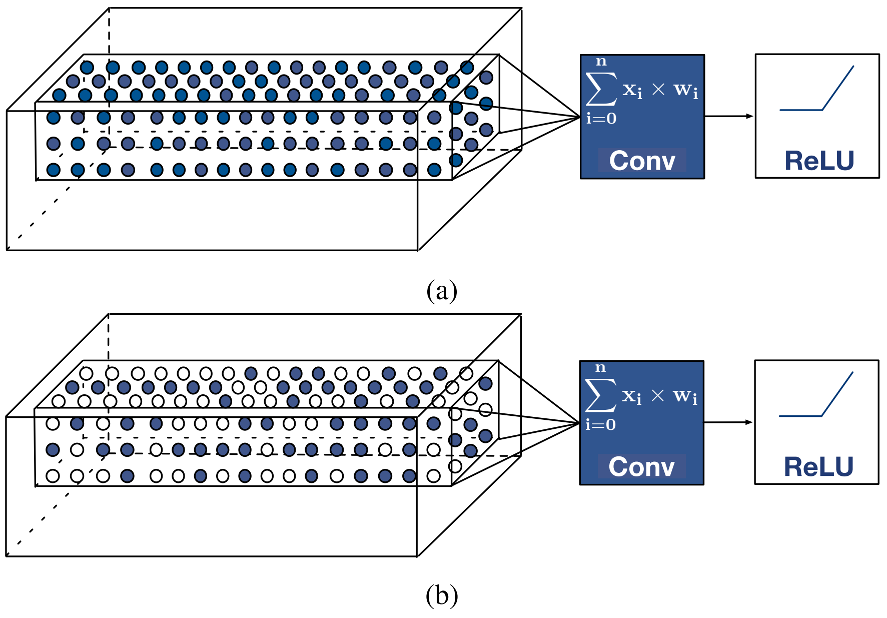

$\sum_{i=0}^{n} x_i \times w_i$
Conv
ReLU
(a)
$\sum_{i=0}^{n} x_i \times w_i$
Conv
ReLU
(b)

Figure 3.5: (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.

3.5 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.

3.5.1 Speculation Parameters

As mentioned in Section 3.3.1, 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 $(\mathrm{Th}, \mathrm{N})$, in which Th is a list of threshold values and N is a list of the number of groups for selecting the corresponding operations.

3.5.2 Problem Formulation

The problem of finding the speculation parameters (i.e., $(\mathrm{Th}, \mathrm{N})$) 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 (3.1). 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 (3.1) 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 3.1), 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_l^k is the kernel width.

$$\mathrm{Op}(o_{l,k}^d, \mathrm{Th}_l^k, \mathrm{N}_l^k) = \begin{cases} \mathrm{N}_l^k, & \text{if } \mathrm{PartialSum}_{\mathrm{N}_l^k} \leq \mathrm{Th}_l^k, \\ \mathrm{Idx}_{w^-}, & \text{if } \mathrm{PartialSum}_{\mathrm{N}_l^k} > \mathrm{Th}_l^k \,\&\, \mathrm{PartialSum}_{w^-} \leq 0, \\ \mathrm{C}_{in,l} \times \mathrm{D}_l^k \times \mathrm{D}_l^k, & \text{otherwise} \end{cases} \tag{3.1}$$

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, $\mathcal{D}$ an optimization dataset, ε an acceptable accuracy loss, Th_l^k and N_l^k the speculation parameters of kernel k of layer l, $\mathrm{O}_{l,k}^d$ the outputs of the convolution generated by kernel k in layer l for the input image d from $\mathcal{D}$, and $\mathrm{Accuracy}_{CNN}$ and $\mathrm{Accuracy}_{SnaPEA}$ the classification accuracy of the CNN and the classification accuracy obtained by SnaPEA, respectively. Now, $(\mathrm{Th}, \mathrm{N})$ can be determined by solving the following problem:

$$\begin{aligned} &\min_{\mathrm{Th},\mathrm{N}} \quad \sum_{d \in \mathcal{D}} \sum_{l \in L} \sum_{k \in K_l} \sum_{o \in O_{l,k}^d} \mathrm{Op}(o, \mathrm{Th}_l^k, \mathrm{N}_l^k) \\ &\text{subject to} \quad \mathrm{Accuracy}_{CNN} - \mathrm{Accuracy}_{SnaPEA} \leq \varepsilon \end{aligned} \tag{3.2}$$

3.5.3 Finding the Speculation Parameters

In order to solve the optimization problem formulated as (3.2), we devise a greedy algorithm (i.e., Algorithm 1). The algorithm takes a CNN, an optimization dataset $\mathcal{D}$, 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

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

```
 1: Inputs:     CNN: a CNN model, $\mathcal{D}$: an optimization dataset,
                $\varepsilon$: Acceptable loss in classification accuracy
 2:
 3: Outputs:    ParamCNN: Speculation parameters (Th,N) for the CNN
 4: function KernelProfilingPass(CNN, $\mathcal{D}$, $\varepsilon$)
 5:     Initialize ParamK[l][k]$\rightarrow \emptyset$
 6:     for $\forall$ layer $l$ in CNN do
 7:         for $\forall$ kernel $k$ in layer $l$ do
 8:             for a set of values (th,n) do
 9:                 op, err = Simulate(CNN, $\mathcal{D}$, k, th, n)
10:                 if err$\leq \varepsilon$ then
11:                     ParamK[l][k].append((th,n,op))
12:             Sort ParamK[l][k] based on op
13:     return ParamK
14: function LocalOptimizationPass(CNN, $\mathcal{D}$, $\varepsilon$, ParamK)
15:     for layer l in CNN do
16:         for t in range(0,T) do
17:             for k in layer l do
18:                 param = ParamK[l][k][t]
19:             op, err = Simulate(CNN, $\mathcal{D}$, $\varepsilon$,param)
20:             if err $\leq \varepsilon$ then
21:                 ParamL[l].append((param,op,err))
22:     return ParamL
23: function AdjustParam(CNN,ParamCNN,ParamL)
24:     for $\forall$ layer $l$ in CNN do
25:         for $\forall$ t in range(len(ParamL[$l$])) do
26:             meritL$[l][t] = \frac{\text{-(ParamL}[l]\text{[2]-ParamCNN}[l]\text{[2])}}{\text{(ParamL}[l]\text{[1]} - \text{ParamCNN}[l]\text{[1])}}$
27:     l,t = Argmax(meritL)
28:     return (l,t)
29: function GlobalOptimizationPass(CNN,$\mathcal{D}$, $\varepsilon$,ParamL)
30:     for $\forall$ layer $l$  in CNN do ParamCNN[l] = ParamL[l][0]
31:     err = Simulate(CNN,$\mathcal{D}$,ParamCNN)
32:     while err$> \varepsilon$ do
33:         l,t=AdjustParam(CNN,ParamCNN,ParamL)
34:         ParamCNN[l] = ParamL[l][t]
35:         remove ParamL[$l$][$t$] from ParamL[$l$]
36:         err = Simulate(CNN, $\mathcal{D}$, $\varepsilon$,ParamCNN)
37:     return ParamCNN
38: Initialize ParamCNN[l]$\rightarrow \emptyset$
39: ParamK = KernelProfilingPass(CNN, $\mathcal{D}$, $\varepsilon$)
40: ParamL = LocalOptimizationPass(CNN, $\mathcal{D}$, $\varepsilon$,ParamK)
41: ParamCNN =GlobalOptimizationPass(CNN, $\mathcal{D}$, $\varepsilon$,ParamL)
```

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.

3.6 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 [CES16, PRM+17, GPY+17, AJH+16, CDS+14b, SPM+16]. 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 [CES16, GPY+17, SPM+16]. 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 3.6 (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 3.6 (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

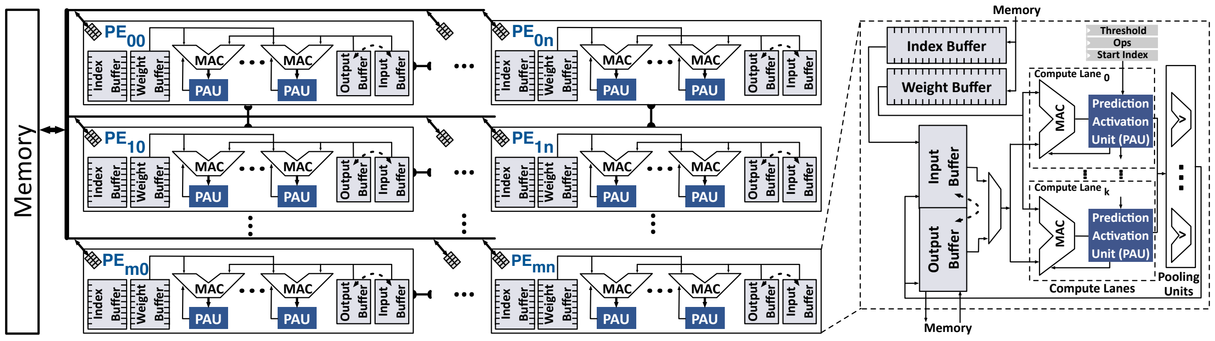


Figure 3.6: (a) The overall structure of the SnaPEA architecture and its multilevel memory hierarchy, containing an off-chip memory and a distributed on-chip buffer for input and outputs. (b) The micro-architecture of each PE. The weights are shared across the compute lanes.

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 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 predetermined order (See Section 3.5). 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 3.7 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 3.7, we evaluate the effect of increasing computation lanes on the idle cycles and their effects on the performance and energy savings.

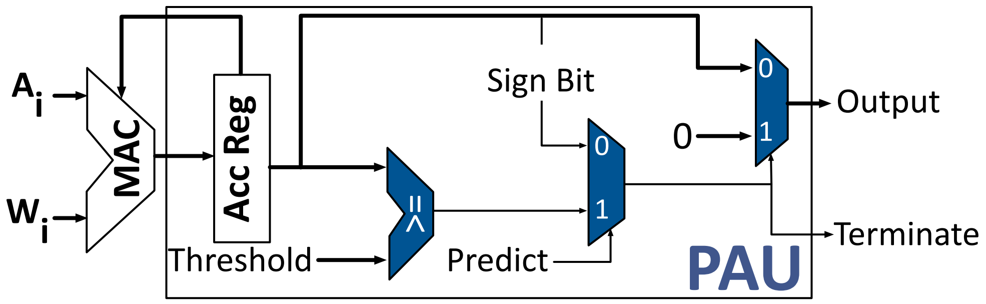


Figure 3.7: Prediction Activation Unit (PAU). The Predict signal determines the PAU operation mode (exact or predictive). The Terminate signal, once asserted, terminates the computation early.

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 3.12, 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 3.7.2 by analyzing the sensitivity of performance to the number of compute lanes per each PE.

3.7 Evaluation

3.7.1 Methodology

We use several popular medium to large scale dense CNN workloads. We also include SqueezeNet [IHM+16] 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 3.1 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 [RDS+15] validation dataset. We use Caffe v1.0 [JSD+14] to run the pre-trained networks on a GPU. We compile Caffe using NVCC v8.0.62 and 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.

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 [RDS+15] 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.

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

Table 3.1: 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 3.2: SnaPEA and EYERISS [CES16] 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	

architecture and obtain the area, delay, and energy numbers of the logic hardware units.

In this work, 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 3.2 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 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 [CES16] accelerator. Table 3.2 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 3.2 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 3.2). This increase in the area is mainly attributed to the added predictive activation units (PAUs) in the PEs and the controllers.

Table 3.3 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 [GPY+17], 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 3.3). 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 [micb] to estimate the energy cost of accesses to the off-chip memory.

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 3.6 into the simulator in a cycle-level manner. To measure the energy savings, we use the synthesis results

Table 3.3: 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).

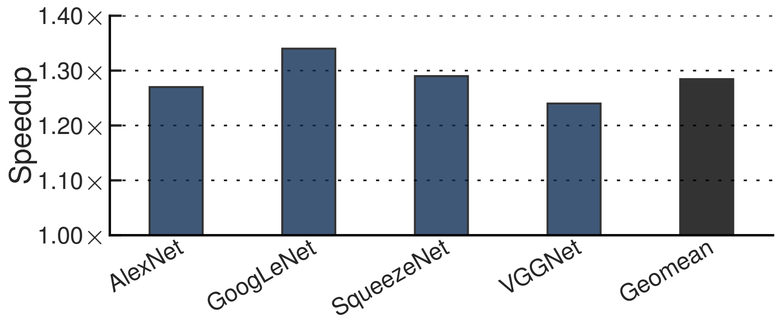
Operation	Energy (pJ/Bit)	Relative Cost
Register File Access	0.20	1.0
16-bit Fixed Point PE	0.30	1.5
Inter-PE Communication	0.40	2.0
Global Buffer Access	1.20	6.0
DDR4 Memory Access	15.00	75.0

and the reported energy numbers from some of the recent works [GPY+17, CES16, Gal12]. Furthermore, we use CACTI-P [LCA+11] 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, 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.

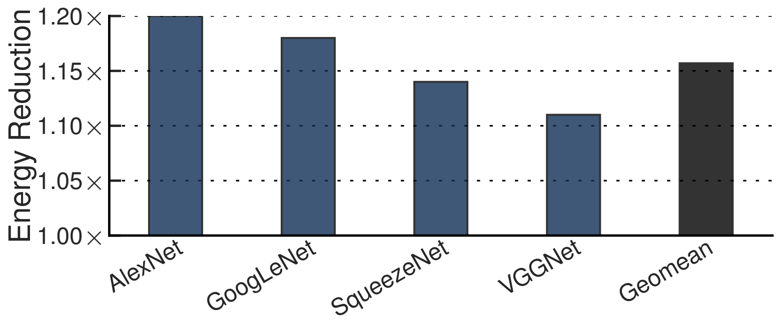
3.7.2 Experimental Results

Overall Benefits in the Exact Mode

Figure 3.8 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 3.6). 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 [IHM+16]—a statically pruned convolutional neural network—SnaPEA yields 1.3×



(a)



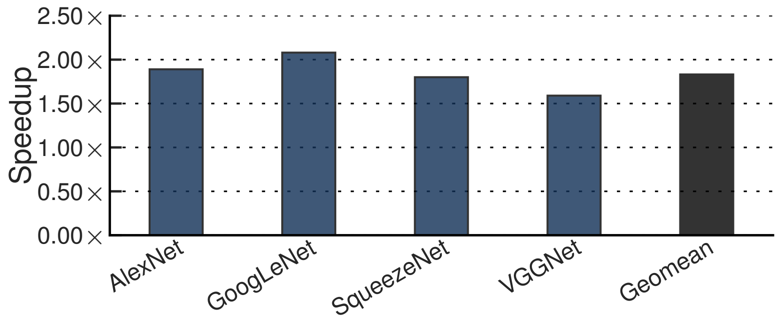
(b)

Figure 3.8: Overall (a) speedup and (b) energy reduction with exact mode.

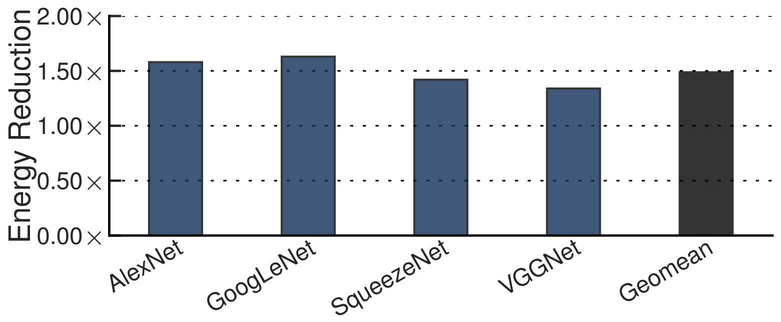
and 1.14×. These savings for SqueezeNet show that static pruning techniques are complimentary to the dynamic approach of SnaPEA. Overall, the results in the exact mode show the practicality of SnaPEA in delivering speedup and energy reductions even in the pure exact mode, in which the CNN classification accuracy remains untampered (Table 3.1).

Overall Benefits in Predictive Mode

Figure 3.9a 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 3.1). 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



(a)



(b)

Figure 3.9: Overall (a) speedup and (b) energy reduction with SnaPEA over EYERISS [CES16] in the predictive mode. The acceptable classification accuracy drop is maintained within ≤3% range of its baseline value.

are negative, and hence the saving is larger.

Figure 3.9b illustrates the energy reduction with SnaPEA in predictive mode over EYERISS [CES16]. 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 [IHM+16], a statically pruned CNN model, our 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 [IHM+16], in exploiting the runtime information to provide significant savings.

Figure 3.10 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

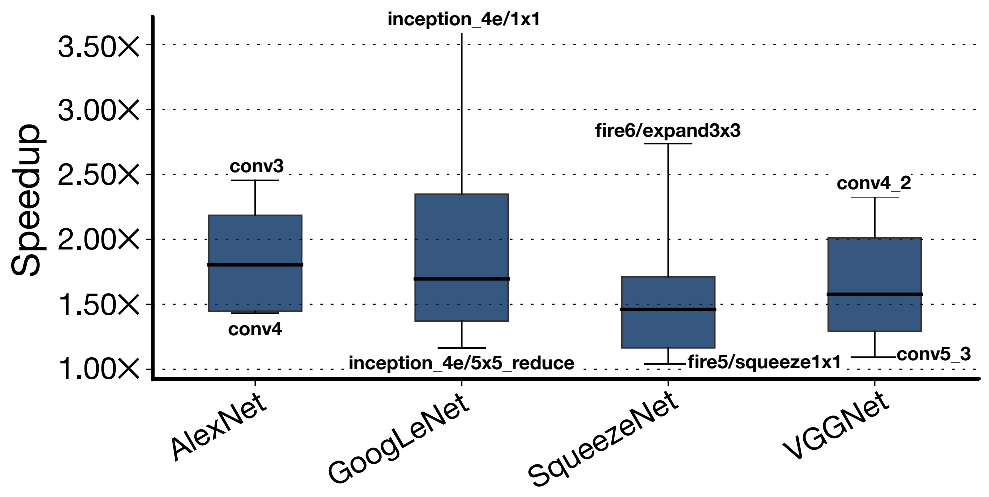


Figure 3.10: Speedup of convolutional layers in each network for the predictive mode when the degradation in classification accuracy is set to $\leq 3\%$.

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 3.4 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 3.5 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

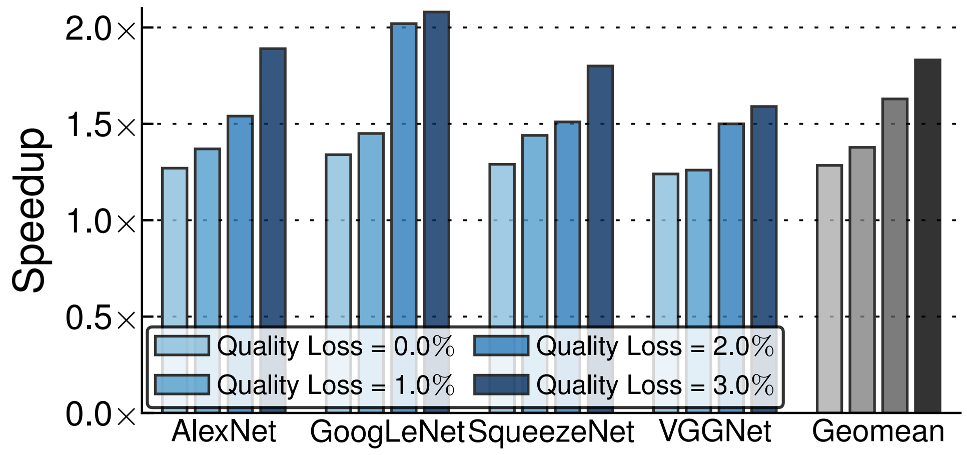


Figure 3.11: Speedup vs. loss in the CNN classification accuracy. Each bar indicates the speedup when the acceptable degradation in the classification accuracy is 0% (pure exact mode), 1% (predictive mode), 2.0% (predictive mode), and 3.0% (predictive mode), respectively.

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.

Sensitivity to the Degree of Speculation

To study the effect of our proposed predictive early activation technique, Figure 3.11 illustrates the speedup with SnaPEA over EYERISS [CES16] 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%,

Table 3.4: The percentage of convolution layers that operates in the predictive mode, when classification accuracy drop is set to $\leq 3\%$. 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×

Table 3.5: 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%

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 3.12 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) shows

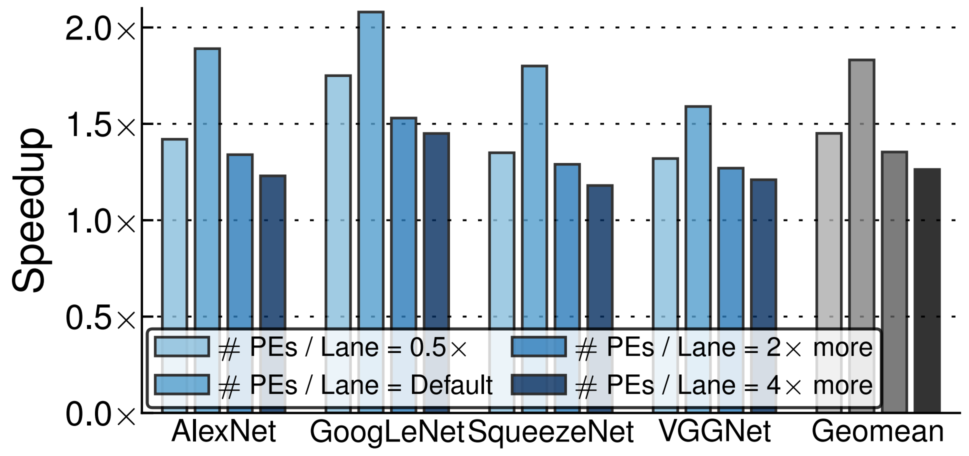


Figure 3.12: Sensitivity of speedup with SnaPEA over EYERISS to the number of compute lanes per each PEs. Each bar indicates the speedup when the number of compute lanes per each PEs is altered by different factors (acceptable classification accuracy drop $\leq 3\%$).

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 3.6, 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 [CES16], 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.

3.8 Summary of the Chapter

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$\times$ and 1.89$\times$, 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.

Chapter 3, is a reprint of the material appeared in Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh Gupta, and Hadi Esmaeilzadeh, "Snapea: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks", *in Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018.

Chapter 4

Platform-Aware Algorithm-Hardware Approximation

To improve the execution costs of many applications, approximate computing is one of the promising solution that exploits the intrinsic error-tolerance of applications to trade accuracy of computation for improved performance or/and energy efficiency. Sometimes, depending on the application error tolerability, a combination of various approximate methods at both algorithm and hardware level helps to maximize the related efficiency. However, an optimal approximation that maximizes the efficiency under a given user-specified output quality is hard to achieve due to several reasons such as large design space and limitations imposed by the underlying hardware platforms. In addition, taking the underlying hardware architecture into account for selecting approximate designs and/or modifying the underlying hardware architecture to comply with the approximate configurations are necessary for maximum efficiency. Therefore, a systematic way to find a proper configurations for the algorithm and hardware related approximate methods given a set of constraints imposed by both model and hardware along with selecting an optimal hardware architecture design is required. In this chapter, we provide a novel machine learning based strategy for fast design space exploration, proper selection of approximate configurations considering the model and platform related constraints, as well as selection of proper hardware architecture related parameters to maximize the efficiency. In particular, we model the impact of approximate methods as a function of approximate design parameters, modify CNNs computation as a function of approximate design parameters, model the execution cost as a function of approximate design, hardware architecture and platform related parameters, formulate the problem of finding an optimal approximate and architecture settings under a given set of quality and platform related constraints coming from the underlying CNN model and platform as a constrained optimization problem and solve the problem with stochastic gradient descent (SGD).

4.1 Introduction

Many applications and algorithms such as multimedia, machine learning, and signal processing admit some level of tolerance to the reduced precision of computations. For example, the accuracy of deep learning algorithms are not impacted by some level of imprecise computation due to several factors such as over-parameterization [NBA$^+$18]. To improve their execution costs, approximate execution of these algorithms by reducing computation precision at the algorithm or hardware level can be deployed. At the algorithm level, approximate versions of these algorithms with reduced operations, parameters and data precision, which are more efficient alternatives in terms of the computation complexity, can replace the original ones. At the hardware level, approximate hardware units that are low cost but less accurate can be employed to execute such applications and algorithms approximately with low cost. For example, reducing the numerical precision of data (i.e., quantization) in CNNs [MSC$^+$16, WH19, LMC$^+$16, RORF16, KWW$^+$17] at the algorithm level and using approximate arithmetic units (e.g., adders [KK12] and multipliers [VKAKP17]) and approximate memory units with lower operating voltages and timing constraints [CGW$^+$14, LKP$^+$15, LPMZ11, LJVM12] at the hardware level show improvements in the execution cost of aforementioned applications.

Despite the proven system-level benefits of these techniques in the literature, there is still a gap between the actual benefits of these techniques in practice and what has been explored and designed so far. To bridge this gap and allow approximate computing techniques to realize their full potential in improving DNNs execution time and energy consumption, several challenges and opportunities are required to be addressed and explored.

The first challenge is in controlling the output quality. The efficiency of approximate methods must be balanced against the degradation of output quality in a measurable way. Therefore, for a given CNN model, it is necessary to find a proper configuration for the underlying approximate methods (e.g., the number of bits to represent data in quantization or the voltage/timing parameters of approximate memories) such that its execution cost (i.e., energy and latency) is minimized while the final output quality remains acceptable.

The second challenge is that the approximate methods are application and input dependent [BC10, ZPL14]. Exploiting the opportunities offered by the approximate computing requires selection of proper approximate methods for a given application and a given input dataset. Further, the current approaches use a single approximate method [NHKL20, LMC$^+$16] with uniform conservative configuration for all the target instructions across the entire program, which limits the potential gains and calls into question the value of approximate computing. Practical and useful means for approximate computing must consider the impact of simultaneous use of multiple approximate methods at both hardware and algorithm levels with non-uniform configurations on output quality.

Finally, the underlying platform and their related hardware architectures are the key factors that are required to be considered in selecting non-uniform configurations of approximate methods used throughout the model execution and/or optimized to support the non-uniform configurations and maximize the benefits. The following examples show the ineffectiveness of current methods in finding optimal configurations because of neglecting the impact of underlying platform in their approaches. EDEN [KOY$^+$19], in which different settings for approximate memories are determined for accessing and storing the input and output feature maps of different layers in CNNs, ignores the impact of pipelining the execution of a group of layers on some reconfigurable and smaller sizes of platforms; therefore, in such platforms where feature maps of some layers are not moved to DRAM units, the selection of approximate configurations for those layers are useless and proper approximate configurations will not be achieved due to the impractical tolerable error estimation for all layers. As another example, HAQ [WLL$^+$19] learns various bit-width for various layers of a CNN through reinforcement learning, and then, the learned bit-width for all layers is adjusted manually to meet the constraints on the hardware resources. However, the final bit-width suggestion may not be an optimal one due to ignoring the limitation of hardware directly in the optimization process.

Based on the mentioned challenges, the key approach to maximize the efficiency of ML accelerators is non-uniform platform aware approximation at the algorithm and hardware level as well as architectural level optimizations. To reach this point, a simple and systematic methodology is required for a fast exploration of a large design space, a proper selection of non-uniform approximate configurations for the model and hardware related approximate methods and a proper design choice for the hardware architecture such that the computation cost of a CNN model is maximized while the algorithm related constraints (e.g., acceptable classification accuracy) and the hardware related constraints (e.g., using limited number of hardware resources) will be satisfied. To find a proper approximate configurations and an efficient architectural design systematically, we introduce a machine learning-based optimization framework, which (1) integrates the impact of each approximate methods at both the algorithm and hardware level into a given CNN model considering the underlying architectural support, (2) formulates the problem of minimizing the computation cost under given model and hardware related constraints into an optimization problem and (3) learns the proper configurations and architectural design choice through stochastic gradient descent procedure.

4.2 Related Work

To expand the applicability of deep neural networks and enable them to run on edge devices, various types of designs and optimizations are explored and studied on both DNNs algorithms and computing architectures. One of the main approaches to improve their performance on the hardware is approximate computing. DNN models are intrinsically tolerant, to some extent, to the reduced precision in their computation. This characteristic pave the way for reducing the precision of computation in order to improve the execution costs. There are several approximate techniques at the algorithm or hardware level employed to accelerate deep learning algorithms. Quantization [MSC+16] and using approximate DRAM units [KOY+19] are examples of such approximate computing techniques, at the algorithm and the hardware level, respectively. Despite their effectiveness in reducing the cost, most of the existing methods only employ one source of approximation or lack of a systematic optimization approach to maximize their benefits. In this work, we show that optimal algorithm-hardware approximation with non-uniform configurations throughout the program enhances the benefit of approximate computing. Also, considering the hardware limitation and optimizing architectural parameters can further improve the cost and practicality of these approaches.

Quantization Techniques

One of the main approximate methods at the algorithm level that significantly reduces the cost of arithmetic operations, memory accesses, and data transfers is using narrower bit-width for presenting data in DNN models. [SCYE17] provides a review of set of quantization techniques in DNNs. One of the common quantization methods used in DNN accelerators is representing data in fixed-point format [MSC+16, CES16]. Despite the significant benefits of quantization, the main challenge is to determine the number of bits to represent data without significant impact on the final accuracy. For fixed-point values, for example, as well as the total number of bits, specifying the number of bits to represent the fractional parts and the number of bits for the integer parts without changing the accuracy is challenging. [MSC+16] shows that using 8 bits for the weights and 10 bits for feature maps can maintain the accuracy within 1% of the original accuracy. Therefore, selecting a proper bit-width configuration is necessary for keeping the accuracy high.

Approximate DRAM Units

To reduce the energy consumption of DRAM accesses, the main approach is to reduce the supply voltage, thus reducing the refresh rate and relaxing their timing constrains such as activation latency and pre-charge latency [CGW+14, LKP+15, LJVM12, CYG+17, LPMZ11]. However, these relaxed settings introduce errors during read and write operations [CYG+17]. Therefore, to control the accuracy proper DRAM parameters settings are required. The authors in [KOY+19] provide a framework to enable safe use of approximate DRAM units in DNNs by modeling the error of a set of DRAM chips and controlling their voltage reduction based on the tolerance of DNN models.

Determining Approximate Configurations

As discussed above, although approximate computing techniques reduce the computation cost, the main challenge is a proper choice of level of approximation to control their impact on the final model accuracy. Most of the research in the area of approximation computing for reducing computation costs, especially in DNNs, determines the proper approximate configurations with a heuristic search by increasing the approximation level gradually until a safe configuration is found [KOY+19]. In [LLH+16], a statistical quality model is proposed in which an error model is first extracted from a given set of approximate functional units for fast exploration, and then their impacts on the output quality are explored by error propagation techniques. Although [LLH+16] offers an error model for approximate computing for fast design space exploration, the method is not scalable and limited to a specific set of approximate functional units with uniform configurations, which inhibits maximum gain. Despite the effectiveness of these techniques in finding a safe configuration, a major challenge appears when the design space becomes large due to various levels of approximations, various approximation methods at each level, and non-uniform approximation of a large numbers of computation blocks in the algorithms, which makes the search exhaustive. HAQ [WLL$^+$19] deploys reinforcement learning to find the proper bit-width for different layers of CNNs. However, this approach requires complex formulation of the problem to use reinforcement learning; thereby, not only finding the optimal configurations becomes highly dependent on the proper formulation and definition of the reward function, but in a setting with different approximate methods, using HAQ is not straightforward. To overcome these challenges, our proposed scalable framework uses a simple formulation of the problem and solves it based on a machine learning technique to explore various configurations fast and learns proper configurations in different approximate settings.

4.3 Overview of the Framework

In this section, the overall process to find the proper approximate setting for algorithm and hardware related approximate methods based on the underlying platform and architecture along with optimizing architecture to support such approximation efficiently will be explained in details. The first step is to model the impact of approximate methods into the computation of a given DNN models as a function of a set of approximation related parameters. The second step is to determine the model and hardware related constraints and formulate them as a function of a set of parameters. The parameters required for formulating the constraints can be a combination of the approximate related parameters defined in the previous step and a set of new parameters that are related to the underlying platform, architecture and the DNN models. The third step is to formulate the main problem of finding proper configurations in the form of minimizing an objective function under a set of constraints formulated before. The objective function represents the execution cost. Finally, the proper configurations can be determined by finding a solution to the defined problem through stochastic gradient descent (SGD).

4.3.1 Parameterized Error Injection

In order to integrate the impact of the approximation methods used to improve model computation and the cost of hardware units on a given DNN model's output, we need to obtain a proper model that imitates their true error behaviors. Here, we explain several methods that can be categorized into two groups. Depending on the approximate method and the available information about how it impacts the computation, one of the methods is selected to inject the error of approximation into the DNN computation.

Model-Based Error Injection

One of the most precise methods to inject the error of an approximate method is to find a mathematical model to represent the actual impact of a given method (e.g., quantization) and modify the corresponding computation part of the DNN model accordingly. To explain this method in more detail, let's consider quantization as an example of the approximate methods to reduce the computation burden of CNNs. Suppose we quantize a value represented as a floating-point number (V_{float}) to a fixed-point number (V_{fixed}) with m bits for the integer (int) and q bits for the fractional part ($fraction$). Assuming that m is fixed for different quantization levels and large enough to represent the integer part of the value, with the fractional part represented with q bits, the new quantized value of V_{float} with the mentioned fixed-point format can be obtained by the following model:

$$V_{fixed} = \frac{round(V_{float} \times 2^q)}{2^q} \tag{4.1}$$

With the model formulated in 4.1, the quantization error can simply be injected through quantizing the values directly in a given computation. Here, in this formula, the quantized value is represented as a function of parameter q that controls the level of quantization. In this work, we call the parameter q as the error parameter related to quantization that can be considered as a controlling knob for the quantization.

Despite the true representation of the error behavior offered by this model-based method, one of the main challenges is lack of such explicit models for some of the approximate methods.

Statistics-Based Error Injection

For the approximate methods with no explicit model to inject their errors into a given computation, statistical analysis are used to estimate and model their error behaviors. In the statistical-based error modeling, the distribution of the error is mainly computed and injected into the computation. However, since finding a true error distribution is not feasible, the expected value of the impacted computation is estimated based on the available and easily-computing error metrics related to the approximate methods. Then, the estimated expected values are used to replace the corresponding computation part in a DNN model so that the impact of approximation is injected and propagated to the output.

As an example of the error metrics that can be used for error estimation is the error rate, which is the probability of occurrence of the error related to an approximate method. The impact of the error expressed with the error rate is modeled with the following formula:

$$\tilde{X} = \alpha \times E + (1 - \alpha) \times X \tag{4.2}$$

where X and $\tilde{X}$ are the value before and after being exposed to the approximation method, respectively; E is the error value as a result of the approximation method when error occurs and α is the probability of the occurrence of the error.

Another prevailing error metric to express the behavior of approximate methods, especially in the approximate hardware units, is the average of relative error. For these cases, the impact of the average of relative error ω on a value X is formulated in 4.3.

$$\tilde{X} = X \times (1 + \omega) \tag{4.3}$$

In equations 4.2 and 4.3, the parameters α and ω are the error parameters that can be used as the controlling knobs for controlling the degree of approximation in the underlying approximate methods.

With these two error injection methods, the error of approximation is integrated into the model computation and propagated to the model output. Therefore, the output of the model impacted by a set of approximate methods ($\tilde{O}$) can be formulated as equation 4.4 that is a function of its inputs (X), the model related parameters ($\mathcal{W}$), i.e., the parameters of the original model such as convolutional filters in a CNN, and the error parameters ($\mathcal{E}$), i.e., the parameters related to controlling the approximation degree of the approximate methods used to approximate model's algorithms and/or hardware units.

$$\tilde{O} = f(X, \mathcal{W}, \mathcal{E}) \tag{4.4}$$

4.3.2 Model and Hardware Related Constraints

Using approximate alternatives of computation in DNN models or approximate units at the hardware level decreases the execution costs of these models; on the other hand, it may adversely impact the quality of the models outputs. For example, in CNNs, approximate methods degrade their classification accuracies to an unacceptable level if the degree of approximation is not controlled properly. Therefore, improvement in execution costs is required to be balanced against the degradation on the models output quality, which necessitates a proper selection of approximate methods such that the execution cost will be minimized while the output quality remains within the user-specified acceptable range. In addition, to maximize the cost efficiency requires that the approximate alternatives of computation in DNN model are executed efficiently on the hardware. To ensure the hardware efficiency of approximate models, hardware level information is required to decide which alternative to be selected. The hardware level information could be the underlying hardware architecture and the related dataflow mechanism, the number of possible arithmetic units implemented on the chip and the number of available on-chip buffers.

Based on the aforementioned issues, due to the necessity of maintaining a balance between the cost improvement and model output quality and taking the hardware level constraints into consideration in selecting a cost-efficient approximate implementation of a DNN model, we require to include model and hardware related constraints into the process of learning optimal algorithm-hardware approximation. Accordingly, we formulate these constraints as below. The output quality related constraints can be specified as:

$$\mathrm{Q}(\tilde{O}) \geq C_{out} \tag{4.5}$$

where Q is a function that specifies the model output quality which depends on the type of model and its application (e.g., for CNNs, Q is the classification accuracy), and C_{out} is the lower bound on the output quality.

The main hardware related constraint is the number of available hardware resources (e.g., number of Look Up Tables on FPGA) which varies across different platforms. In addition to the model and hardware related approximate methods and their configurations that determine the required number of hardware resources, the underlying hardware architecture (e.g., the number of parallel MAC units on an ML accelerator) is also important. To show the impact of the architecture, we define a new set of architecture related parameters ($\mathcal{A}$) representing the underlying architecture design that along with the error parameters ($\mathcal{E}$) help to formulate the hardware related constraints. Accordingly, the following formula determines the hardware related constraints:

$$\mathrm{R}_{\mathcal{H}}(\mathcal{E}, \mathcal{A}) \leq C_{\mathcal{H}} \tag{4.6}$$

where $R_{\mathcal{H}}$ is a function that determines the required hardware resources $\mathcal{H}$ of the underlying hardware platform to implement a given approximate version of computation, and $C_{\mathcal{H}}$ is the maximum number of available hardware resources $\mathcal{H}$ on a given platform.

4.3.3 Problem Formulation

Providing an optimal non-uniform algorithm-hardware approximation requires to select a proper approximate configuration for the model algorithm and hardware units as well as a proper hardware architecture design such that the computation cost is minimized while the model and hardware related constraints are satisfied. The approximate configurations that inject higher error to the output is expected to offer lower computation cost. However, as mentioned before, the lower cost must be balanced against the output quality. On the other hand, given a limited number of resources on the underlying platform, for non-uniform selection of approximate configurations, the underlying hardware architecture is required to be taken into account and designed properly to maximize the efficiency. Accordingly, the problem of providing optimal platform-aware algorithm-hardware approximation can be formulated as the following optimization problem:

Problem 4.1. Find a set of error parameters $\mathcal{E}$ and architecture parameters $\mathcal{A}$ such that:

$$\begin{gathered}\min_{\mathcal{E},\mathcal{A}} f_c(X,\mathcal{W},\mathcal{E},\mathcal{A}) \\ s.t. \quad \mathrm{Q}(\tilde{O}) \geq C_{out} \quad where \quad \tilde{O} = f(X,\mathcal{W},\mathcal{E}) \\ \mathrm{R}_{\mathcal{H}}(\mathcal{E},\mathcal{A}) \leq C_{\mathcal{H}}\end{gathered}$$

where f_c is a function that determines the computation cost, which depends on the input and model parameters as well as the error parameters controlling the degree of approximation and architecture parameters determining a suitable architecture for this design.

To find an optimal platform-aware approximate configurations for computation and hardware units, in the next step, we require to find a solution, i.e., finding a proper values of $\mathcal{E}$ and $\mathcal{A}$, for Problem 4.1.

4.3.4 Learning Optimal Algorithm-Hardware Level Approximation and Architecture Design

In order to find a solution for Problem 4.1., we use a well-known technique, stochastic gradient descent, employed in machine learning algorithms to find the model parameters that minimize a loss function. Stochastic gradient descent is useful for finding an optimal solution for an unconstrained optimization problem; thus, for solving Problem 4.1, which is a constrained optimization problem, it is required to be converted to an unconstrained one. We convert Problem 4.1 to an unconstrained optimization problem using Penalty and Barrier method [Die13]. Penalty and Barrier method defines a continuous function called Penalty function, the value of which increases to infinity for the points approaching the boundary of feasible regions. The penalty function defined by this method will be integrated into the objective function of the problem so that any violation of the constraints will be penalized significantly, which leads the solver toward finding solutions to satisfy the constraints.

For Problem 4.1., the penalty functions for the constraints can be defined as follows:

$$\text{Penalty(Q)} = \lambda_{\text{Q}}(\text{Q}(\tilde{O}) - C_{out}) \tag{4.7}$$

$$\text{Penalty(R)} = \lambda_{\text{R}}(\text{R}_{\mathcal{H}}(\mathcal{E}, \mathcal{A}) - C_{\mathcal{H}}) \tag{4.8}$$

where λ_{Q} and λ_{R} are large positive numbers that penalize violation of the constraints defined in 4.5 and 4.6 and used in Problem 4.1. with a large factor.

By integrating the penalty functions defined in equations 4.7 and 4.8 into Problem 4.1., the problem of providing optimal platform-aware algorithm-hardware approximation can be formulated as 4.9 and be solved with the SGD.

$$\begin{aligned} \min_{\mathcal{E},\mathcal{A}} \quad & f_c(X,\mathcal{W},\mathcal{E},\mathcal{A}) + \lambda_{\mathrm{Q}}(\mathrm{Q}(\tilde{O}) - C_{out}) + \lambda_{\mathrm{R}}(\mathrm{R}_{\mathcal{H}}(\mathcal{E},\mathcal{A}) - C_{\mathcal{H}}) \\ & \textit{where} \quad \tilde{O} = f(X,\mathcal{W},\mathcal{E}) \end{aligned} \tag{4.9}$$

Now, stochastic gradient descent can find an optimal set of parameters, i.e., $\tilde{\mathcal{E}}$ and $\tilde{\mathcal{A}}$, for Equation 4.9 by iteratively updating the parameters using the gradient of the objective function with respect to each parameter at training data points. Finally, the optimal set of parameters $\tilde{\mathcal{E}}$ learned in 4.9 determines proper configurations of approximate methods for the algorithm and hardware units and $\tilde{\mathcal{A}}$ determines proper architecture design that reduce the cost while satisfying the conditions on output quality and hardware resources.

4.4 Optimal Algorithm-Hardware Approximation for FPGA-based CNN accelerators

In this section, we explain how the process of providing optimal platform-aware algorithm-hardware approximation can be used to optimize CNN algorithms on FPGA-based accelerators. In particular, we consider quantization as an approximate method to reduce the burden of model computation and DRAM voltage scaling as an approximate method to reduce the cost of hardware units. As mentioned before, to maximize energy saving of CNNs with acceptable accuracy, finding proper bit-widths for quantizing the weights and feature maps of each convolutional layer given the underlying resource limitations, proper voltage levels for storing and accessing weights and feature maps of each layer in DRAM, and a proper hardware architecture setting is required.

Before defining the optimization problem to find such configurations, let's look at an example of FPGA-based accelerator architecture shown in Figure 4.1. The accelerator architecture consists of several parts: an interface to DRAM unit to buffer data coming from or written back

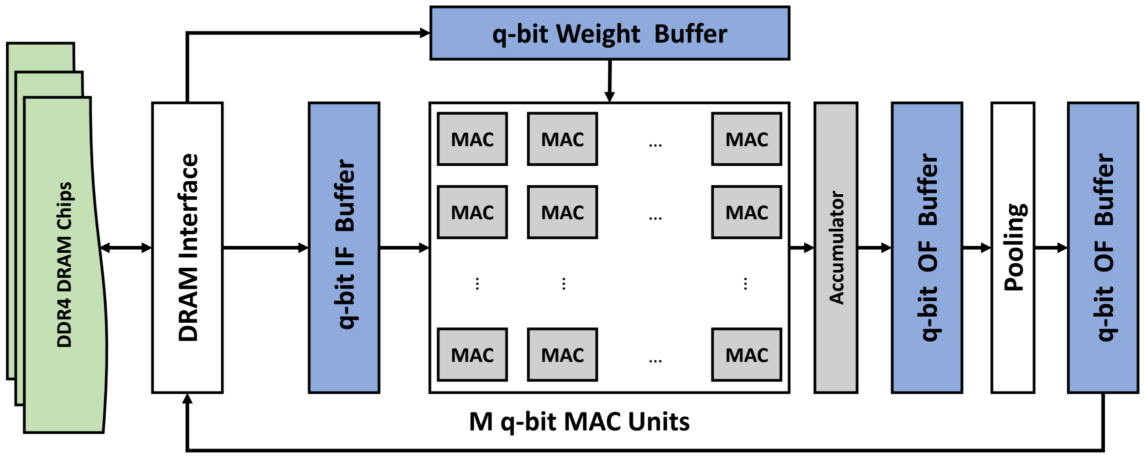


Figure 4.1: Overview of FPGA-based accelerator architecture with uniform quantization and DRAM voltage for all layers. M q-bits MAC units are used in parallel to execute convolutional layers. IF/OF maps buffers and weight buffers all store q-bits values. All the chips and DRAM partitions are working with the same voltage.

to memory, a set of buffers to store weights (Weight Buffer) and input feature maps (IF Buffer) and output feature maps (OF Buffer), a set of M MAC units to execute convolutional layers by parallelizing their operations and a pooling unit to execute pooling layers. In this architecture, all the units are designed to execute a CNN, the feature maps and weight of which are quantized to q bits. Therefore, all the MAC units execute q-bit operations and buffers store q-bit values. In addition, all the data related to each layer is stored in and accessed through a DRAM where all the partitions and DRAM chips work under a fixed operating voltage. Therefore, this architecture supports uniform approximate configurations.

Now, to use different bit-width for different layers based on their eror tolerance to save more energy, we require an architecture to support such non-uniformity of the operation precision across various layers. Such architecture, as shown in Figure 4.2, needs a dedicated set of resources for different groups of layers (the layers in a group have similar quantization setting), which impacts its compute efficiency due to limited resources on FPGA platforms. As an example, the architecture in Figure 4.2 has *d* separate set of resources to support the execution of *d* groups of convolutional and pooling layers, meaning that layers weights and feature maps can be

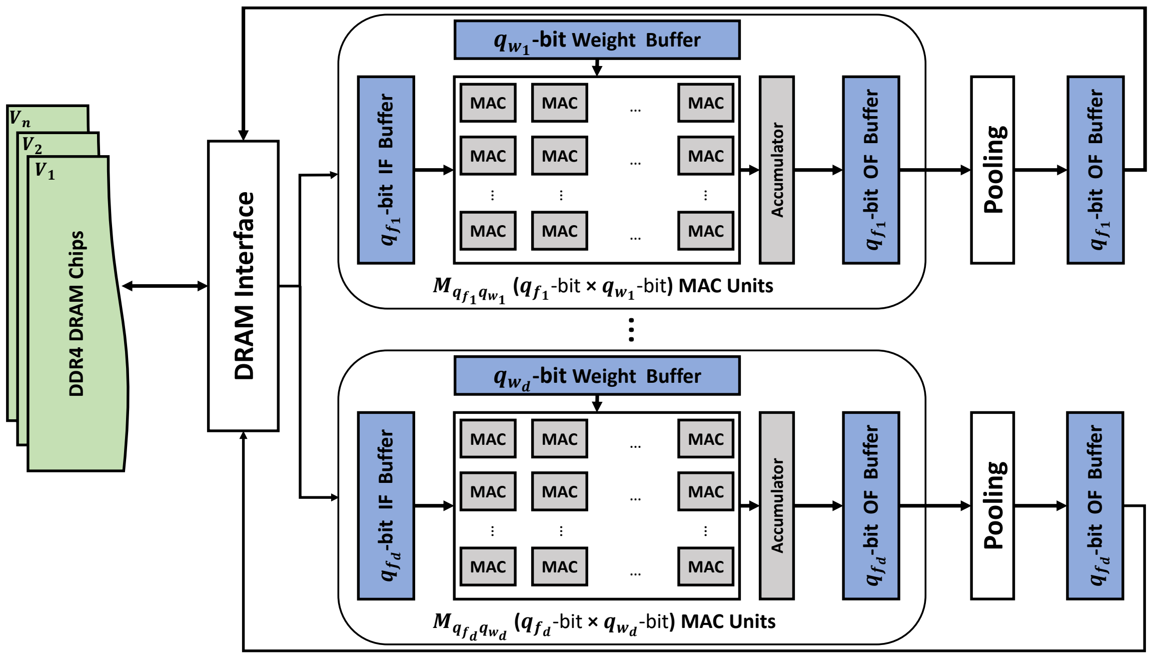


Figure 4.2: CNN accelerator architecture to support non-uniform quantization and approximate main memory error

quantized to one of q_{w_1} and q_{f_1} bits to q_{w_d} and q_{f_d} bits configurations. Dedication of various set of resources on FPGA to each group of convolutional layers limits parallelization of operations in each layer depending on the size of FPGA mainly due to resource under-utilization for each layer. The resource under-utilization leads to deployment of limited number of MAC units to parallelize operations of a layer at each time while the rest of the MAC units and resources are idle. Therefore, to maximize the energy efficiency, it is required to control this heterogeneity across layers (i.e., selecting a proper value for d) and make a proper decision on the number of dedicated MAC units for each group g (i.e., selecting a proper value for $M_{q_{fg}q_{w_g}}$) based on the number of available resources on FPGA, as well as a proper number of bits for feature maps and the weight of the layers (i.e., selecting a proper value for a set of parameters $\{q_{fi}q_{w_i} \forall i \in \{1,2,...,d\}\}$ and assigning a proper bit-width configuration from the set to a layer depending on its tolerance to the quantization).

In the followings, we elaborate on the process of defining the optimization problem and learning the bit-width and DRAM voltages for each layer and proper number of MAC units for each group.

4.4.1 Injecting Quantization and DRAM Voltage Scaling Error into CNNs

Here, we explain the process of injecting error of quantization and DRAM voltage scaling into CNNs and define a set of related parameters that will be used later to control the degree of approximation in these methods.

The impact of quantizing the weights and feature maps can be directly modeled based on the quantization method. As mentioned in Section 4.3.1, for fixed point quantization method, the formula in 4.1 can be used. With a proper formula, for any given quantization method, the weights and feature maps in each layer of a CNN model can be modified directly throughout the entire model based on the number of bits represented as a quantization parameter assigned for the weights and feature maps of the layer. Therefore, for the entire CNN model, to inject the impact of quantization, a set of quantization parameters is defined as follows:

$$Q = \{q_f^l | \forall l \in L\} \cup \{q_w^l | \forall l \in L\}, \tag{4.10}$$

where L is a set of all convolutional layers in a CNN model, q_f^l is the number of bits to quantize input and output feature maps of layer l and q_w^l is the number of bits for the model weights.

For injecting the impact of voltage scaling of DRAM into CNN models, we use Bit Error Rate (BER), which is the number of bits that are flipped due to scaling the operating voltage of memories or reducing the activation/pre-charge latency [CGW+14, LKP+15, CYG+17]. Thus, BER can be interpreted as the probability of a bit flip (p). With this information obtained from the approximate DRAM, the probability of having error p_{err} on an n-bit value is:

$$p_{err} = 1 - (1 - p)^n. \tag{4.11}$$

For injecting the impact of DRAM, the statistics-based error injection shown in Equation 4.2 can be used. Assuming that the bit flips are randomly distributed across different bits, the value of E can be considered as a random number. Therefore, here, we consider BER represented by p as the error parameter corresponding to approximate DRAM. Accordingly, different parameters of p can be defined for weights and IFM/OFMs in different layers of a CNN model as follows:

$$\mathcal{P} = \{p_f^l | \forall l \in L\} \cup \{p_w^l | \forall l \in L\}, \tag{4.12}$$

where L is a set of all convolutional layers in a CNN model, p_f^l corresponds to BER for the feature maps of layer l and p_w^l corresponds to BER for the weights in that layer.

4.4.2 Constraints on Model Accuracy and FPGA Resource Budget

The next step is to specify a set of model and hardware related constraints similar to those mentioned in Section 4.3.2. For determining the model related constraints, the main factor for quality measurement in CNNs is classification accuracy. Therefore, in 4.5, Q is the classification accuracy of the CNN impacted by quantization and DRAM voltage scaling. In addition, in computer vision tasks, classification accuracy within 1% of the original classification accuracy is acceptable. Thus, C_{out} in 4.5 which is the maximum acceptable quality degradation is equal to the original accuracy reduced by 1%. Therefore, the model related constraints in CNNs can be formulated as follows:

$$\begin{gathered} a\tilde{c}c \geq acc - 1 \\ a\tilde{c}c = \tilde{f}(X, \mathcal{W}, \mathcal{Q}, \mathcal{P}) \quad \& \quad acc = f(X, \mathcal{W}) \end{gathered} \tag{4.13}$$

where $a\tilde{c}c$ is the accuracy of the CNN impacted by quantization and DRAM voltage scaling which is a function of CNN input, weights, and error parameters $\mathcal{Q}$ and $\mathcal{P}$, and acc is the accuracy of the original CNN as a function of CNN input and weights.

The main hardware related constraints for CNNs running on FPGA platforms are the limited FPGA resources mainly the limited number of Look Up Tables (LUT), Flip-Flops (FF), Block BRAMs (BRAM) and Digital Signal Processors (DSP) which varies across various platforms. As mentioned earlier in this section, non-uniform quantization of the weights and feature maps in different layers of CNNs, although, in theory, reduces the latency compared to uniform quantization of layers due to selecting the highest quantization level that can be tolerated by each individual layer, in practice on hardware platforms such as FPGAs due to the limited number of resources and the necessity of allocating separate resources to perform MAC operations with different bit-widths, it suffers from resource under-utilization of FPGA resources, which impacts the execution costs. Therefore, proper selection of bit-width along with proper selection of the number of MAC units for each group of convolutional layers such that the energy consumption is minimized while the number of required resources for such configuration does not exceed the number of available resources on FPGAs is necessary.

To formulate the hardware related constraint, we require to formulate the number of required resources for a given non-uniform bit-width configuration of layers. Here, let's assume that the required number of bits are less than or equal to 8 bits. Since in synthesis tools such as Vivado HLS tool, only LUTs and FFs are used to implement MAC operations between values having less than or equal to 8 bits, we only require to compute the number of required LUTs (T) and FFs (F).

The number of LUTs (T_l) for each layer l can be computed directly by the number of MAC units that are dedicated for the layer to parallelize the operations in the convolutions.

To estimate the number of LUTs and FFs to implement M Mac units on FPGA, we obtain a model based on an FPGA synthesis of a $3x3$ convolutional layer with different bit-width settings and the number of MAC units. The experiments show that for all bit-width configurations, there is a linear relation between the number of LUTs and FFs for different number of MAC units (power-2 multiples of 16 units) and the number of LUTs and FFs required for 16 MAC units with similar bit-width ($T^{\mathsf{Base}}_{q^l_w q^l_f}$) as the baseline configuration. Therefore, for a given bit-width configuration $q_w q_f$ and its corresponding number of MAC units $M_{q_w q_f}$, the number of LUTs ($T^{\text{conv}}_{q_w q_f}$) and FFs ($F^{\text{conv}}_{q_w q_f}$) can be computed with the following formula.

$$\begin{aligned} T^{\text{conv}}_{M_{q_w q_f}} &= \alpha_{M_{q_w q_f}} T^{\text{Base}}_{q_w q_f} \\ F^{\text{conv}}_{M_{q_w q_f}} &= \beta_{M_{q_w q_f}} F^{\text{Base}}_{q_w q_f} \end{aligned} \tag{4.14}$$

Therefore, the architecture related parameters are:

$$\mathcal{A} = \{M_{q_w q_f} | \forall q_w q_f \in \cup_{l \in L} \{q^l_w q^l_f\}\} \tag{4.15}$$

4.4.3 Problem Formulation

Now, based on the error parameters defined in Section 4.4.1 and model and hardware related constraints and architecture parameters defined in Section 4.4.2, the problem of finding proper platform-aware approximate configurations along with proper architecture design parameters to minimize latency under a given resource budget while maintaining an acceptable accuracy is formulated as the following:

Problem 4.2. For a given CNN with L convolutional layers, convolutional parameters $\mathcal{W}$, and input image X, and a given FPGA platform with $\mathcal{R}_{LUT}$ LUTs and $\mathcal{R}_{FF}$ FFs, find a set of error parameters $\mathcal{E} = \{\mathcal{Q} \cup \mathcal{P}\}$ and architecture parameters $\mathcal{A} = \{M_{q_w q_f} | \forall q_w q_f \in \cup_{l \in L} \{q^l_w q^l_f\}\}$ such that:

$$\begin{aligned} &\min_{\mathcal{E},\mathcal{A}} \quad (\sum_l \frac{OP_l}{M_{q^l_w q^l_f}}) \\ s.t. \quad & \tilde{acc} \geq acc - 1 \quad where \quad \tilde{acc} = \tilde{f}(X, \mathcal{W}, \mathcal{Q}, \mathcal{P}) \quad \& \quad acc = f(X, \mathcal{W}) \\ & \sum_{M_{q_w q_f}} T^{\text{conv}}_{M_{q_w q_f}} \leq \mathcal{R}_{LUT} \quad where \quad T^{\text{conv}}_{M_{q_w q_f}} = \alpha_{M_{q_w q_f}} T^{\text{Base}}_{q_w q_f} \\ & \sum_{M_{q_w q_f}} F^{\text{conv}}_{M_{q_w q_f}} \leq \mathcal{R}_{FF} \quad where \quad F^{\text{conv}}_{M_{q_w q_f}} = \beta_{M_{q_w q_f}} F^{\text{Base}}_{q_w q_f} \end{aligned}$$

In Problem 4.2. $\frac{OP_l}{M_{q_w^l q_f^l}}$ estimates the latency of layer l.

4.4.4 Optimal Quantization and DRAM Voltage

To learn proper number of bits for the quantizing different layers and proper voltage of DRAM to store feature maps and weights along with proper number of MAC units for implementing non-uniform quantization given a set of resource constraints, we use SGD to solve Problem 4.2. Before applying SGD, the problem is required to be reformulated and represented in the form of unconstrained optimization problem. Using the Penalty and Barrier method [Die13] as described in Section 4.3.4 the problem will be in the following form.

$$\min_{\mathcal{E},\mathcal{A}} \quad (\sum_l \frac{OP_l}{M_{q_w^l q_f^l}}) + \lambda_{acc}(\tilde{acc} - (acc-1)) + \lambda_{LUT}(\sum_{M_{q_w q_f}} T^{\text{conv}}_{M_{q_w q_f}} - \mathcal{R}_{LUT}) + \lambda_{FF}(\sum_{M_{q_w q_f}} F^{\text{conv}}_{M_{q_w q_f}} - \mathcal{R}_{FF})$$
$$where \quad T^{\text{conv}}_{M_{q_w q_f}} = \alpha_{M_{q_w q_f}} T^{\text{Base}}_{q_w q_f} \quad \& \quad F^{\text{conv}}_{M_{q_w q_f}} = \beta_{M_{q_w q_f}} F^{\text{Base}}_{q_w q_f} \tag{4.16}$$

4.5 Experimental Evaluation

In this section, we evaluate our proposed framework in finding the proper model and approximate hardware configurations to minimize hardware cost and accuracy drop. Through different sets of experiments, we show the generality of the proposed framework to find the proper approximate configurations of different combinations of approximate methods for different machine learning models under various model and hardware related constraints.

4.5.1 Optimal Hardware Approximation for the Weight Updates in Linear Regression and SVM

We use the proposed process for efficient execution of two machine learning training algorithms (i.e., two gradient descent algorithms used for updating the weights in Linear Regression and Support Vector Machine (SVM) [MPA$^+$16]) on the hardware with approximate methods only at the hardware level. Here, we assume that the underlying hardware platform has three types of configurable approximate hardware units (i.e., Adders, Multipliers and DRAM units). The goal is to find the best approximate configurations for these units so that the energy consumption is minimized while the average of relative errors of the weight updates at each iteration remain within a specific range. To find the best configuration for this problem, we customize the process explained in Section 4.3 and call the framework LEMAX.

We consider an approximate platform that contains an approximate DRAM, and a set of approximate multiplier units and adders which are proposed in [LPMZ11],[AKAKP17], [VKAKP17], respectively. Accordingly, the specification of these units including their energy-error model and the error metric, is extracted and summarized in Table 4.1.

DRAM- Relaxing data preserving schemes in DRAM has been proposed earlier to improve its energy consumption. As an approximate DRAM, we chose Flikker [LPMZ11] in which the refresh rate is reduced. The error metric is error rate in the applications input data. Therefore, we use Equation 4.2 to integrate the effect of approximate DRAM on the output quality and energy consumption. Since Flikker uses the same refresh rate for all pages, we use the same α for all the inputs. Moreover, the corresponding approximate erroneous value (E) is chosen uniformly at random. The refresh energy of baseline DRAM is extracted from [mica]. In the experiments, the energy of refreshing DRAM in only one DRAM operating cycle is accounted in the cost function.

Adder- We chose RAP-CLA [AKAKP17], a configurable approximate carry look-ahead adder where an external correction unit for the exact add operation is omitted. The error metric of average of relative error is chosen to define the error parameters and Equation 4.3 to inject the corresponding error into the computation. The energy-error model is extracted from [AKAKP17] which are obtained in 45 nm technology.

Multiplier- For the approximate multiplier, we use AQ-LETAM [VKAKP17] which is composed of approximate multiplication, shift and add operations on the truncated inputs. The accuracy of

Table 4.1: Energy-error of approximate units extracted from [LPMZ11, AKAKP17, VKAKP17]

Adder	**Avg. Rel. Err**	0	0.004	0.056	0.12	-
	Energy (aJ)	5789	1906	1815	1145	-
Multiplier	**Avg. Rel. Err**	0	72E-5	24E-4	8E-3	24E-3
	Energy (fJ)	4920	1000	930	740	590
DRAM	**Error Rate**	0	1E-9	1E-8	1E-7	1E-6
	Energy (nJ)	132.84	120.8	110.2	102.6	99.6

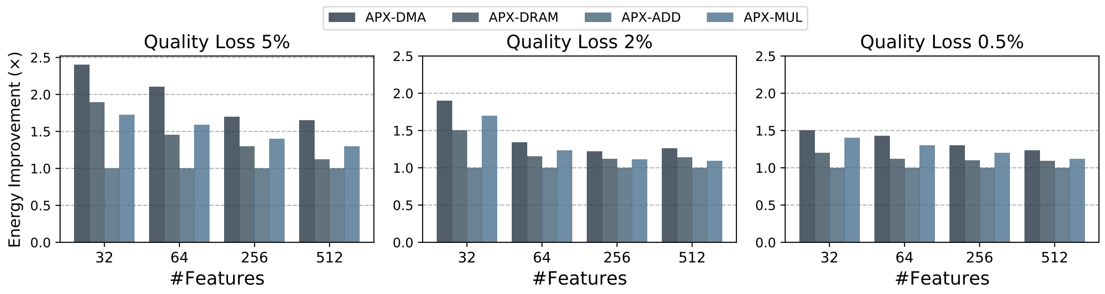


Figure 4.3: Energy consumption improvement of different approximate systems compared to the baseline in executing Linear Regression training algorithm with different number of features under different quality constraints.

AQ-LETAM is adjusted by selecting the proper truncation length. The chosen error metric and the process of error injection is similar to the case of adders. The reported energy numbers for different configurations were obtained using 45 nm technology.

We evaluate LEMAX effectiveness on four approximate systems with different approximate modules. APX-DMA comprises of approximate DRAM, approximate multipliers and approximate adders. APX-DRAM only employs approximate DRAM to improve the energy consumption of a system while APX-MUL and APX-ADD only use approximate multipliers and adders, respectively. To determine the best configurations of approximate units in each system for the corresponding instruction throughout the program, we require to define and solve a problem similar to the one defined in Equation 4.9. Here, in this case, we assume that the area budget is not limited or impacted significantly by different approximate arithmetic units, therefore, the term related to resource constraints in Equation 4.9 can be removed. To define the

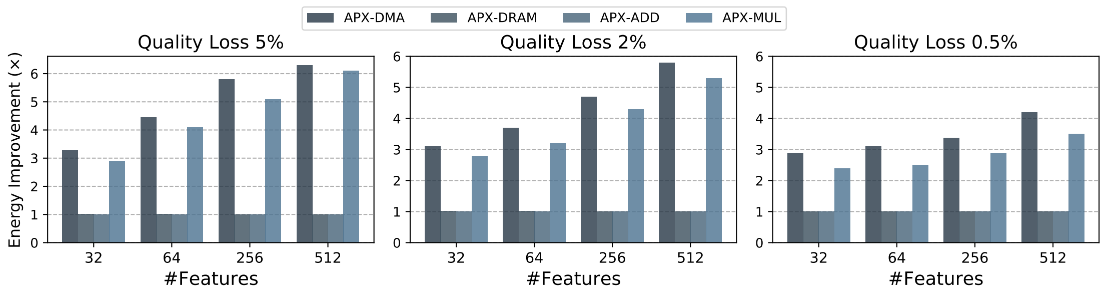


Figure 4.4: Energy consumption improvement of different approximate systems compared to the baseline in executing SVM training algorithm with different number of features under different quality constraints.

objective function, the corresponding part of the program (i.e., weight update loop that includes computing the gradient descent algorithm in LR and SVM) is implemented in Python and the related error of the approximate unit in each system is injected into the program by defining a set of error parameters $\mathcal{E}$. Accordingly, the quality loss $Q(\tilde{O})$ is defined as a function of error parameters by traversing the original and parameterized approximate version of the program. Moreover, the cost function f_c is defined based on the energy-error model obtained from the hardware specifications (Table 4.1) and the operations in the program. In order to learn the error parameters to minimize the modified objective function defined in Problem 4.9, we use Adam optimizer [KB14]. As such, we exploit Autograd (version v1.2)[aut], which enables the gradient descent based optimization by providing the gradient of a function at a given point with respect to the parameters. The obtained gradient of the objective function is then fed into the Adam optimizer, implemented by the Autograd developers, to learn the error parameters.

To learn the error parameters, we use 6000 data samples from UCI machine learning repository [DG17] as the training set and 1500 samples as the validation set. In order to perform mini-batch gradient descent, the training set is divided into 15 batches, each of which has 400 samples. To evaluate the scalability of LEMAX in learning the error parameters of more inputs/operations in larger programs, we change the size of the applications by varying the number of features used in the data sets. We evaluated LEMAX for the chosen applications with the input sets consisting of 32, 64, 256, and 512 features. Accordingly, Table 4.2 shows the number of

Table 4.2: Number of error parameters, adders and multipliers, and the quality loss on the validation dataset for Linear Regression and Support Vector Machine learning algorithms with different number of input features

Application	#Features	Error Parameters	# Adders	# Multipliers	Tested Quality Loss (%)		
					5%	2%	0.5%
LR	**32**	161	64	96	5.2	2.0	0.52
LR	**64**	321	128	192	5.1	2.07	0.49
LR	**256**	1281	512	768	4.98	1.9	0.47
LR	**512**	1024	1536	2269	5.1	2.1	0.49
SVM	**32**	226	95	130	4.8	2.04	0.49
SVM	**64**	450	191	258	4.9	2.1	0.49
SVM	**256**	1794	767	1026	4.9	2.03	0.48
SVM	**512**	3586	1535	2050	5.1	1.8	0.5

error parameters, and the number of adders and multipliers. As the output quality metric, i.e., function Q, in this work, we chose average of relative error and evaluate LEMAX under different quality losses of 5%, 2% and 0.5%. LEMAX can be used for any other quality metrics, as well.

To evaluate training time of LEMAX for finding the error parameters for Linear Regression and SVM with various quality constraints, we use the Python package timeit. Results show that LEMAX converges relatively fast (i.e., on average, 354 for smaller sizes of applications (32 and 64 features) and 1594 for larger applications). We also tested quality loss of the applications on the validation data set shown in Table 4.2. The results show that LEMAX can successfully achieve a configuration that meets the desired quality even for the validation data.

To estimate the energy consumption of applications in different approximate systems defined above, the refresh energy of DRAM, energy of adders and energy of multipliers are summed up together. The energy numbers for each module is dependent on the respective error parameter learned by LEMAX. Since the goal of this work is to show the effectiveness of simultaneous non-uniform use of multiple approximate units, in this case, we only consider the energy of refresh in approximate DRAM, and energy of adders and energy of multipliers, and exclude the effect of other modules in estimating the energy consumption. Figure 4.3 and 4.4 show the energy improvement of the Linear Regression and SVM training algorithms, respectively, in four mentioned approximate systems compared to the baseline case where all the units operate in the exact mode. The evaluation is performed for different quality losses and different number of input features. The energy is calculated based on the error parameters obtained by LEMAX.

In Linear Regression, use of multiple approximate units achieves energy improvement of 96.2%, 43.3% and 36.4%, on average, compared to the exact system for quality loss of 5%, 2%, and 0.5%, respectively, while only a single approximate unit improves the energy by 31.47%, 17.1%, and 12.75%. In addition, the results show that the energy improvement is lower for the larger circuits (e.g., 65% for quality loss 5% in APX-DMA for 512 features) compared to the smaller one with less number of input features (e.g., 2.4× for quality loss 5% in APX-DMA for 32 features). The happens because of similar impact of all features on the output quality in Linear Regression algorithm, which limits the degree of approximation.

In SVM, APX-DMA decreases the energy consumption by 4.96×, 4.32× and 3.39×, on average, compared to the exact system for quality loss of 5%, 2%, and 0.5%, respectively. In approximating a single unit, only APX-MUL results in high energy reduction compared to the exact system (i.e., 4.54×, 3.5× and 2.8× for quality loss of 5%, 2%, and 0.5%, respectively). In the other systems, i.e., APX-DRAM and APX-ADD, the improvement is negligible (0.2%). Although APX-MUL improves energy consumption significantly, using approximate multipliers along with approximate DRAM and adders is still a better design option achieving, on average 41.7% additional improvement. We note that because of the training algorithm of SVM in which the weight parameters are updated only for the mis-classified data samples, and the consequential effect of a small set of features on the classification accuracy, the tolerance of SVM to approximation is higher than Linear Regression. The higher tolerance of SVM to the approximation leads to achieving larger gains. Similarly, for SVM, the benefits of approximating larger circuits that have larger number of input features are higher since in these circuits, there are more operations that are less consequential, hence less sensitive to the approximation.

Table 4.3 breaks down the energy consumption of Linear Regression executing on the exact

Table 4.3: Energy consumption (nJ) of adders, multipliers and refreshing DRAM of Linear Regression in the exact mode and in APX-DMA with quality loss of 5%.

# Features	Mode	Adders	Multipliers	DRAM Refresh
32	Exact	0.33	499.2	132.8
	APX-DMA	0.18	167.52	102.6
64	Exact	0.66	998.4	132.8
	APX-DMA	0.44	416.1	102.6
256	Exact	2.66	3993.6	132.8
	APX-DMA	2.05	2194.28	110.2
512	Exact	5.32	7987.2	132.8
	APX-DMA	4.43	4698.35	110.2

and approximate system APX-DMA under quality loss of 5% into energy of multipliers, adders and DRAM refresh energy. Results show that although DRAM refresh energy is significantly larger than energy consumption of a single multiplier unit, due to a large number of multiply operations, total energy consumption of multiplier units becomes dominant even in the relaxed systems with proper parameter settings.

4.5.2 Optimal Algorithm-Hardware Approximation

In this section, we evaluate the proposed process to approximate the algorithm and the hardware for CNN accelerators by finding a proper approximate setting for the algorithm related approximate method and the approximate hardware units. We assume quantization as an approximate method to optimize the model computation and approximate DRAM unit as an approximate hardware unit in the underlying computing system. In this case, we assume that weights and feature maps of each layer can be quantized with different bit-width and read/written from/to the approximate DRAM with different voltage levels. Here, we show the deployment and the effectiveness of the proposed process to find the best bit-width and DRAM voltage level for weights and feature maps of each layer to maximize the compute efficiency while maintaining the classification accuracy within an acceptable range. We employ the proposed process to determine a proper configuration of the approximate hardware for DNN accelerators for ResNet models (ResNet56 and ResNet110) used for CIFAR-10 dataset. We implemented the process in PyTorch [PGC$^+$] by modifying the implementation of models provided in [res]. We choose two models – ResNet-56 and ResNet-110 – for CIFAR-10 dataset as the DNN models studied in this work. In order to train the approximate hardware parameters and fine-tune the models if necessary, we use NVIDIA GeForce RTX 2080 Ti GPU. We also estimate reduction in DRAM energy accesses due to the underlying approximation techniques and the learned configurations, with SCALE-Sim [SZW$^+$18] and DRAMPower [CWL$^+$18]. The rest of this section explains various approximation techniques used in this work in more details, defines the parameters and the problem for this case study, and evaluates the effectiveness of this method in finding proper approximate settings for the mentioned ResNet models.

Quantization

For the quantization method, we use the model-based error injection approach provided in Section 4.3.1. We assume that the original values are in fixed-point format with 16 bits (based on various techniques compared in [SCYE17], 16-bit fixed point format for the weights and feature maps is commonly used to get similar accuracy to the original model accuracy with 32-bit floating-point values.) Now, the goal is to reduce the number of bits for representing weights and feature maps in each convolutional layers to less than 16. To preserve the range of values, we assume that the number of bits for the integer part of the fixed-point values are fixed. Therefore, only the number of bits for the fractional part of fixed point values will be learned. To determine the number of bits for the integer part, we evaluate different under-study models during the inference phase with a subset of input dataset, and based on the values of weights and feature maps observed in various layers, we choose 5 bits for the integer part of the feature maps and 1 bit for the integer part of the weights. To learn the number of bits for the fractional part of the weights and feature maps in convolutional layers of a CNN model, in this work, we represent them as the parameters defined in Equation 4.10 and inject the quantization impact into the model through converting the values into their equivalent fixed point value using Equation 4.1 in the PyTorch implementation of CNN models.

Approximate DRAM Units

For approximate DRAM units, the voltage and the activation latency (t_{RCD}) and precharge latency (t_{RP}) [KOY$^+$19, CGW$^+$14, LKP$^+$15] can be reduced to save energy and expedite DRAM accesses. However, due to low voltage levels and insufficient time for data to be read/written properly, the values coming from approximate DRAM units are erroneous. To model this type of error in DNN models, we use the error model and parameters introduced in [KOY$^+$19, CYG$^+$17]. According to [CGW$^+$14, LKP$^+$15, CYG$^+$17], using faster and energy efficient DRAM by reducing their voltage and timing parameter is possible at the cost of increased bit error rate (BER). Therefore, we assume that bits of values coming from DRAM are erroneous at the rate of a specific BER depending on the level of reduction in their voltage and timing settings. Accordingly, to inject this error into DNN models, we assume that weights and feature maps coming to each layer are erroneous due to approximate DRAM units. As mentioned in Section 4.4, the error rate of their values (p_{err}) is a function of BER (p) which depends on the number of bits of the values (see equation 4.11). The impact of approximate DRAM units is therefore injected to the layers inputs and weights according to equations 4.2 and 4.11 and the error parameters defined in Equation 4.12. Since reducing the activation latency (t_{RCD}) and precharge latency (t_{RP}) uniformly flips the bits across several banks [CGW$^+$14, LKP$^+$15, CYG$^+$17], the erroneous value (E) in equation (4.2) is chosen uniformly at random within the minimum and maximum values that can be represented by the corresponding number of bits for weight and feature maps, which is similar to the Error Model 0 introduced in a prior work [KOY$^+$19].

In order to determine the voltage level for the learned BER obtained for each layer's weights and feature maps, we use the error model obtained in [CYG$^+$17] by applying various

Table 4.4: CNN model size, original model accuracy, and accuracy of the models with the approximate configurations learned at layer-wise granularity without fine-tuning (ft.) the original model parameters.

Model	Model Size	Original Acc.	Acc. with Layer-wise Appx. w/o ft.
ResNet-56	3.3 MB	92.48%	92.01%
ResNet-110	7.1 MB	94.36%	93.5%

voltage levels to several DRAM chips from different vendors. The model provided in [CYG+17] determines the fraction of beats (i.e., 64-bits of data transferred on the data bus) that are erroneous with a specific supply voltage level in DRAMs. Since the process learns the tolerable BER, we convert BER to beats error rate according to equation 4.17 and find the proper voltage level. Further explanation of this conversion will be provided in Section 4.5.2.

Problem Formulation

To formulate the problem for this case study we modify Equation 4.9 as below. For this problem, we make no assumption on the number of available resources on the underlying computing platform; therefore, like the previous case, the term related to resource constraint in Equation 4.9 can be removed. In addition, since in this case, our goal is to find the tolerability of different layers to quantization and memory error disregarding the architecture of the underlying computing platform, instead of formulating the cost function, the execution cost is indirectly minimized by adding the regularization terms into the objective function. The regularization terms guide the process toward learning lower bit-width and higher memory error rate to minimize the cost. Therefore, the regularization terms in this case are the L-2 norm of the parameters q and inverse of the parameters p regularized by specific decay factors.

Experimental Evaluation

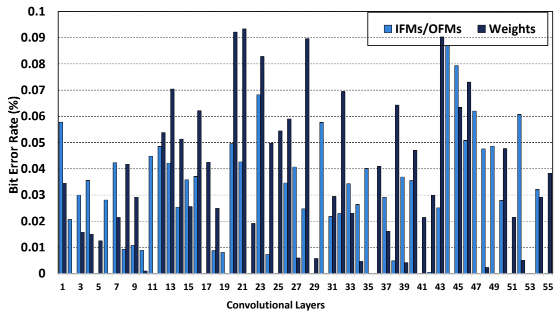
The accuracy and model size of the DNN models studied in this work are presented in Table 4.4. To learn the approximate hardware configurations for these models, we use the Pytorch implementation provided in [res] and modify them to inject the error of approximation methods explained earlier. To obtain the original model accuracy, we train original models from scratch and feed the trained models into the process to learn the approximation related parameters. To learn the approximate parameters, we initialize the weights in all layers in the modified CNN model by the pre-trained model weights. However, the approximate hardware parameters are initialized by a Gaussian distribution with a specific mean and variance. The mean and variance depend on the approximate technique. For quantization, we select 7.0 as the mean and 0.9 as the variance for the corresponding hardware parameters (q_f^l and q_w^l). For approximate DRAM, for the corresponding parameters that help to inject error to the weights (i.e., p_w^l), we set mean to 0.01 and variance to 0.009, and for the IFMs (i.e., p_f^l), we set mean to 0.001 and variance to 0.0009.

For the learning rate, we set it to 0.1 and train the modified model for a few epochs until the final accuracy of the modified model reaches within 1% of the original model accuracy. We observe that, in most of the cases, four training epochs are enough recover the accuracy. In order to allow higher degree of approximation, fine-tuning the model parameters (i.e., weights) after finding a set of approximate hardware configurations will help to compensate the accuracy drop. For this part, after a configuration is learned for the hardware parameters with a fixed set of weights, the hardware parameters are fixated and the weights of the model will be updated for a few iterations with the starting learning rate of 0.0001, and then after each 10 iterations, the learning rate will be multiplied by 0.1.

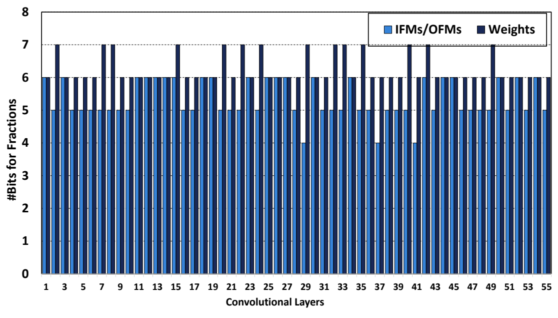
To evaluate the proposed process, we choose ResNet-56 and ResNet-110 models and specify approximate hardware configurations for different computation granularity. Based on the granularity of the computation blocks, each block can be executed with a separate quantization level with its own set of quantization parameters for weights and feature maps. The blocks can

Table 4.5: Accuracy obtained for ResNet-56 in various settings: layer-wise granularity with fine-tuning (ft.) the model parameters, filter-wise approximation without fine-tuning of original model, and layer-wise approximation when only quantization is considered.

Layer-wise Appx. w/ ft.	Filter-wise Appx. w/o ft.	Layer-wise Quantization
91.8%	92.0%	91.8%



(a) Bit Error Rate



(b) Number of Bits

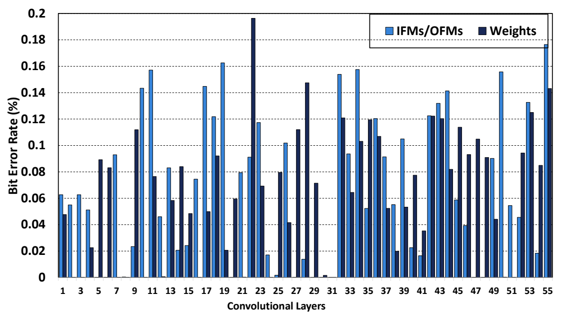
Figure 4.5: Bit error rate and the number of bits of feature maps (IFMs/OFMs) and weights in ResNet-56 learned by the proposed process without fine-tuning the original network parameters.

be a layer, a kernel, etc. Here, we assume two granularity levels: one at the layer level, called layer-wise approximation, where each layer can have its own set of error parameters, and one at the filter level, called filter-wise approximation, where each filter within a layer can have a separate set of error parameters.

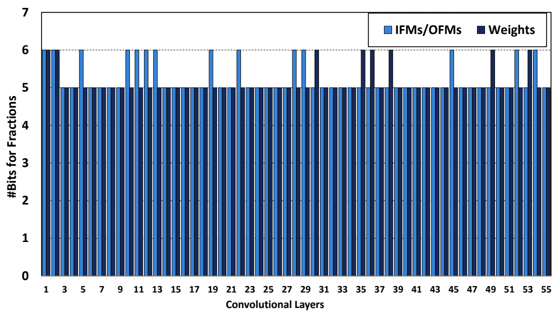
Layer-Wise Approximation

For layer-wise approximation, we assume that data in each layer (i.e., weights and feature maps) can use a specific approximate hardware configuration, and the configuration can vary across various layers. To apply layer-wise approximation to the model, a set of parameters shown in equations (4.10 and 4.12) are defined.

ResNet-56. For this model, we train the approximate parameters once without fine-tuning the original model weights and once with fine-tuning the weights. For the first case, the learned bit error rates and the number of bits for weights and feature maps for each convolutional layer



(a) Bit Error Rate



(b) Number of Bits

Figure 4.6: Bit error rate and the number of bits of IFMs/OFMs and weights in ResNet-56 learned by the proposed process with fine-tuning the original network parameters. Fine-tuning the model parameters help to choose more efficient configurations (compare the parameters in this figures with those in Figure 4.5.)

are shown in Figure 4.5. Table 4.5 shows the accuracy of the model with this approximate configuration. Figure 4.5a shows several key points about the tolerable memory error of ResNet-56. First, it shows that most of the layers with high tolerable memory error are those in the middle of the network, while the layers in the beginning and end of the network have lower error tolerance. Comparing the BER of weights and feature maps, we see that, in the first seven layers, feature maps can tolerate higher error rate than the weights, whereas in the last layer, when the feature maps cannot tolerate any memory error, the BER of weights are around 0.04%. On average, in 56.36% of the layers, the weights can tolerate more memory error than the feature maps.

Regarding quantization, Figure 4.5b shows that the number of tolerable fractional bits for the weights in different layers are either 6 or 7. While around 25% of the layers require 7 bits, we can assign 6 bits to the fractional part of the weights in 75% of the layers. For the feature maps, the tolerable bit-width for the fractions is 4-6, where only 3 layers in the middle of the network out of 55 layers can work with 4 bits.

In the next set of experiments, we allow fine-tuning the model parameters after a set of approximate hardware parameters are learned in the case that the obtained accuracy is not acceptable. The bit error rates and the number of bits learned with this setting are summarized in Figure 4.6. As shown in the figure, both the BER and the number of bits are improved. The number of bits for the fractional part of feature maps is between 5-6 but only 25% of the layers are quantized with 6-bit fractional part. For the weights they are reduced to 5-6 bits and only 14% of the layers require 6 bits and the rest work under 5-bit fractional parts. The accuracy of the model with this approximate setting and updated model parameters is also provided in Table 4.5. As we can see, by fine-tuning the model parameters, we can choose more relaxed hardware parameters for the quantization as well as BER, while the accuracy is also maintained.

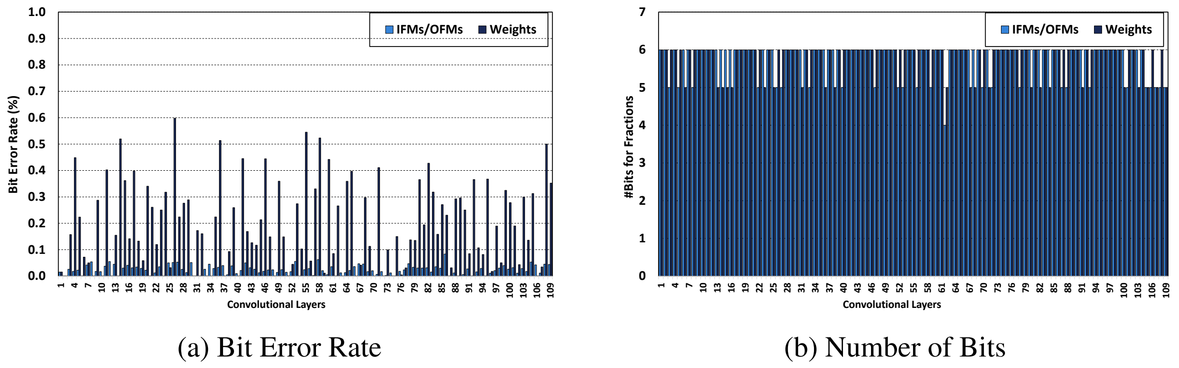
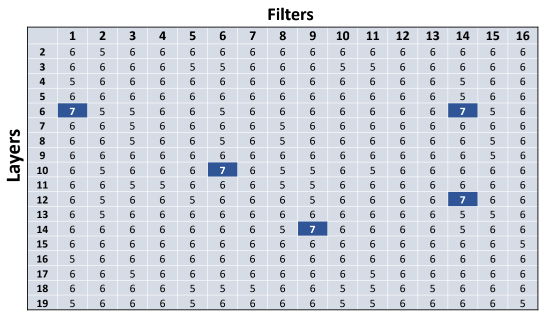


Figure 4.7: Bit error rate and the number of bits of feature maps (IFMs/OFMs) and weights in ResNet-110 learned by the process without fine-tuning the original network parameters. The learned approximate hardware parameters for ResNet-110 are more efficient than those of ResNet-56 due to its larger size and higher error tolerance.

ResNet-110. Since the approximation degree highly depends on the tolerance of DNN models to errors, and larger-sized models (due to over-parameterization and higher generalizability) are more error-tolerant, the process is expected to find higher degree of approximation (i.e., more efficient hardware configurations) for larger models than ResNet-56. To evaluate the efficacy of the framework for larger models, we use the process to find approximate configurations for ResNet-110. For a fair comparison, we train the parameters of ResNet-110 without fine-tuning the original model parameters and use a similar initialization of the parameters as for ResNet-56. The BER and quantization parameters for different layers are summarized in Figure 4.7.

As expected, ResNet-110 model has higher tolerance to approximation than ResNet-56. Higher error tolerance of larger model such as ResNet-110 is more realizable on both the bit error rates and quantization. Comparing Figure 4.5 and 4.7, we see significant difference of these two

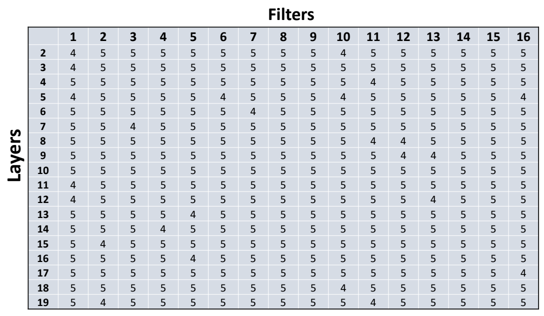


Filters

Layers

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	6	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6
3	6	6	6	6	5	5	6	6	6	5	5	6	6	6	6	6
4	5	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6
5	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6
6	7	5	5	6	6	5	6	6	6	6	6	6	6	7	5	6
7	6	6	5	6	6	6	6	5	6	6	6	6	6	6	6	6
8	6	6	5	6	6	5	6	5	6	6	6	6	6	6	5	6
9	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6
10	6	5	6	6	6	7	6	5	5	6	5	6	6	6	6	6
11	6	6	5	5	6	6	6	5	5	6	6	6	6	6	6	6
12	6	5	6	6	5	6	6	6	5	6	6	6	6	7	6	6
13	6	5	6	6	6	6	6	6	6	6	6	6	6	5	5	6
14	6	6	6	6	6	6	6	5	7	6	6	6	6	5	6	6
15	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5
16	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
17	6	6	5	6	6	6	6	6	6	6	5	6	6	6	6	6
18	6	6	6	6	5	5	5	6	6	5	5	6	5	6	6	6
19	5	6	6	6	5	6	6	6	6	5	5	6	6	6	6	5

(a) Weights



Filters

Layers

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	4	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5
3	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
4	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5
5	4	5	5	5	5	4	5	5	5	4	5	5	5	5	5	4
6	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5
7	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5
8	5	5	5	5	5	5	5	5	5	5	4	4	5	5	5	5
9	5	5	5	5	5	5	5	5	5	5	5	4	4	5	5	5
10	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
11	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
12	4	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5
13	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5
14	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5
15	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5
16	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5
17	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4
18	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5
19	5	4	5	5	5	5	5	5	5	5	4	5	5	5	5	5

(b) Feature Maps

Figure 4.8: Number of bits for the fractional part of quantized feature maps and weights in different filters of layers 2-19 of ResNet-56 for CIFAR-10 dataset learned by the proposed process without fine-tuning the original network parameters. The highlighted boxes indicate the filters with 7-bit fractional parts, which are sparse due to filter-wise approximation.

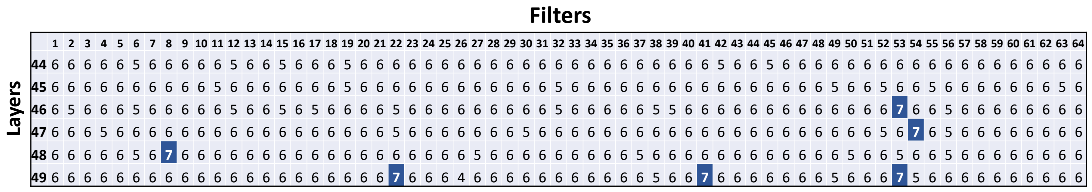
models in their error tolerance characteristics. The results show that tolerable BER of the weights in most of the layers in ResNet-110 is 6.47× higher than that of ResNet-56 (the average BER of the weights in ResNet-56 is 0.03% while in ResNet-110 is 0.19%). The average BER for the feature maps are similar in two models (≈0.027%).

The results for the number of bits (see Figure 4.7b) show that the tolerable bits for the fractional parts of the weights and feature maps are either 5 or 6 (except one layer that can work with 4 bits for its feature maps). While the weights in ResNet-56 in ≈ 25% of the layer require 7 bits for their fractional parts, in ResNet-110, none of the layers require 7 bits (78.8% of the layers require 6 bits and 21.2% can work with 5 bits).

Filter-Wise Approximation

In this set of experiments, we assume that approximation can be performed at a finer granularity of computation, and evaluate the process in finding a proper approximate configurations for the resulting computation blocks.

We set the granularity of computation for DNN models at the filter level for the quantization technique and keep the layer-wise granularity for approximate memory units. This setting allows the weights in a layer for different filters to have different bit-widths. It also allows the

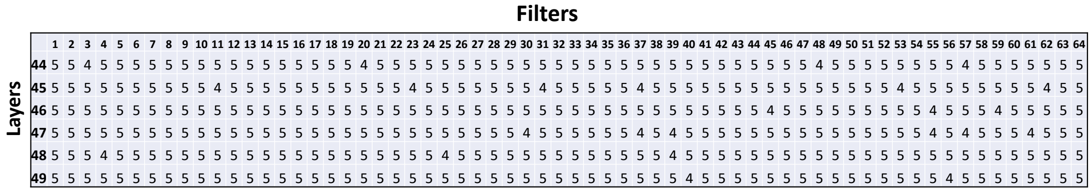


Filters

Layers

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
44	6	6	6	6	6	5	6	6	6	6	6	5	6	6	5	6	6	6	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
45	6	6	6	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	5	6	6	5	6	6	6	6	6	6	6	5	6
46	6	5	6	6	6	5	6	6	6	6	6	5	6	6	5	6	5	6	6	6	6	5	6	6	6	6	6	6	6	6	6	5	6	6	6	6	6	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	**7**	6	6	5	6	6	6	6	6	6	6	6
47	6	6	6	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	**7**	6	5	6	6	6	6	6	6	6	6
48	6	6	6	6	6	5	6	**7**	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	6	6	5	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	5	6	6	5	6	6	6	6	6	6	6	6
49	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	**7**	6	6	6	4	6	6	6	6	6	6	6	6	6	6	6	5	6	6	**7**	6	6	6	6	6	6	6	5	6	6	6	**7**	5	6	6	6	6	6	6	6	6	6	6

(a) Weights



Filters

Layers

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
44	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5
45	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	4	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	4	5	5
46	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5	4	5	5	5	4	5	5	5	5	5
47	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	4	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	4	5	5	5	4	5	5	5
48	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
49	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	5	5	5	5	5	5	5	5

(b) Feature Maps

Figure 4.9: Number of bits for the fractional part of quantized feature maps and weights in different filters of layers 44-49 of ResNet-56 for CIFAR-10 dataset learned by the proposed process without fine-tuning the original network parameters. The highlighted boxes indicate the filters with 7-bit fractional parts, which are sparse.

feature maps across different channels to have different bit-widths. We then use the proposed process to determine the quantization degree for weights in various filters in each layer and various channels of input feature maps in a DNN model as well as memory BER for various layers. With this setting, we use the proposed process to learn the approximate hardware parameters for ResNet-56. To compare it with layer-wise approximation, we use similar training hyper-parameters.

Figures 4.8 and 4.9 show the number of fractional bits of the weights and feature maps learned by the proposed process for a few layers in the beginning and end of ResNet-56 model. Due to space limit, we only show the results of a subset of layers in those figures. The results show that for the weights, while in layer-wise approximation, 25% of the layers weights require 7 bits for the fractions, here in filter-wise approximation, 7 bits are required by only a few filters across the model. As we can see, the highlighted boxes in figures 4.8a and 4.9a showing the filters with 7 fractional bits are sparse. Compared to layer-wise approximation, the lower number of the required bits for weights in filter-wise case happens because of varying significance of different filters in the same layer that leads to varying levels of error tolerance. Therefore, unlike layer-wise

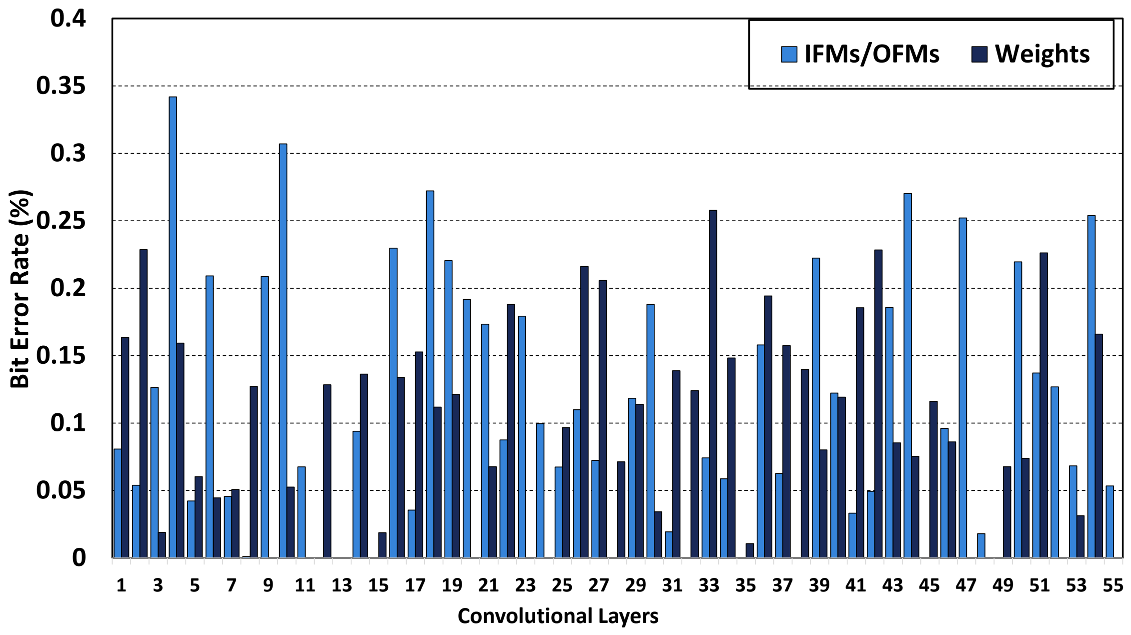


Figure 4.10: Bit error rate of feature maps and weights in different layers of ResNet-56 for CIFAR-10 dataset learned by the proposed process without fine-tuning of the original model parameters when feature maps and weights are quantized with filter-wise approximation setting (each filter (input channel) in weights (input feature maps) is considered to be quantized with a separate number of bits.)

approximation where the bit-width of all the filters in a layer is set to the tolerable bit-width of lowest error tolerant filters, filter-wise approximation can reduce the bit-width more. With layer-wise approximation, most of the layers of ResNet-56 work with 6-7 bits for the weights and 5-6 bits for feature maps. The number of bits for the feature maps in most of the filters in different layers are 4-5 and 5-6 for the weights.

The bit error rate of different layers learned by the proposed process along with filter-wise quantization is provided in Figure 4.10. In addition to lower bit width, higher tolerable bit error rate is also achieved in filter-wise approximation. The average BER for feature maps (weights) in filter-wise approximation is 0.11% (0.098%) while in layer-wise approximation it is 0.028% (0.032%). The results for filter-wise approximation indicate that allowing different approximate hardware configurations for different computation blocks at a finer granularity can increase the degree of approximation, hence improving the cost of execution.

Energy Estimation for Main Memory Accesses

We also evaluate the main memory energy consumption that can be saved by the configurations the proposed process determines for a DNN model. In order to estimate the energy consumption of accessing main memory units, we assume that approximate memory units can be achieved by scaling the main memory supply voltage (V_{dd}). Now, the first step to estimate energy consumption of memory units is to choose a proper voltage level based on the learned bit error rates (BER). To specify which voltage level leads to a specific BER, as mentioned before we use the model provided in [CYG$^+$17]. In [CYG$^+$17], various DRAM chips from different vendors are tested with various voltage levels and the error rates are measured. Based on the experiments performed in [CYG$^+$17], both the minimum voltage that no error occurs and the rate of the error vary across different vendors and DRAM chips. For this work, we choose the vendor that is more sensitive to error, and estimate the proper voltage for the learned BERs based on the model provided for that vendor in [CYG$^+$17]. The error metric in the provided model is the rate of the errors in a set of data beats. Beat is the unit of data transferred through the data bus. Each beat is considered as 64-bit data. The fraction of beats that are erroneous ($p_{faultybeats}$) for each voltage levels is provided in Table 4.6. To use this error characteristics to estimate the energy of DRAM units in our work, we need to first convert BER to the beats error rate, and choose a proper voltage accordingly. This conversion is necessary since the proposed process provides the tolerable BERs for each specified computation block in a given DNN model. To do this, we use the equation below (4.17) to obtain the probability of n-bit values that are erroneous (p_{err}) based on a given beat error rate ($p_{faultybeats}$), and then according to equation (4.11), we can find the BER by replacing p_{err} with the obtained p_{err} value:

$$p_{faultybeats} = 1 - (1 - p_{err})^{64/n}, \tag{4.17}$$

Table 4.6: Error model (percentage of beats that are erroneous) in approximate DRAM units under various voltage levels obtained from [CYG+17]

Supply Voltage	**V $>$ 1.25**	**V=1.25**	**V=1.2**	**V=1.15**	**V=1.1**
Beats Error Rate (%)	0.0	10%	35%	40%	50%

where $64/n$ is the number of n-bit values that each beat represents. $(1-p_{err})^{64/n}$ represents the probability that no error occurs in a beat.

For each supply voltage level and a given bit-width for the values, we have a specific BER and we can use it to determine the voltage level for each computation blocks in a DNN (based on the tolerable BER learned for that block.)

Now that the supply voltage is computed, the second step is to estimate the energy consumption of DRAM accesses. For energy consumption, we use Eyeriss [CES16] as an underlying DNN accelerator and estimate the DRAM energy consumption using the cycle-accurate SCALE-Sim simulator [SZW+18] and DRAMPower [CWL+18]. SCALE-Sim simulator is used to obtain memory traces of a given DNN on Eyeriss (i.e., required DRAM bandwidth and bytes for transferring feature maps and model weights) based on the topology of the DNN models (ResNet-56 and ResNet-110) we provided to SCALE-Sim. Then, we use the obtained traces and the bit-width of data to calculate DRAM read/write cycles, and feed the cycles to DRAMPower to estimate the energy consumption. In DRAMPower, we choose a 2GB DDR3 DRAM with the nominal voltage of 1.5V from a set of predefined memory specifications, and change its voltage to consider the impact of voltage scaling in DRAM energy consumption.

For the energy consumption comparison, we assumed that the baseline models' weights and feature maps are 16-bit fixed-point values since this setting does not impact the accuracy [SCYE17]. Also, we assumed that the baseline voltage for DRAM units is 1.5V. The summary of energy consumption is provided in Figure 4.12. We estimate the energy reduction of ResNet-56 and ResNet-110 based on the configurations for quantization and approximate memories

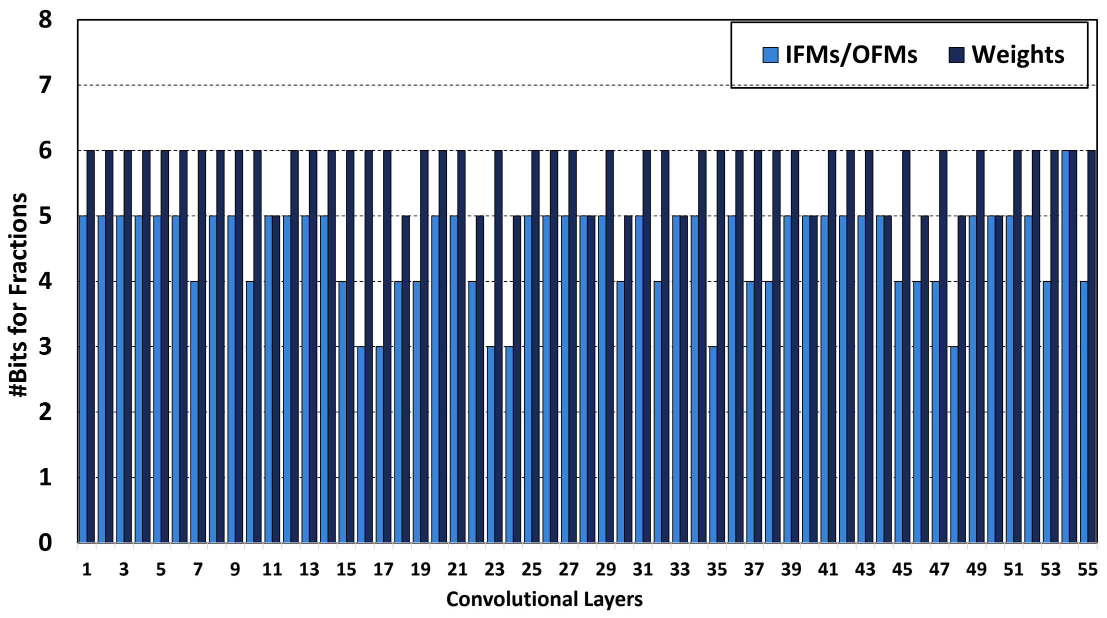


Figure 4.11: The number of bits for the fractional parts of feature maps and weights in different layers of ResNet-56 learned by the proposed process without fine-tuning of the original model parameters when the only source of error is quantization with layer-wise approximation setting. For ResNet-56, the number of width is lower than the case where approximate memories are also available.

summarized in figures 4.6 and 4.7. For ResNet-56, we consider the settings that are learned with additional fine-tuning of the model parameters and estimates its energy consumption. For ResNet-56, due to low tolerance of this network to bit errors, the supply voltage for almost all the layers (except for the weights of two layers) is required to be within the safe range (i.e., 1.3-1.5V based on the evaluations performed on different DRAM chips under different voltage levels provided in [CYG+17].) Therefore, the only source of energy reduction is quantization that reduces the memory accesses. Assuming the DRAM voltage of 1.3V for executing the model on both baseline and approximate hardware, the DRAM energy reduction of the obtained configurations by the proposed process is 45.86%. If we assume no voltage reduction for the baseline accelerator (i.e., 1.5V) and safe reduction for approximate accelerators (1.3V), the reduction in energy saving is 68.03%. This energy saving is acquired by quantization not the bit errors.

For ResNet-110, due to the larger size of this model, bit error rates are higher. However, these larger bit error rates still cannot significantly change the voltage. We observed that weights

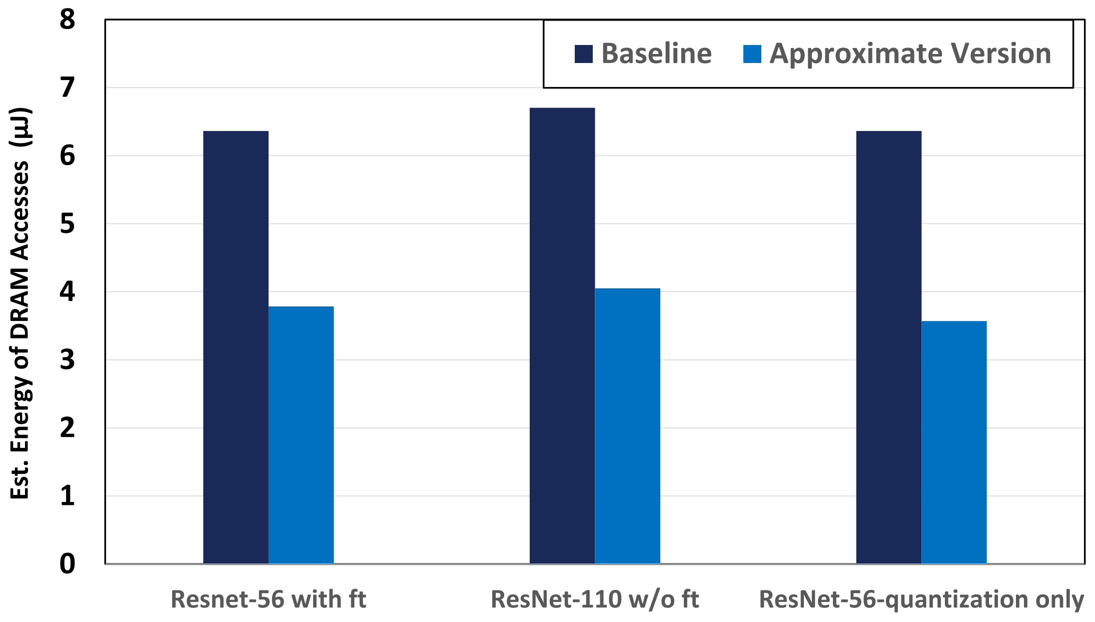


Figure 4.12: Estimated energy (μJ) of DRAM accesses for various models under various configurations. Nominal baseline voltage for DRAM is 1.5V. The first two configurations are for ResNet-56 with fine-tuning of the model parameters (Figure 4.6) and ResNet-110 without any fine-tuning (Figure 4.7). The last configuration is for ResNet-56 under the quantization without any bit error rates (Figure 4.11).

in 50% of the layers can work with voltage level of 1.25V, while the range of BER for feature maps still requires 1.3V as the supply voltage for DRAM units. Based on this voltage settings and the bit-width of feature maps and weights, the DRAM energy reduction is 43.85% considering 1.3V of the DRAM supply voltage for baseline model, and 65.46% considering 1.5V DRAM voltage for the baseline setting. Only 1.9% of the energy reduction comes from the scaled voltage of DRAM beyond the safe point and 98% of the reduction is from quantization.

Based on this evaluation, with the proposed process, we can conclude that for small models, the rate of acceptable memory errors is not significant and can be discarded from approximate design space to make room for other sources of approximation such as quantization. To evaluate this point, we only inject the error of quantization in ResNet-56 and learn the number of bits for the fractional parts without fine-tuning the original model. As we see in Figure 4.11, the number of required bits is much smaller compared to Figure 4.5b. We can see some layers that work with 3 and 4 bits as their fractional part. This configuration leads to higher DRAM energy saving compared to the one with the bit error rate. Estimating the DRAM energy saving shows that, compared to 1.5V DRAM in the baseline model, 78.25% energy will be saved and compared to 1.3V, 54.74% energy reduction is achieved.

Since we use a statistical approach to model the memory errors, to validate the configurations and energy saving obtained by the proposed process, we compare the obtained results with the work that is mostly related to our work [KOY$^+$19]. In this work, the error is induced into DNN models based on the actual data obtained from DRAM Chips with scaled voltages. Similar behavior for small size of models is also observed in [KOY$^+$19]. It shows that small models with the size of less than 10MB cannot tolerate high memory errors (e.g., the tolerable BER is less than 0.5% for SqueezeNet model [KOY$^+$19]), which results in insignificant energy saving.

4.5.3 Optimal FPGA-Aware Algorithm-Hardware Approximation

In this section, we evaluate the proposed process on optimizing the FPGA-based CNN accelerators by finding a proper approximate setting for both model and hardware related approximate methods based on the available resources on FPGA as well as finding an optimal configurations for the number of MAC units. Similar to Section 4.5.2, we assume quantization as an approximate method to optimize the model computation and approximate DRAM unit as an approximate hardware unit in the underlying computing system. In this set of experimental evaluation, for quantization, we consider BFloat16 format as the underlying quantization approach and aim to lower the bit-width of the mantissa to lower than 8-bit. We assume that weights and feature maps of each layer can be quantized with different bit-width.

In this section, we also consider the type of dataflow used in the underlying architecture that impacts off-chip memory accesses. We assume that the dataflow is input-output stationary, meaning that part of input feature maps will be brought on-chip and reused until all the related operations of a layer are performed. In addition, the related intermediate output of a set of subsequent layers for that part of input remain on-chip until all the operation of all the layers within the set are executed. Therefore, only the input/output feature maps of each set of layers will experience the error of DRAM voltage reduction. Since the weights of all layer will be always read from DRAM, thus being impacted by approximate DRAM error.

We employ the proposed technique explained in Section 4.4 to determine a proper platform-aware algorithm-hardware approximation and architecture configuration for DNN accelerators for ResNet-56 used for CIFAR-10 dataset. We use PyTorch [PGC$^+$] to modify the implementation of the model provided in [res]. In order to train the approximate hardware parameters, we use NVIDIA GeForce RTX 2080 Ti GPU. We also estimate reduction in DRAM energy accesses with DRAMPower [CWL$^+$18].

Quantization

In this case, for the quantization method, we use the model-based error injection approach similar to the one provided in Section 4.3.1.

We assume that the original values are in BFloat16 format with 16 bits. BFloat16 uses one bit for the sign, 8 bits for the exponent and 7 bits for the mantissa. Therefore, 8-bit MAC units are used to perform operations in different layers of a CNN. Now, the goal is to explore the possibility of using less than 7 bits for the mantissa to improve the execution cost without drastic degradation of the accuracy.

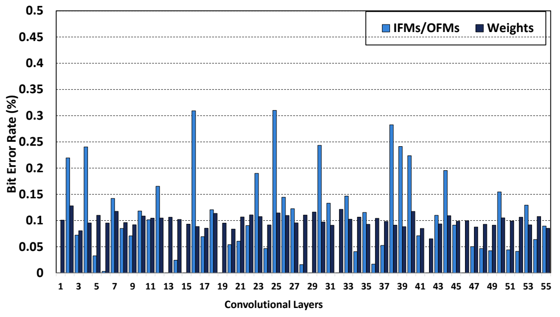
To learn the number of bits for the mantissa of the weights and feature maps in convolutional layers of a CNN model, in this work, we represent them as the parameters defined in Equation 4.10 and inject the quantization impact into the model through converting the values into their equivalent BFloat value. To approximate values to their BFloat value we extract their mantissa, converting them to the corresponding q number of bits and turning them back to the BFloat format, which is implemented in the PyTorch and applied to the CNN computation.

Table 4.7: The estimated coefficients α and β in Problem 4.2. to model the required number of LUTs and FFs for different number of MAC units based on the required resources for 16 MACs

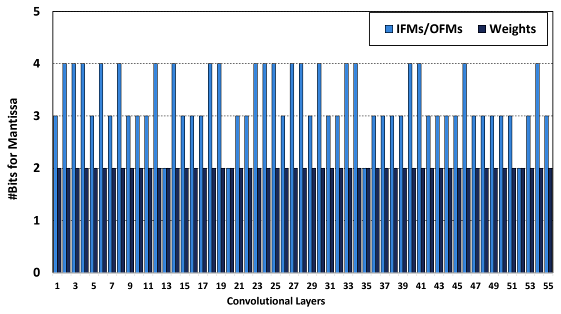
Resources	16 MACs	32 MACs	64 MACs	128 MACs	256 MACs
Coeff. for LUT (α_M)	1.0	1.2	1.68	2.25	3.78
Coeff. for FF (β_M)	1.0	1.04	1.12	1.28	1.54

Approximate DRAM Units

For approximate DRAM units and injecting their impact into a given CNN model, we use a similar process described in Section 4.5.2. However, because of the impact of dataflow of the underlying architecture, as mentioned earlier, in this set of experiments with input-output stationary dataflow, energy saving from DRAM voltage reduction can be achievable through low cost accesses to all layers weights depending on their error tolerability as well as low-cost read/write accesses of feature maps of a subset of layers. For the understudied CNN model, a sequence of layers without any pooling layers or a sequence of layers, the convolutional layers of which have the stride of 1, are considered as a set of layers that are pipelined on the chip; therefore, only the input feature maps of the first layer and output feature maps of the last layer in the set will be impacted by approximate DRAM while the input/output feature maps of the intermediate layers remain intact. As described in Section 4.5.2, to model this type of error in DNN models, we use the error model and parameters introduced in [KOY$^+$19, CYG$^+$17]. Accordingly, to inject this error into DNN models, we assume that weights of all layers and feature maps of the impacted layers, explained above, are erroneous due to approximate DRAM units. The impact of approximate DRAM units is therefore injected to the layers inputs and weights according to equations 4.2 and 4.11 and the error parameters defined in Equation 4.12.



(a) Bit Error Rate



(b) Number of Bits

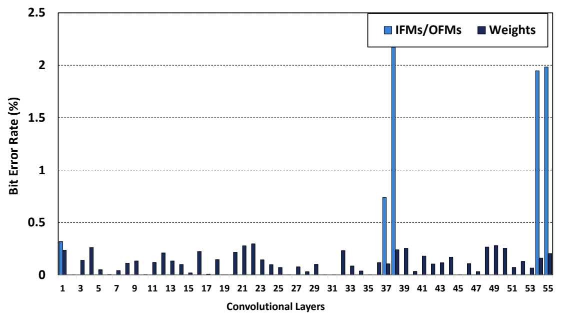
Figure 4.13: Bit error rate and the number of bits for the mantissa of feature maps (IFMs/OFMs) and weights represented in BFloat format in ResNet-56 learned by platform-aware algorithm-hardware approximation process. The numbers are learned assuming that exact values of feature maps and weights are in BFloat16 format.

MAC Units on FPGA

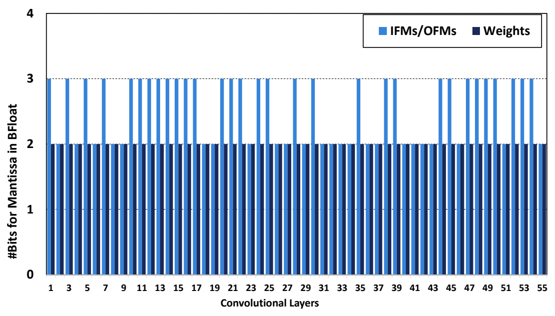
To learn the best number of MAC units for each bit-width configuration given the limited resources on a given FPGA through solving Problem 4.2., the values of parameters $\alpha_{M_{q_w q_f}}$, $\beta_{M_{q_w q_f}}$, $T^{Base}_{q_w q_f}$ and $F^{Base}_{q_w q_f}$ are required to be pre-determined before running SGD. To set these parameters, the base number of MAC units is set to 16, and the goal is to find $M_{q_w q_f}$ as a multiple of 16 MAC units. Therefore, $T^{Base}_{q_w q_f}$ and $F^{Base}_{q_w q_f}$ are the number of LUTs and FFs to implement 16 $q_w - bit \times q_f - bit$ MAC units. To determine $\alpha_{M_{q_w q_f}}$ and $\beta_{M_{q_w q_f}}$, we implemented one convolutional layer with different bit-widths and number of MAC units (multiples of 16 MAC units), and profiled their required number of LUTs and FFs. Based on the obtained values, $\alpha_{M_{q_w q_f}}$ and $\beta_{M_{q_w q_f}}$ are estimated and summarized in Table 4.7.

Experimental Evaluation

To compare this optimization process with the one described in the previous example in Section 4.5.2 that does not consider the underlying platform in the optimization process, we apply the process of finding the optimal configurations for algorithm-hardware level approximation on ResNet-56 once without any assumption on the underlying hardware platform (the results are shown in Figure 4.13) and once assuming that the model is executed on small FPGAs used for



(a) Bit Error Rate



(b) Number of Bits

Figure 4.14: Bit error rate and the number of bits for the mantissa of feature maps (IFMs/OFMs) and weights represented in BFloat format in ResNet-56 learned by platform-aware algorithm-hardware approximation process. The numbers are learned assuming that exact values of feature maps and weights are in BFloat16 format.

edge devices as its underlying computing platform (the results are shown in Figure 4.14). For this set of experiments, we choose a Xilinix FPGA of Zynq-7000 series (i.e., xc7z020-clg484).

As shown in Figure 4.14b, for ResNet-56, 2 bits for the mantissa of the weights in all layers and 2-3 bits for the feature maps are learned. Compared to the learned bit-width setting without considering any specific hardware platform (Figure 4.13b) which requires three convolutional groups implemented on the underlying platform, this set of learned bit-width setting for different layers only requires two groups of convolutional layers with lower bit-width.

BERs learned for the weights of different layers summarized in Figure 4.14a allow to reduce the voltage of DRAM for the weights to 1.25V in $\approx$35% of layers. For the feature maps of the layers that are accessed through DRAM, the voltage can be reduced to 1.1-1.2V. The number of MAC units that are learned by our method is 128 for both convolution groups, one with 2-bit weights and 3-bit feature maps and one with 2-bit weights and 2-bit feature maps. Total number of utilized resources of Xilinx FPGA of xc7z020-clg484 for ResNet-56 implemented with BFloat16 and the learned quantization setting and their corresponding energy consumption of FPGA are presented in Table 4.8. Compared to the model with BFloat16, the model implemented by the bit-width settings and the number of MAC units learned by our method achieves 24% energy saving. In addition, the reduction in DRAM energy consumption is estimated as 52.85% which

Table 4.8: On-chip energy consumption on FGPA, energy improvement of DRAM accesses and resource utilization on a Xilinx Zynq-7000 series FPGA (xc7z020-clg484) for original ResNet-56 in BFloat16 and platform-aware quantization and approximate DRAM settings.

	Original	**Approximate**
Energy Cons. on FPGA (mJ)	63.4mJ	50.9mJ
Energy Saving of DRAM	N/A	52.85%
BRAM	81	56
DSP	5	5
FF	20536	13762
LUT	48890	51703

comes from the reduced size of data with the learned quantization settings and reduced voltage based on the learned tolerance of layers to bit error rates.

4.6 Summary of the Chapter

Approximate computing improves energy consumption of many applications by relaxing the design constraints imposed at various levels of design from algorithms to hardware. To effectively use this capability requires simultaneous and non-uniform use of multiple approximate resources at both the algorithm and hardware level that together result in overall efficiency gains while ensuring bounds on the quality of the results. In addition, considering hardware limitations such as limited resources and the underlying architecture design when using various approximate methods with non-uniform configurations throughout the computation of a DNN model along with optimizing the underlying architecture to support such non-uniform approximation guarantee the effectiveness of approximate computing in practice. This chapter addresses the mentioned challenges by formulating the problem of determining an optimal approximation setting as a parametric optimization problem that is amenable to recent advances in optimization methods in machine learning. This is implemented in a framework which can be integrated into design automation tools such as synthesis tools to provide efficient platform-aware algorithm-hardware approximate settings and optimal supporting architecture level design choices given the underlying platform constraints.

Chapter 4, partly contains the materials of Vahideh Akhlaghi, Sicun Gao, and Rajesh Gupta, "LEMAX: learning-based Energy Consumption Minimization in Approximate Computing with Quality Guarantee", *in Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2018. In addition, this chapter contains the unpublished materials co-authored by Vahideh Akhlaghi, Dezhi Hong, Sicun Gao, Hadi Esmaeilzadeh and Rajesh K. Gupta. This dissertation author is the primary author of the mentioned materials.

Chapter 5

Parameter Approximation

DNN algorithms are proliferating to the“edge” and “end” devices, that is, devices at the edge or near the edge of the network, due to a growing importance of data privacy and real-time data processing and decision making needs. Emergence of federated learning as a distributed training method to train DNNs on edge devices with users’ data is an indication of this increased demand to implement DNNs on edge devices. However, efficient DNN processing on edge and end devices requires to address and solve several key challenges. Limited bandwidth and communication cost are the main bottlenecks for the distributed training algorithms, including federated learning. In addition, limited off-chip storage and on-chip memory and logic on the edge devices prohibit the deployments and efficient computation of large yet highly-accurate models on edge devices. To overcome the mentioned challenges, in this chapter, we propose a novel and simple algorithm-hardware optimization solution that reduces the size of DNN model, in particular CNN models, by reducing their number of parameters. This approach alleviates the communication bottleneck in distributed training and federate learning by reducing the required number of gradients transmitted between devices. It also makes CNN models compatible with and efficient to be executed on embedded devices with scarce resources and limited energy budget by reducing the size of the model residing in their storage and memory units and the memory accesses during the execution of the model. Therefore, the proposed approach paves the way for privacy preserving algorithms to become ubiquitous and for real time data processing and decision making to become realizable through edge and end devices.

5.1 Introduction

For the training of large CNNs, where distributed machine learning approaches are typically used, communication constraints present a key challenge, as gradients of the network parameters need to be communicated among different nodes [WSL$^+$18]. Similarly, in federated learning, in which a neural network is continuously optimized and customized in a distributed manner using numerous users' devices [KMY$^+$16] with limited resources, the size and implementation efficiency of these networks are also critical [MARAM18, LBG$^+$15, WBC$^+$19].

A possible solution to these problems is reducing the number of parameters of the CNN in a way such that its performance is not tangibly affected. Most of today's techniques, however, either focus on the inference phase and do not reduce the number of parameters during the training phase (while maintaining the accuracy of the network), or they reduce the number of parameters during the training, but their accuracy loss, computational burden, or implementation cost is considerable [CWZZ18]. This chapter seeks to overcome these limitations by introducing a novel plug-and-play algorithm-hardware optimization approach to reduce the number of parameters of any CNN architecture. The reduced number of parameters improves the performance of training and inference of CNN models by reducing the size of gradients, the required storage and memory accesses.

Our proposed method exploits the inherent redundancy in the parameters of the convolutional filters and provides a hardware-aware partitioning of the set of filters of convolutional layers and representing these partitions in a low dimensional latent space. This hardware-aware parameter reduction not only helps to reduce the size of gradients during training, it reduces the memory accesses during both inference and training as well.

To obtain this low-dimensional representation of the CNN filters, we introduce an auxiliary neural network, called Convolutional Slice Generator (CSG), that can be used in conjunction with any CNN architecture. The CSG, which is shared among all the convolutional layers, generates four dimensional tensors called "slices" that correspond to the above-mentioned cross-filter partitions, from a low dimensional "code" space. These slices are then concatenated to form sets of convolutional filters of all the layers of the original architecture. Although our technique supports any arbitrary shape of slices, to improve the memory accesses during CNNs' execution, the shape of these slices can be selected such that it complies with the shape of the partitions of the filters accessed repeatedly by CNN accelerators, which leads to accessing fewer parameters from a lower dimensional space (i.e., "code" space).

To elaborate the method, the trainable CSG network takes as input a set of trainable code vectors corresponding to partitions or slices of sets of convolutional filters of the layers and re-produces these slices which are then concatenated to make the regular shaped but approximate version of the set of convolutional filters of the layers of the original architecture. These approximate versions of the filters replace the original filters of the convolutional layers of the CNN both during the training and inference phases (see Fig. 5.1). Thus, by design this compression approach that preserves the original CNN architecture (while approximating its set of filters) can be applied in conjunction with many other methods for additional gains, and can be used during both the training and inference phases. During the training of the CNN, the code vectors, which lie in a space of cardinality $\approx 20\times$ smaller than the cardinality of their corresponding slice of the convolutional filters, are trained. We have explained our method on how this compression ratio is achieved eliminating the need for tuning these parameters. The auxiliary neural network (CSG) can either be trained alongside the code vectors or be provided to the network in advance with pre-trained and fixed parameters. Due to the simplicity of our method, using a recent result [AZLS19] and for a simplified architecture we have theoretically shown the convergence of training in our approach which is backed by our experimental evaluations. In addition, to support

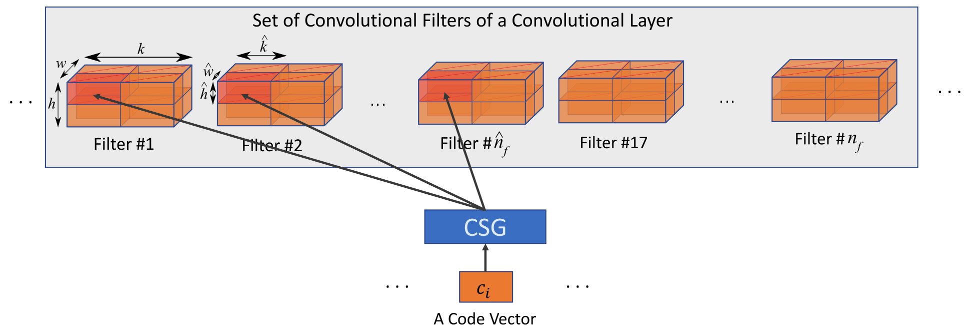


Figure 5.1: Generation of a regular-shaped but approximate set of filters from the concatenation of slices. Each slice of a set of convolutional filters is generated from a code vector using a shared CSG. The generated filters are then used by the corresponding convolutional layer as in regular CNNs. The proposed method can be applied to any filter shape. In this example there are n_f filters with k channels and $h \times w$ kernels. Each slice, generated by the CSG, is assumed to be $\hat{n}_f \times \hat{k} \times \hat{h} \times \hat{w}$. The figure shows one slice in darker color, that spans across multiple channels and multiple filters, and its corresponding code vector (see Sections 5.3 and 5.4).

CSG-augmented CNN architecture on the embedded device for improved inference for edge applications, we introduce Binarized Slice Generator (BSG), which binarizes and optimizes CSG parameters based on the hardware limitation in small edge devices. Accordingly, a BSG-augmented accelerator architecture is proposed to support execution of BSG-augmented CNNs by a providing a slight modification to the conventional CNN accelerators.

We apply our proposed technique to several CNN architectures used for classification and semantic segmentation tasks and compare it with most of the state of the art methods that are comparable to our approach. Our experiments on classification tasks show that while this approach significantly reduces the cardinality of the parameter space of the CNN, the resulting networks, except in extreme compression cases, still achieve top-1 accuracies that are within one percent of the accuracies of the original CNNs. Our approach for some modern wide architectures improves the original accuracy by a compression ratio of $\approx 2\times$, which is compatible with a general trend [CWZZ18] that wider networks can tolerate more compression. In case of narrow networks, when our technique maintains the accuracy within one percent of the original ones, other compression methods lead to higher accuracy degradation with similar compression ratios. To further confirm the generality of our approach, experimental results show the possibility of applying it to architectures that are used for semantic segmentation tasks without significant performance degradation ($\approx 2.3\times$ parameter reduction leads to $\approx 3\%$ in the mean Intersection over Union (mIoU)).

To show the efficiency of our method on the hardware, we implement the BSG-augmented CNN models for the CIFAR-10 dataset on a small FPGA suitable for edge devices. Our experiments show that with approximately $16\times$ fewer parameters our ResNet-56 variation can achieve an accuracy that is within 1% of the original network. With $\approx 30\times$ fewer parameters, the accuracy stays within 2%. These variations lead to $27.9\times$ and $21.77\times$ energy saving for off-chip memory accesses in an accelerator architecture implemented on small size of FPGAs. Further, in case of MobileNetV2, which is a compressed network using separable filters, with our modification, we can achieve $\approx 1.5\times$ compression while having an accuracy $\approx 1\%$ of the original network, which results in 43% energy saving for off-chip memory accesses. This improvement comes from several factors: representation of slices of filters in a lower dimensional space, the hardware-aware selection of slices dimensionality and size, binarization and several bit-level optimizations of the CSG network parameters and design of customized hardware, which help to reduce the memory accesses and communication cost with negligible added computation.

5.2 Related Works

In Chapter 1, we provided an extensive review of the methods that reduce the number of parameters of CNN models. Here, we briefly mention the related techniques, their challenges and relevance of these methods to our work.

Network Pruning. To improve the inference time of CNN models, pruning the network parameters and network connections have been proposed [HPTD15a, LKD$^+$16, AHS17]. These methods, however, are only applicable to the networks after the training phase of the original network with original number of parameters. Extra training and fine tuning are also required to recover the accuracy degradation. Other pruning-based techniques that modify the training phase such as [YYX+19, FC19] are also available, however, they either do not reduce the number of parameters during the training phase or they significantly add to the computational burden and hence are not desired for distributed training.

Knowledge Distillation (KD). An example of such methods is introduced in [CCY+17], which focuses on reducing the number of network parameters, both during the training and inference phases. However, these techniques assume that the parameters of the original network are readily available.

Quantization. These methods are among the very successful methods for reducing the computational burden of CNNs that can be used during both training and inference phases [HCS+17, KWW+17, RORF16]. Excessive quantization of gradients in the training phase, however, can lead to significant reduction in the model's accuracy [RORF16]. In addition, the necessity of performing some optimizations and elaborate modifications to the model and their training process to achieve high accuracy makes these methods not easy to implement.

Efficient Fast-Fourier-Transform (FFT). These types of approaches exploit the computational efficiency of FFT-based multiplications [ASKM18, DLW+17]. To be useful, these schemes require complex multiplications and efficient implementations of FFT. There are also methods based on the Winograd algorithm [Win80] for performing efficient convolutions in the real domain [LG16]. We note that these approaches to compress and accelerate operations in the fully connected layers or to accelerate the convolution operations can yield additional gains when combined with our method.

Parallel Training and Gradient Compression. These methods are concerned with performing different stages of the training in parallel, or to reduce the amount of information that needs to be communicated between different nodes of the distributed computation network using compression, or quantization of the gradients [WSL+18, LHM+18, YA18, LAP+14, RRWN11, WWLZ18]. However, these works are not concerned with the architecture of the network or on how the filters are designed, and can be applied to any architecture including our CSG-augmented CNNs. While some of these methods provide lossless compression for the gradients, others lead to significant accuracy loss or need elaborate modifications.

Neural Architecture Search (NAS). Methods proposed in [GLL19] and [WLF17] require the training of many architectures to find a well performing architecture. Custom designed networks such as [HZC$^+$17, ZZLS18] are tiny networks for mobile devices and focus on reducing the complexity of 1×1 convolutions and hence they do not provide general plug-and-play methods for reducing the number of parameters during the training of a given full architecture.

Structured Convolutional Filters. these approaches have been explored at the intersection of signal processing and computer vision. In [JvGLS16] the authors, inspired by scattering networks [SM13, BM13, Mal12], introduce a structured method based on the family of Gaussian filters and its smooth derivatives, to produce the CNN filters from basis functions that are learned during the training phase. **Steerable filter** design has been studied for about three decades [FA91].

Low-Rank Tensor Decomposition. Another set of approaches such as Canonical Polyadic (CP) decomposition [LGR$^+$15], Singular Value Decomposition (SVD) based methods [TXZ$^+$15], [JVZ14], and separable filter [MG12] focus on finding a low-rank decomposition of the filters in order to achieve a network with improved inference time. However, all these methods except separable filters [MG12] require training on the full set of parameters. In addition, compute-heavy decomposition methods such as [TXZ$^+$15] significantly slow down the training of CNNs. Therefore, due to their inability to reduce the number of trainable parameters and significantly slower training time for some of them, they are not comparable with our approach and are not included in our comparisons.

Transferred Convolutional Filters. Methods such as [SSAL16, CW16], on the other hand, exploit the equivariant group theory and they reduce the number of trainable parameters and accelerate the training. For instance, [RSLF13] show that multiple image filters can be approximated by a shared set of separable (rank-1) filters, and the authors in [SSAL16, CW16] reuse the filters, allowing large speedups with minimal loss in accuracy.

The difference between our approach and methods such as separable filters and transferred convolutional filters is two folds. First, we use a single network (CSG) to generate all the convolutional filters of the entire neural network. Second, we reproduce these filters by approximating slices that expand across multiple filters. Later in this chapter, We provide an experimental comparison of our approach with these methods in terms of compression ratio and the impact on the accuracy.

5.3 Preliminaries

5.3.1 Convolutional Neural Network (CNN)

In a typical classification task, a CNN is composed of several convolutional layers and one or more fully connected layers, at the very end of the network, responsible for the classification. Each convolutional layer consists of a set of filters and perhaps is followed by some batch normalization layers and activation layers. Our goal is to reduce the number of these trainable parameters by providing a compact representation for the parameters of the sets of filters of the convolutional layers.

Let $l \in R^{n_f \times k \times h \times w}$, for $n_f, k, h, w \in N$, denote a set of n_f filters in the CNN, where k is the number of input channels and h and w are the height and width of the kernel, respectively. Let denote the collection of all the sets of filters in a CNN, namely the main parameters of the convolutional layers, by $\mathcal{L}$ and the set of all the other parameters in the CNN by O. Then, we represent the set of all the parameters by $\mathcal{P} := \mathcal{L} \cup O$.

5.3.2 Slices

Instead of focusing on direct compression of filters, in this work we focus on slices. We define a *slice* as a tensor $s \in R^{\hat{n}_f \times \hat{k} \times \hat{h} \times \hat{w}}$, for $\hat{n}_f, \hat{k}, \hat{h}, \hat{w} \in N$. We partition each set of filters $l \in \mathcal{L} \setminus \{l_0\}$, where l_0 denotes the set of filters of the first convolutional layer, into $\lceil n_f/\hat{n}_f \rceil \lceil k/\hat{k} \rceil \lceil h/\hat{h} \rceil \lceil w/\hat{w} \rceil$ slices starting from the first slice $l(0:\hat{n_f}, 0:\hat{k}, 0:\hat{h}, 0:\hat{w})$. In Fig. 5.1, one slice of a set of filters of a convolutional layer is shown in darker color. We denote the set of all such slices for all layers by $\mathcal{S}$. The ordering of these partitions is arbitrary and does not affect the final results. Without loss of generality we assume that this partitioning is possible. [1] To reduce the trainable parameters, we produce same size but approximate versions of these slices denoted by $\hat{s} \in \hat{\mathcal{S}}$ from a compact low dimensional space (codes) using the CSG as explained in detail in the next section.

5.3.3 Code Vectors

To approximate each slice of each set of filters $s \in \mathcal{S}$ by $\hat{s} \in \hat{\mathcal{S}}$ using the CSG, we use a code vector $c \in R^{n_c}$, where $n_c \in N$. The relationship between slices and their corresponding code vectors is detailed in the following section.

5.4 Convolutional Slice Generator

The Convolutional Slice Generator (CSG) provides a linear approximation for the slices of a convolutional filter. This means that each slice of a set of convolutional filters is represented by a code vector that has around $20\times$ fewer elements. Multiplying the CSG matrix by this code vector, followed by an appropriate reshaping, produces an approximation for this slice. Several slices are then concatenated to produce a regular but approximate version of the set of convolutional filters.

[1]In practice we consider additional slices for fractional partitions and only use part of the final slice(s) to reconstruct the set of convolutional filters.

The shared CSG matrix used by all layers provides the association among the convolutional layers. In the following sections we make these statements precise.

5.4.1 The CSG Network

To generate an approximate version of each slice s_i, for $i \in \{1,...,|\mathcal{S}|\}$ denoted by $\hat{s}_i$, we have

$$\hat{s}_i = Reshape(A_{CSG}c_i), \quad \text{for} \quad i \in \{1,...,|\hat{\mathcal{S}}|\}, \tag{5.1}$$

where A_{CSG} denotes an $\hat{n}_f\hat{k}\hat{h}\hat{w}$ by n_c matrix representing the weights of the CSG network, c_i denotes the code vector corresponding to the i'th slice where $i \in \{1,2,...,|\hat{\mathcal{S}}|\}$, and the $Reshape(.)$ operator reshapes the input vector to a tensor of dimensions $\hat{n}_f, \hat{k}, \hat{h}, \hat{w}$ in an arbitrary but consistent order. See Fig. 5.1 for an example of how a single slice of a set of filters for a convolutional layer is generated.

5.4.2 Training the CSG-Augmented Network

Let $\hat{\mathcal{G}}$ denote the parameters of the CSG, i.e., the elements of the matrix A_{CSG}, $\hat{\mathcal{C}}$ denote the set of all the code vectors, and let $\hat{\mathcal{O}} = \mathcal{O}$ denote all the other parameters of the CNN, e.g., biases, batch normalization parameters, fully connected layer(s), and the first convolutional layer filters. Hence, we can denote the set of all the parameters of the network by $\hat{\mathcal{P}} := \hat{\mathcal{C}} \cup \hat{\mathcal{G}} \cup \hat{\mathcal{O}}$. Let $\mathcal{D}$ denote the set of the input data. A general objective function to train the CNN in our approach can be written as

$$f(\mathcal{D}, \hat{\mathcal{P}}) = f(\mathcal{D}, \hat{\mathcal{C}}, \hat{\mathcal{G}}, \hat{\mathcal{O}}).$$

Hence, to train the CSG-augmented CNN, instead of taking the gradients with respect to the kernels' weights ($\mathcal{L}$), they are taken with respect to the set of code vectors and the CSG parameters

$(\hat{\mathcal{C}}, \hat{\mathcal{G}})$.

5.4.3 Cardinality of the Code Vector Space

In this section, we discuss our method for providing a rough estimate on the cardinality of the code vector space n_c. First, we need to choose a shape for the slices. In order to decide about this shape, we considered several widely used CNNs including VGG16, VGG19, ResNet, etc. A 3×3 filter size is the most common size for the filters. Also, these architectures suggests that a slice with channel size of 16 and the depth of 16 would divide most of these filters. Hence, we chose $\hat{s}_1 = 16, \hat{s}_2 = 16, \hat{s}_3 = 3, \hat{s}_4 = 3$ for this part of our work.In order to determine the cardinality of the code vector space, we need an estimate of the number of the elements of the slice in its possible latent domain, namely an estimate for n_c. Inspired by the fact that these filters are responsible for detecting visual features and knowing that usage of DCT leads to a very good encoding of visual representations [Wat94], we looked at the four-dimensional Type-II DCTs (4-D DCT-II) of about 29000 slices of pre-trained filters extracted from VGG-16, VGG-19, ResNet-50, InceptionV3, DenseNet-169, DenseNet-201, InceptionResNetV2 (available in Tensorflow). We then computed the 4-D DCT-II representation of these slices and removed the elements of this representation in such a way that the remaining elements would result in an inverse transform which is not very different from the original slice. Our analysis, presented below, suggests that a code vector that has close to $20\times$ fewer number of elements would be sufficient. In our experiments, we chose code vectors that have $18\times$ fewer elements than the slices, and our experiments on the neural networks confirm this choice.

To determine the size of code vectors, we first take the 4-D DCT-II of each slice. The 4-D

DCT-II that we use, after removing the scaling factors, is stated as follows.

$$\begin{aligned} K[u,v,w,t] := \sum_{i=0}^{\hat{s}_1-1}\sum_{j=0}^{\hat{s}_2-1}\sum_{k=0}^{\hat{s}_3-1}\sum_{l=0}^{\hat{s}_4-1} & k[i,j,k,l]\times \\ & \cos\left(\frac{\pi}{\hat{s}_1}\left(i+\frac{1}{2}\right)u\right)\times\cos\left(\frac{\pi}{\hat{s}_2}\left(j+\frac{1}{2}\right)v\right)\times \\ & \cos\left(\frac{\pi}{\hat{s}_3}\left(k+\frac{1}{2}\right)w\right)\times\cos\left(\frac{\pi}{\hat{s}_4}\left(l+\frac{1}{2}\right)t\right) \end{aligned}$$

After taking the 4-D DCT-II, we then remove the elements of the slice in the transformed domain that were smaller than a threshold. We then took the inverse transform. The inverse 4-D DCT transform, after neglecting its scaling factors, can be stated as follows.

$$\begin{aligned} K[i,j,k,l] := \sum_{u=0}^{\hat{s}_1-1}\sum_{v=0}^{\hat{s}_2-1}\sum_{w=0}^{\hat{s}_3-1}\sum_{t=0}^{\hat{s}_4-1} & k[u,v,w,t]\times \\ & \cos\left(\frac{\pi}{\hat{s}_1}\left(u+\frac{1}{2}\right)u\right)\times\cos\left(\frac{\pi}{\hat{s}_2}\left(v+\frac{1}{2}\right)v\right)\times \\ & \cos\left(\frac{\pi}{\hat{s}_3}\left(w+\frac{1}{2}\right)w\right)\times\cos\left(\frac{\pi}{\hat{s}_4}\left(t+\frac{1}{2}\right)t\right) \end{aligned}$$

In order to measure the similarity between the inverse transformed version of the slice and the original slice, inspired by image compression similarity measures, we use a variation of a known measure called PSNR [Wel99] which we define as follows. Let $\hat{k}^*$ denote the inverse DCT of the pruned DCT of the slice $\hat{k}$. We re-scale the elements of the slices and their corresponding approximate version to $[0,1]$ with a bit of abuse of notation we represent the re-scaled versions with the same notations.

$$PSNR^* = 10\log\frac{1^2}{MSE}, \tag{5.2}$$

where

$$MSE = \frac{1}{\hat{s}_1\hat{s}_2\hat{s}_3\hat{s}_4}||\hat{k} - \hat{k}^*||_2^2. \tag{5.3}$$

We chose the threshold for keeping the elements in the DCT domain such that the average PSNR* is above 20dB which from image compression literature is expected to result in images that are still recognizable (see, for instance [Vel10]). We then calculated the mean of the number of remaining elements in the DCT domain after the pruning step. This suggests that a code size of 20 times fewer elements than its corresponding slice would be sufficient.

5.4.4 Training Convergence

While convergence is always observed in all our experiments, in this section, we provide a proof of convergence for a simple CSG-augmented CNN with only one convolutional layer based on the recent work [AZLS19]. Let m denote the number of channels of the input, and d denote the number of its features (e.g., pixels). For simplicity, let us assume that the number of channels remains m after the convolutional layer. Let n denote the number of data points, and d' denote the number of labels. We assume that the data-set is non-degenerate meaning that there does not exist similar inputs with dissimilar labels. We denote by δ the minimum distance between two training points. We restate the following theorem from [AZLS19] for the CSG-augmented CNNs.

Theorem 1 (CNN [AZLS19]) *As long as $m \geq \tilde{\Omega}(poly(n,d,\delta^{-1})d')$, with a probability that approaches one as $m \to \infty$, Stochastic Gradient Decent (SGD) finds an ε-error solution for l_2 regression in $T = \tilde{\Omega}\left(\frac{poly(n,d)}{\delta^2}\log \varepsilon^{-1}\right)$ iterations for a CNN.*

The above theorem as discussed in [AZLS19] can be easily extended for other convergence criteria including the cross-entropy. Now let us consider our CSG-augmented CNN which we denote by CNN-CSG. For simplicity, in the following theorem, we consider the case when only a single layer convolutional layer is present.

Theorem 2 (CNN-CSG) *If $|\hat{C}| \geq \tilde{\Omega}(poly(n,d,\delta^{-1})d')$, with a probability that approaches one as $|\hat{C}| \to \infty$, then SGD finds an ε-error solution for l_2 regression in $T = \tilde{\Omega}\left(\frac{poly(n,d)}{\delta^2}\log\varepsilon^{-1}\right)$ iterations for a CNN-CSG.*

The proof of the above theorem, which follows from the fact that the code vectors following the CSG layer can simply be viewed as an additional fully connected layer is provided below. Similar to Theorem 1, Theorem 2 can be easily extended for other convergence criteria including the cross-entropy.

Proof of Theorem 2 First of all we note that since the number of weights in the convolutional layer is a polynomial function of $|\hat{C}|$, it has replaced the m in Theorem 2. Now, let

$$C = \left[c_1, ..., c_{|\hat{C}|}\right], \tag{5.4}$$

denote a matrix whose columns are the code vectors corresponding to the slices of the convolutional layer.

Now, instead of assuming that the convolutional filter is first generated and then it is used for the convolution operation, equivalently, using associativity, we can assume that each column of the matrix A_{CSG} denotes a vectorized version of a slice of a convolutional filter. It means that, for each column of A_{CSG}, we need to calculate the convolution of a slice for its $|\hat{C}|$ possible locations in the filter. But each of these would be an ordinary convolution with appropriate zero-paddings. Now, the matrix C can be viewed as an additional fully connected layer before the final classification layers. Hence we are dealing with a CNN with an additional fully connected layer at the final stage for which the results in [AZLS19] and specially Theorem 1 hold.

5.5 Binarized Slice Generator (BSG)

In this section, we explain the optimized CSG method that is specifically designed to improve the memory accesses during inference phase when a given CNN is executed on small sizes of embedded devices for edge applications.

To optimize CSG on edge devices during the inference phase, we deploy several optimization methods such as binarization of CSG parameters, permutations and bit level manipulations.

5.5.1 Binarization

In order to further reduce off-chip and on-chip memory accesses during inference phase, instead of generating the slices of convolutional filters from floating point CSG matrix and code vectors, we use binary CSG matrix and code vectors. We called this method as Binarized Slice Generator (BSG). In order to maintain the classification accuracy, the binarized CSG matrix, called A_{BSG} and the binary code vectors are trained and specified as follows. In the beginning of training, the elements of A_{BSG} are selected uniformly at random from $\{-1,+1\}$ and fixated. To train the code vectors, their elements are trained in floating point format during the backward pass; however, in the forward pass, the elements of code vectors are set to their sign. Therefore, after training the BSG-augmented CNN models, the code vectors are set to either -1 or $+1$.

5.5.2 Permutation

To generate each slice of convolutional filter during inference on the underlying hardware, the whole BSG matrix is required. Although binarization of A_{BSG} helps to reduce memory accesses, we can further reduce the memory accesses and the required on-chip memory by providing further compression of A_{BSG} through generating BSG matrix from a single binary vector. The binary vector can be then circularly shifted each time to generate the corresponding part of the BSG matrix that is required to generate a weight of the slice. Therefore, A_{BSG} is represented by a vector of size n_c, which is equal to the size of the code vectors. The process of generating the weights are as follows.

In the beginning of the inference phase, only a binary vector of size n_c is brought on-chip that is a representative of A_{BSG}. For generating one slice of convolutional filters, one binary code vector of size n_c is loaded on-chip. Then, to generate $\hat{n}_f\hat{k}\hat{h}\hat{w}$ weights, the binary vector corresponding to A_{BSG} is shifted circularly and multiplied by the code vector available on-chip.

5.5.3 Bit-Level Manipulation

To generate i-th weight of slice s of a layer l, the inner product of $i-th$ circular shift of the binary vector corresponding to A_{BSG} shown as (BSG^i) by the code vector corresponding to the slice of the layer l that the weight belongs to (shown as CV[l][s]) is performed according to Equation 5.5.

$$W_{i,s}^l = \sum_{v=1}^{n_c} (BSG^i)[v] \times CV[l][s][v] \tag{5.5}$$

However, due to the binary representation of BSG parameters and the code vectors, with binary values from $\{-1,+1\}$, the inner product of the two vectors in BSG can be replaced by simpler operations. For hardware efficiency, we represent $+1$ by 1 and -1 by 0. Therefore, instead of multiplications in the inner product, the elements of two vectors of n_c binary values are now required to be XORed and the final output will be the difference between the number of zeros and ones. The process is shown in Equation 5.6, where *count_ones* is a function counting the number of ones of its binary input vector and *count_zeros* counts the number of zeros.

$$\begin{aligned} BCX &= CV[l][s] \oplus BSG^i \\ W_{i,s}^l &= count\_ones(BCX) - count\_zeros(BCX) \end{aligned} \tag{5.6}$$

5.6 Accelerator Architecture for BSG-Augmented CNNs

This section details the accelerator architecture called BSG-augmented accelerator to support the efficient inference of CNNs augmented by BSG on FPGA devices.

5.6.1 Hardware Implementation of BSG

To efficiently implement the process of generating weights of a slice of the convolutional filters in a layer through BSG, we design a module called BSG at the hardware level that execute the process shown in Equation 5.6.

To minimize the latency of computing the weights by BSG, all n_c XORs are parallelized and the function ($count\_ones$) is implemented by an adder tree with $log_2(n_c)$ levels. For area and latency efficiency, the bit-width of adders in each level of the adder tree is increased by one from top to the bottom. The hardware implementation of this process is shown in Figure 5.2a, where the elements of two code vector and BSG vector are XORed and the number of ones in the output are counted.

Finally, the number of zeros can be computed by subtracting the number of ones from n_c, which is the length of the vector. It should be noted that to generate each weight, the corresponding vector of A_{BSG} that is generated by proper number of circular shifts of the base vector is required to be passed to the BSG module.

5.6.2 BSG-based Architectural Modification of Accelerators

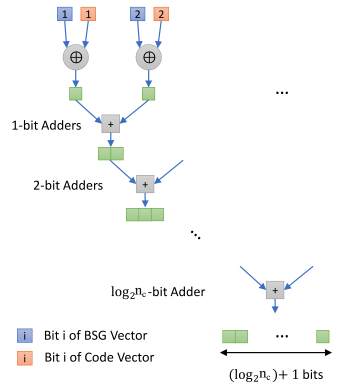
We now consider the baseline FPGA based accelerator architecture and the modification applied to the baseline architecture to support the execution of BSG-augmented CNNs on FPGA during inference. Figure 5.2b illustrates the baseline accelerator architecture augmented by BSG.

Baseline Architecture

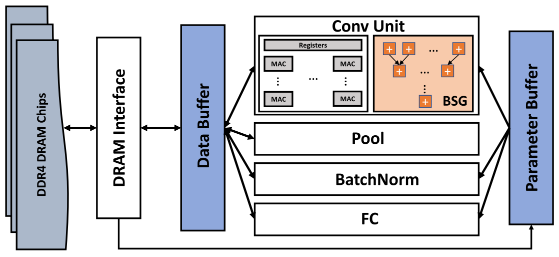
In general, the CNN accelerator architecture consists of on-chip buffers to store on-chip data, processing units for executing different types of layers in CNNs, and memory interface to access the main memory. On-chip buffers can be used to implement two types of buffers: Parameter buffer to store the parameters of CNN models and Data buffer to store input and output feature maps of each layer as well as intermediate data of the processing units. Convolution processing unit (Conv) consisting of several MAC units performs the convolution operations in convolutional layers of CNN models. Pool, BatchNorm and FC are the processing units responsible for executing the pooling, batch normalization and fully connected layers, respectively. Due to the limited resources of FPGA platforms, to minimize off-chip memory accesses the following dataflow for executing a given CNN is generally used.

An input image of shape ($c_{in} \times W \times H$) is partitioned vertically and horizontally and each partition (with the shape of $c_{in} \times w \times h$) is brought onto the on-chip buffer for processing. The layers of a CNN models are also grouped so that each group contains a number of consequent layers. The process starts with the first image partition and loads it onto the data buffer and the operations related to the first group of layers are performed on that partition in sequence. To perform the operations corresponding to a layer in a group, the related parameters are loaded onto the parameter buffer and the operations are performed on the data available in data buffer. While the outputs of layers within a group except those of last layer in the group are stored on the on-chip data buffer to be accessed immediately by their next layers, the output of last layer in the group is stored in the main memory. This process is repeated until the results of all the input partitions related to the first group of layers are generated. Then, the inputs of the next group of layers are similarly partitioned and loaded onto the chip to be processed by the corresponding layers. The process continues until the whole CNN is executed on the entire input image.

Here, since BSG impacts only the convolutional layers, we explain the implementation of convolutional layers on FPGA. According to the dataflow described above, for each input



(a) The hardware implementation of XORing BSG vector and code vector and the function *count_ones* used in Equation 5.6 to generate the weights in BSG



(b) BSG-augmented accelerator architecture

Figure 5.2: a) The hardware implementation of BSG and b) the BSG-augmented accelerator architecture

partition, the parameters of a set of layers in a group are required to be loaded on-chip. However, due to limited on-chip buffer, the parameters of a convolutional layer of a regular CNN are partitioned into a set of slices and one partition is loaded on-chip at a time. The partition remains on-chip until all the related operations on the available on-chip data are performed. Inside the convolution unit, which consists of $m \times n$ MAC units, the operations of several convolutional windows are parallelized by dividing filters among the rows and input feature maps among the columns. MAC units on each row have access to the same filters and on each column have access to the same input feature maps. Therefore, at each cycle, MAC units on a row, multiply the same weight of their assigned filter by their own corresponding input feature map.

BSG-Augmented Architecture

To support BSG during inference, the BSG module, which performs the operations in Equation 5.6 with the aid of the module shown in Figure 5.2a to perform XOR and *count_one*, is integrated into the convolutional unit on the accelerator. Before performing the convolutions, the parameters in one slice is required to be generated by BSG in the order required by the convolution operations. To generate the slice, one code vector and one vector corresponding to the BSG matrix is required. As mentioned earlier, the whole BSG matrix can be generated by a set of circular right-shift of the base BSG vector. The vector corresponding to the BSG matrix is brought on-chip once and remains on-chip until the end of the CNN execution. Since it is binarized, we packed its bits and stored it as an array of $\frac{n_c}{16}$ 16-bit values. Since the weights are generated one by one in sequence, each time the base BSG vector is shifted one bit to the right and sent to BSG module to generate the weight. Unlike the vector for the BSG matrix, the code vectors are required to be loaded on-chip whenever their corresponding slice is required for performing convolutions. Similarly, since the code vectors are binarized, each can be represented as $\frac{n_c}{16}$ 16-bit values.

The BSG-augmented architecture can lead to better energy efficiency due to the following reasons. 1) Unlike the baseline architecture, BSG-augmented architecture utilizes less on-chip buffer for each slice. 2) While in baseline architecture, one slice is required to be loaded for each input partition separately, due to fewer BSG related parameters, in BSG augmented architecture, with similar sizes of on-chip buffer, the BSG parameters corresponding to more slices can be brought on-chip and reused until the whole input of a group of layers is processed. Therefore, due to higher reuse distance, the off-chip memory accesses will be reduced significantly depending on the vector size and the shape of slices. We evaluate the hardware efficiency of our method, which is provided in the next section.

5.7 Experiments

We evaluated our approach on five different CNN models (ResNet-56, DenseNet-BC-40-48, DenseNet-BC-40-36, ShuffleNet V2, MobileNet V2) on CIFAR-10 dataset, two CNN models (ResNet-50 and ResNet-101) on ImageNet-1000 dataset, and two Deeplab models on Pascal VOC dataset. The CSGs are integrated into the models implemented in PyTorch. For training the models on the CIFAR-10 dataset, we used a machine with a single GPU (Nvidia Geforce 2080 Ti) and

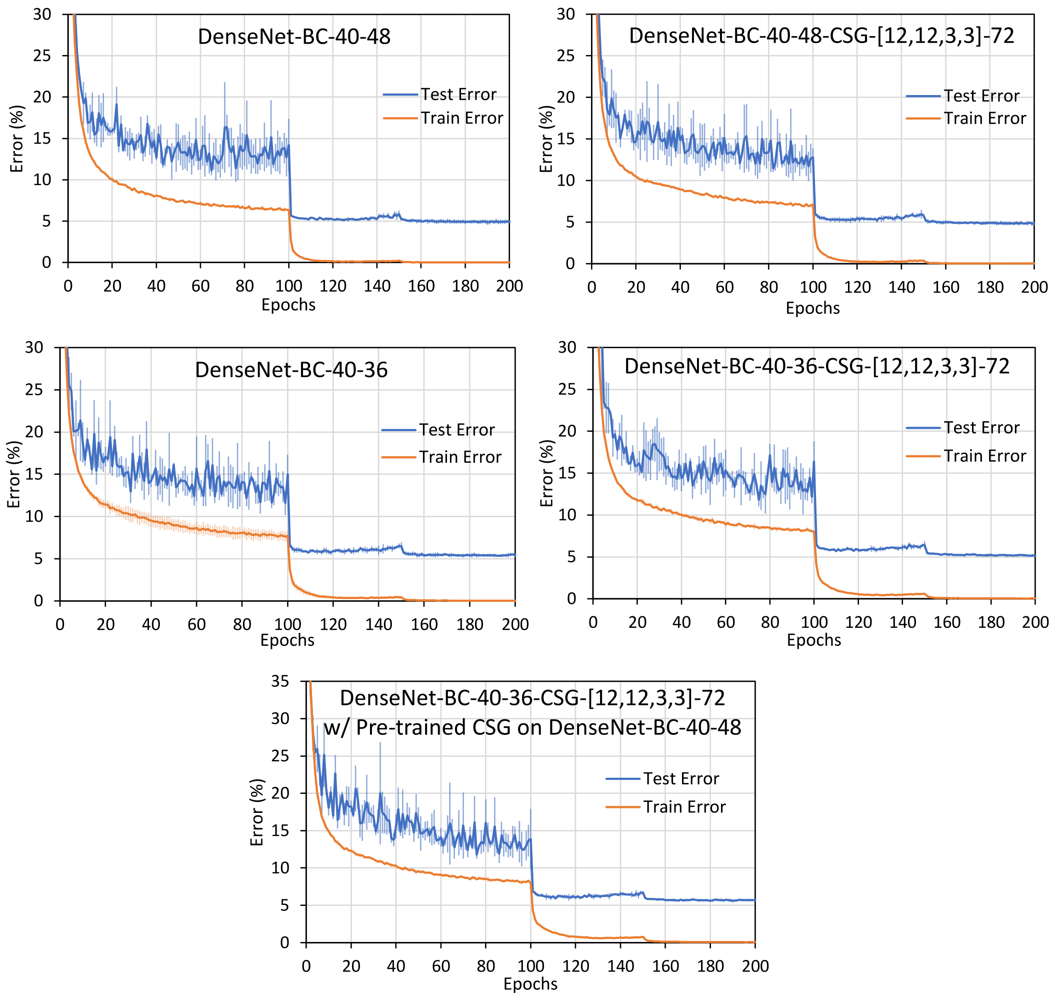


Figure 5.3: Training and test error for DenseNet-BC-40-48, DenseNet-BC-40-36 and their CSG-augmented versions on CIFAR-10 dataset over the course of 200 epochs with batch size of 128. For the first 100 epochs the learning rate was 0.05 and for the two final 50 epochs, 5×10^{-3} and 5×10^{-4}, respectively.

for the training on the two other datasets we used four (NVidia Geforce 1080 Ti) GPUs. Note that we did not do any parameter tuning for any of our CSG-augmented networks and the experiments are all done using the same settings that were used for the original networks. Also, as it is clear from the previous sections, we did not apply our method to the very first convolutional layer.

Table 5.1: Training results on CIFAR-10 dataset with similar hyperparameters. When CSG is used, the slice shape and the code vector size are indicated as CSG-[$\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4$]-$n_c$ following the name of the original network. In the "Top-1 Err." column the average and standard deviations of test errors at the last epoch for three non-selective trainings and on the "Ratio" column the compression ratios with respect to the original networks are reported.

Network Architecture	**# Param.**	**Top-1 Err.**	**Ratio**
DenseNet-BC-40-48 (Original)	2,733,130	4.97 ± 0.26	1.00×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72	1,416,394	4.83 ± 0.24	1.92×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72 w/ Pre-trained CSG on DenseNet-BC-40-48	1,323,082	5.07 ± 0.11	2.06×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72 w/ Pre-trained CSG on DenseNet-BC-40-36	1,323,082	5.14 ± 0.23	2.06×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72 w/ Compressed 1x1 Kernels	904,906	5.62 ± 0.28	3.02×
DenseNet-BC-40-36 (Original)	1,542,682	5.38 ± 0.27	1.00×
DenseNet-BC-40-36-CSG-[12,12,3,3]-72	842,842	5.12 ± 0.09	1.83×
DenseNet-BC-40-36-CSG-[12,12,3,3]-72 w/ Pre-trained CSG on DenseNet-BC-40-48	749,530	5.61 ± 0.21	2.05×
ResNet-56 (Original)	853,018	6.28 ± 0.20	1.00×
ResNet-56-CSG-[16,16,3,3]-128	347,162	7.24 ± 0.11	2.45×
ResNet-56-CSG-[12,12,3,3]-72	160,450	8.01 ± 0.27	5.31×
ResNet-56-CSG-[16,16,3,3]-128 w/ Pre-trained CSG on ResNet-20	52,250	11.98 ± 0.28	16.3×
ShuffleNet-(0.5x) (Original)	352,042	9.81 ± 0.23	1.00×
ShuffleNet-(0.5x)-CSG-[16,16,1,1]-16	171,818	10.15 ± 0.16	2.04×
MobileNetV2 (Original)	2,296,922	6.64 ± 0.18	1.00×
MobileNetV2-CSG-[16,16,1,1]-16	1,595,322	7.65 ± 0.18	1.44×

5.7.1 CSG on CIFAR-10 Dataset

CIFAR-10 dataset includes 50K training images and 10K test images from 10 different classes. See Table 5.1 for a summary of the results. As we can see, when we used $[16, 16, 3, 3]$ slices and code vectors of size 128 for ResNet-56 [HZRS16b], we achieved $\approx 2.5\times$ reduction with less than 1% increase in top-1 error. If we allow a higher accuracy degradation of $\approx 1.5\%$, we can achieve over $5.3\times$ parameter reduction by using $[12,12,3,3]$ slices and code vectors of size 72. In case of DenseNet [HLVDMW17], we considered the most challenging cases, namely, when bottlenecks are used and the network has a 50% compression factor (i.e., $\theta = 0.5$), which is abbreviated as DenseNet-BC. We only considered 3×3 kernels and did not compress the bottleneck or transition layers in these implementations. Since the number of filters is a multiple of 12, we chose slices of shape $[12,12,3,3]$ and code size of 72 to keep the ratio between the number of elements in the slice and code vector size n_c the same. We considered two cases when $L = 40, K = 48$, and $L = 40, K = 36$, where L is the number of layers and K is the growth rate. For the first case, we could achieve $\approx 2\times$ reduction with a slight improvement in accuracy. For the second case, the use of CSG had little effect on the accuracy of the network while reducing its parameters by over $1.8\times$. For ShuffleNet V2 (CIFAR version) we have compressed the last convolutional layer of the networks which is again a 1x1 kernel and constitutes a large portion of the network weights resulting in $\approx 2\times$ compression while the accuracy loss is within 0.5%. For the case of MobileNetV2 (CIFAR version) we have compressed the first 1x1 kernel in each block (namely kernels w/ largest number of parameters) resulting in $1.44\times$ reduction with an accuracy loss of 1%.

The training and test error of different models for CIFAR-10 dataset and their CSG-augmented versions reported in Table 5.1 at each epoch over 200 epochs are presented in figures 5.3 through 5.6. These errors show that training of the CSG-augmented models is converged and over-fitting does not occur. In addition, we can see that the trend of training and test error in CSG-augmented models is similar to the original models, which indicates the applicability of our approach to other CNN models that have not studied here in this dissertation.

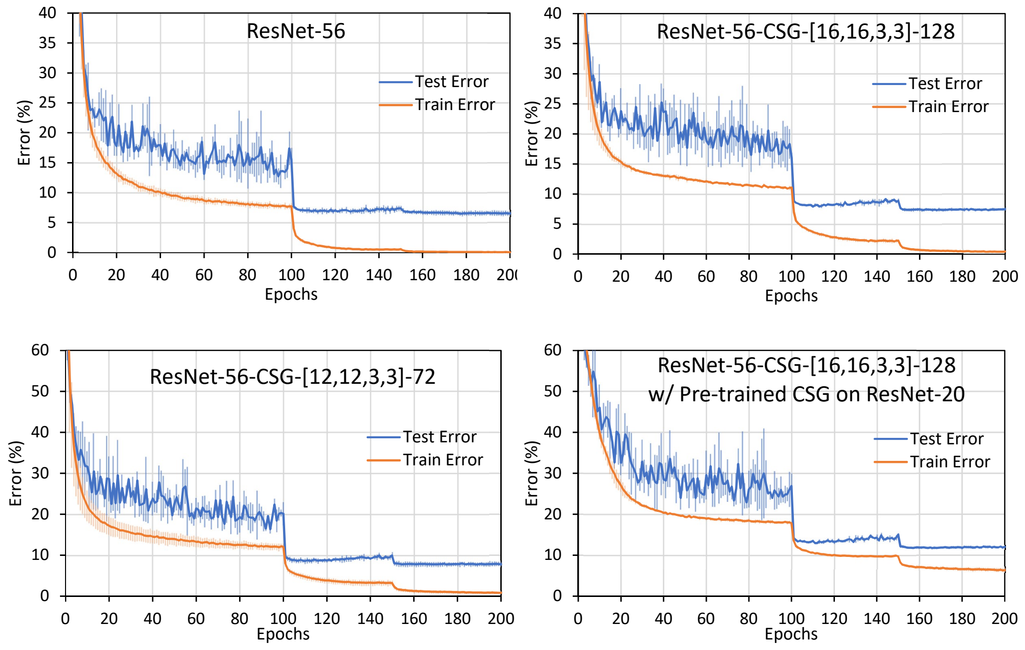


Figure 5.4: Training and test error for the ResNet-56 and its CSG-augmented versions over the course of 200 epochs. The batch size was 128. For the first 100 epochs the learning rate was 0.05 and for the two final 50 epochs, 5×10^{-3} and 5×10^{-4} respectively.

5.7.2 CSG on ImageNet-1000 (ILSVRC2012) Dataset

We trained the CSG-augmented versions of ResNet-50 and ResNet-101 on the ImageNet-1000 (ILSVRC2012) dataset which consists of ≈1.3 million images for training and 50K images for validation. We used the same hyper-parameters as the ones mentioned in the original paper [HZRS16a], namely we used batch sizes of 256 images, and started from the learning rate of 0.1 and divided the learning rate by 10 every 30 epochs. We continued the training for 100 epochs (which is 20 epochs less than the original paper).

The number of parameters and the Top-1 error for the original model and the CSG-augmented ones and the corresponding compression ratios are summarized in Table 5.2. The results for ResNet-50 show that While ResNet-50-CSG-[16,16,3,3]-128 has a compression ratio of 1.68×, it achieves a top-1 error of 24.9% which is within 1% of the error of the original ResNet-50 implemented in PyTorch and reported by TorchVision [Tor30].

ResNet-101-CSG-[16,16,3,3]-128 achieves 23.1% top-1 error with a compression ratio of

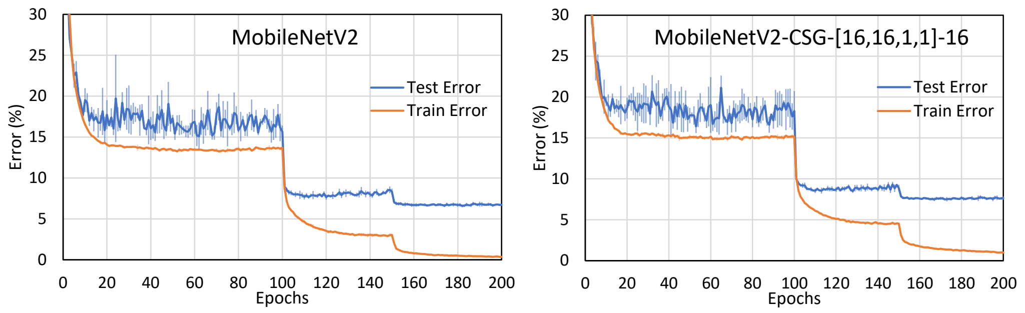


Figure 5.5: Training and test error for MobileNet V2 (CIFAR version) and its CSG-augmented version over the course of 200 epochs. The batch size was 128. For the first 100 epochs the learning rate was 0.05 and for the two final 50 epochs, 5×10^{-3} and 5×10^{-4} respectively.

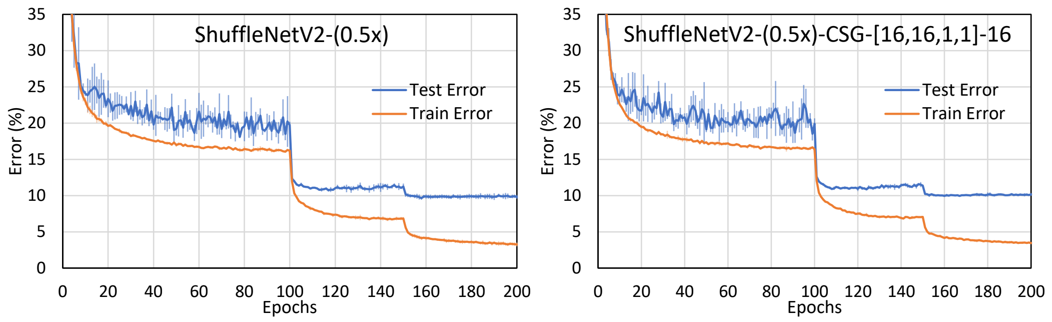


Figure 5.6: Training and test error for ShuffleNet V2 (0.5x) (CIFAR Version) and its CSG-augmented version over the course of 200 epochs. The batch size was 128. For the first 100 epochs the learning rate was 0.05 and for the two final 50 epochs, 5×10^{-3} and 5×10^{-4} respectively.

$1.81\times$. While this model now has around one million parameters less than original ResNet-50, its error-rate is still $\approx 0.8\%$ smaller. The results are summarized in Table 5.2. More details of training and validation errors over the course of 100 epochs are shown in Fig. 5.7.

5.7.3 Training the CSG alongside the CNN

In this set of experiments we train all the models from scratch. We initialize the parameters of CSG $\hat{\mathcal{G}}$ with random initial values and train it alongside the code vectors $\hat{\mathcal{C}}$ as well as other parameters of the network $\hat{\mathcal{O}}$. We refer the reader to the supplamentary material for a detail account of the datasets and implementation as well as convergence figures.

Table 5.2: Training results on ImageNet-1000 (ILSVRC2012) dataset. When CSG is used, the slice shape and the code vector size are indicated as CSG-$[\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4]$-$n_c$ following the name of the original network. In the "Top-1 Error" column the validation error for the center cropped images at the last epoch for the training and on the "Ratio" column the compression ratios with respect to the original networks are reported. The results indicated with a "*" are reported from [Tor30]

Network Architecture	# Param.	Top-1 Err.	Ratio
ResNet-50 (Original)	25,557,032	23.9*%	1.00×
ResNet-50-CSG-[16,16,3,3]-128	15,163,432	24.9%	1.68×
ResNet-101 (Original)	44,654,504	22.6*%	1.00×
ResNet-101-CSG-[16,16,3,3]-128	24,685,608	23.1%	1.81×

5.7.4 Using Pre-Trained CSG

When using pre-trained CSG parameters during the training of the CSG-augmented CNNs, the number of parameters to be trained reduces to $|\hat{C}| + |\hat{O}|$. This can result in significant reduction in the number of the parameters of the network depending on its architecture. We employed pre-trained CSGs to train DenseNet-40-48, DenseNet-40-36 and ResNet-56 models from CSG-augmented DenseNet-40-36, DenseNet-40-48 and ResNet-20, respectively. The summary of our results are provided in Table 5.1 and figures 5.3 and 5.4.

5.7.5 CSG for Semantic Segmentation Tasks

To explore the possibility of applying our approach to tasks other than image classification, in these experiments we applied it to Deeplab V3 [CPK+17] for semantic segmentation on the Pascal VOC 2012 [EVGW+10] dataset *without any hyper-parameter tuning*. We considered the Deeplab V3 architecture alongside Resnet-50 and Resnet-101 for feature extraction. For each

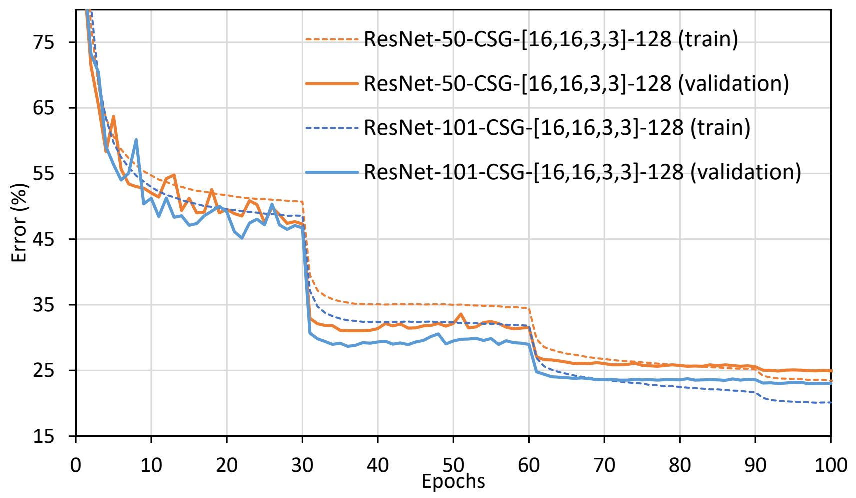


Figure 5.7: Train and validation errors of ResNet-50-CSG-[16,16,3,3]-128, and ResNet-101-CSG-[16,16,3,3]-128 on ImageNet dataset.

case, two scenarios were studied against two baseline scenarios where no modification to the models were done. The baseline scenarios in our implementation achieve mean Intersection Over Unions (mIOUs) of 73.38% and 74.73% respectively.

In the first scenario, we used Resnet-50-CSG-[16,16,3,3]-128 and Resnet-101-CSG-[16,16,3,3]-128 for the feature extraction with pretrained parameters and did not modify the rest of the architecture. In this settings our approach achieves mIoUs of 71.41% and 72.98% with 1.35× and 1.5× compression ratios. In the second scenario, in addition to using CSG-augmented versions of Resnet-50 and Resnet-101 for the feature extraction with pretrained parameters, we also considered a second CSG for the Atrous Spatial Pyramid Pooling (ASPP) modules. In these settings, we achieve mIoUs of 70.28% and 71.63% respectively with 2.39× and 2.24× compression ratios. The results are summarized in Table 5.3.

5.7.6 Comparison of CSG with Related Methods

In this section, we provide a detailed comparison of our proposed technique with the relevant methods, as discussed in more detail in Section 5.2: Kernel decomposition (separable filters) [JVZ14] and two transferred convolutional filters methods, CRELU [SSAL16] and G-CNN [CW16]. We applied these techniques on both wide and narrow CNN models DenseNet-BC-40-48

Table 5.3: Results of semantic segmentation on Pascal VOC 2012 validation dataset after training for 50 epochs with batch size of 4. With CSG, the slice shape and the code vector size are indicated as CSG-[$\hat{n}_f, \hat{k}, \hat{h}, \hat{w}$]-$n_c$ following the name of the original network.

Network Architecture	**# Param.**	**mIoU** (%)	**Ratio**
DeeplabV3-ResNet-50 (Original)	40,352,181	73.38%	1.00×
DeeplabV3-ResNet-50-CSG-[16,16,3,3]-128	29,958,581	71.41%	1.35×
DeeplabV3-CSG-[16,16,3,3]-128-ResNet-50-CSG-[16,16,3,3]-128	16,884,149	70.28%	2.39×
DeeplabV3-ResNet-101 (Original)	59,344,309	74.73%	1.00×
DeeplabV3-ResNet-101-CSG-[16,16,3,3]-128	39,480,757	72.98%	1.50×
DeeplabV3-CSG-[16,16,3,3]-128-ResNet-101-CSG-[16,16,3,3]-128	26,406,325	71.63%	2.24×

and ResNet-56 for CIFAR-10 dataset. Each modified model is trained three times with similar hyper parameters as in Section 5.7 and the results are summarized in Table 5.4. As shown in the table, for wide CNN models such as DenseNet-BC-40-48, the compression ratio of all methods is similar ($\approx 2\times$). While the error of the model modified with other methods is increased, the CSG-augmented model improves the error slightly. For ResNet-56 which is a narrow model, when half of the convolutional layers are replaced by separable filters or when we compress the model using the CRELU we achieve $\approx 2\times$ reduction. In both cases compared to our CSG-augmented model with even a slightly higher compression ratio, the top-1 errors are higher. The accuracy of G-CNN based model achieving compression ratio of 3.92× is still lower than the one in CSG-augmented one with 5.31× ratio. We also applied separable filters to all the layers of ResNet-56 that achieves significant parameter reduction $\approx 7\times$ at the cost of high accuracy degradation. In summary, the results show that for narrow models, the other techniques degrade accuracy which is compatible with their mentioned drawbacks in [CWZZ18], while our proposed technique does not degrade the accuracy significantly in both narrow and wide CNNs. We also train both the original and CSG-augmented model from scratch with 16-bit floating point numeric format to show the impact of quantization on their performance. The results validate that CSG can be applied on top of other

Table 5.4: Comparison of various compression techniques with our method on CIFAR-10 dataset. As discussed before, Kernel Decomposition methods such as SVD and CP, due to not reducing the number of trainable parameters, are not comparable, thus not included. The "Top-1 Err." column shows the test errors at the last epoch ± standard deviation in three runs and the "Ratio" column shows the compression ratios with respect to the original networks. Ratios decorated with a "*" denote bit-wise ratios.

Compression Method	Network Architecture	# Param.	Top-1 Err.	Ratio
CSG	DenseNet-BC-40-48-CSG-[12,12,3,3]-72	1,416,394	4.83 ± 0.24	1.92×
Low Rank Decomposition	DenseNet-BC-Separable Filters	1,441,450	5.13 ± 0.05	1.90×
Transferred Convolutions	DenseNet-BC-CReLU	1,369,210	5.39 ± 0.28	2.00×
	DenseNet-BC-GCNN (p4)	1,613,602	5.25 ± 0.29	1.60×
Quantization	DenseNet-BC-40-48-FP16	2,733,130	5.15 ± 0.05	2.00*×
CSG+Quantization	DenseNet-40-48-CSG-[12,12,3,3]-72-FP16	1,416,394	5.43 ± 0.08	3.84*×
CSG	ResNet-56-CSG-[16,16,3,3]-128	347,162	7.24 ± 0.11	2.45×
	ResNet-56-CSG-[12,12,3,3]-72	160,450	8.01 ± 0.27	5.31×
Low Rank Decomposition	ResNet-56-Separable Filters (1/2 Layers)	494,709	7.56 ± 0.45	1.72×
	ResNet-56-Separable Filters (All Layers)	117,066	8.41 ± 0.05	7.29×
Transferred Convolutions	ResNet-56-CReLU	427,066	8.27 ± 0.10	2.00×
	ResNet-56-GCNN (p4)	217,618	8.79 ± 0.31	3.92×
Quantization	ResNet-56-FP16	853,018	6.64 ± 0.35	2.00*×
CSG+Quantization	ResNet-56-CSG-[16,16,3,3]-128-FP16	347,162	7.54 ± 0.01	4.90*×

compression methods to provide higher compression ratios while keeping the accuracy acceptable.

5.7.7 BSG for Improved Inference on FPGA

For evaluating the efficacy of BSG to improve the inference efficiency of CNNs, we modify ResNet-56 and MobileNetV2 on CIFAR-10 dateset with BSG and compare its efficiency with the original model. To evaluate the impact of this approach on classification accuracy, the models and their BSG-augmented versions are implemented and trained using PyTorch from scratch with the exact same training hyperparameters in their original papers. The parameters and feature maps are converted to BFloat16 format for the inference phase. To evaluate the hardware efficiency of this approach, we implement the models and their BSG versions using the accelerator architecture for the baseline and the BSG-augmented architecture supporting operations with BFloat16 in C++ and use Xilinx Vivado HLS and Vivado Suite 2019.1 to synthesize the architectures and estimate the latency and resource utilization. We choose a small size of FPGA platform from Xilinx Zynq-7000 series (xc7z020-clg484 FPGA) as the target platform in order to evaluate the inference efficiency of BSG-augmented CNNs with the proposed architecture on the edge devices. To parallelize the operations of convolutional layers 128 MAC units are used and each performs 8-bit multiplications and additions.

To estimate the energy consumption improvement of the BSC-augmented models executed on FPGA, we use Xilinx Power Estimation tool (XPE) to estimate the power consumption using the utilized FPGA resources and use the latency reported by Vivado HLS tool. To estimate the energy consumption saving of DRAM accesses, we profiled required memory accesses by the implemented accelerator architecture and estimate the energy saving of our approach compared to the original model with DRAMPower [CWL$^+$18].

BSG for ResNet-56

For ResNet-56, all the layers except the first convolutional layer and fully connected layer are augmented by BSG. The shape of each slice is set to $(16,16,3,3)$ and different code sizes are selected in each set of experiments. The BSG matrix is represented by a binary vector of size n_c and generated by performing a set of circular shift on the FPGA, as described in Section 5.5.2.

Parameter Reduction and Accuracy- We evaluate the impact of BSG on classification accuracy and the size of parameters during training and inference phase for different sizes of the code vectors (i.e.,n_c). The accuracy, the number of parameters for both inference and training and the compression ratio during inference are summarized in Table 5.5.

As shown in the table, depending on the code vector size, the size of the model during inference is decreased by 16.3$\times$-77.2$\times$, while the accuracy is dropped by $1\% - 5\%$. Larger sizes of

Table 5.5: Accuracy and the model size of ResNet-56 and its BSG augmented version for different sizes of code vectors (n_c)

	Original	$n_c = 256$	$n_c = 512$	$n_c = 1024$	$n_c = 2048$
Accuracy	93.19%	88.14%	90.33%	91.52%	92.26%
Param. Size (Train)	853KB	99KB	193KB	382KB	759KB
Param. Size (Inference)	853KB	11KB	17KB	29KB	52KB
Comp. Ratio (Inference)	1×	77.2×	50.3×	29.7×	16.3×

the code vector due to lower compression decrease the degree of parameters approximation, thus leading to lower accuracy drop. Compared to the inference phase, the size of trainable parameters of the model during training is higher because the code vectors are trained in 32-bit floating point format and their sign is used during the forward pass. Only the base vector corresponding to the BSG matrix is binarized and fixated during training. In the most accurate case (i.e., $n_c = 2048$), the size of the trainable parameters is reduced by 11%.

Hardware Efficiency- Hardware efficiency of the BSG-augmented variations of ResNet-56 with different code vector sizes compared to the original model in terms of on-chip latency and energy consumption reported by Vivado HLS tool and XPE, energy improvement of off-chip memory accesses and the resource utilization is summarized in Table 5.6. The latency of each model is reported by Vivado HLS tool that only considers latency of on-chip modules. Based on the reported latency, for the case with $\approx 1\%$ drop compared to original model, 1.3% and with $\approx 5\%$ accuracy drop, 15.8% improvement of latency is reported by Vivado HLS tool. The estimated energy consumption of FPGA, calculated by the estimated on-chip power from XPE tool and the reported latency from Vivado HLS, is reduced from 50.11mJ to 47.30mJ-40.25mJ depending on the size of the code vector. The reason is that larger code vectors require more cycles and resources on FPGA compared to smaller ones due to increased computation complexity of their BSG process.

Table 5.6: Latency reported by Vivado HLS and the corresponding on-chip energy consumption of FGPA, energy improvement of DRAM accesses and resource utilization on a Xilinx Zynq-7000 series FPGA (xc7z020-clg484) for BSG-augmented ResNet-56 compared to the original model for different sizes of code vectors (n_c)

	Original	$n_c = 256$	$n_c = 512$	$n_c = 1024$	$n_c = 2048$
Latency on FPGA	28.8ms	24.25ms	24.54ms	27.09ms	29.2ms
Energy Cons. on FPGA	50.11mJ	40.25mJ	41.22mJ	46.05mJ	47.30mJ
Energy Saving of DRAM	1.0×	35.38×	32.48×	27.9×	21.77×
BRAM	39 (13%)	35 (12%)	35 (12%)	36 (12%)	38 (13%)
DSP	6 (2%)	6 (2%)	6 (2%)	6 (2%)	6 (2%)
FF	20345 (23%)	13697 (12%)	14763(13%)	16888 (15%)	19131 (17%)
LUT	48994 (92%)	43435 (81%)	45842 (86%)	49802 (93%)	53214 (100%)

The reduction in the energy consumption of DRAM accesses, shown in Table 5.6, due to lower number of parameters to be accessed during inference time, is $21.77\times - 35.38\times$. The results show that $21.77\times$ energy saving in off-chip memory accesses with only $\approx 1\%$ accuracy drop is achieved and $27.9\times$ with $\approx 1.5\%$ accuracy drop. This high energy saving is achieved because of lower parameter size and higher reuse distance of the on-chip parameters compared to the original model. To estimate the energy of off-chip memory accesses, accessing both parameters and feature maps according to the dataflow described in Section 5.6.2 are considered. Since the size of feature maps to be transferred between FPGA and DRAM even with the mentioned dataflow is still high compared to the model parameter, the energy saving of the cases with small parameter sizes such as the case corresponding to $V = 256$ is less than the ratio of their parameter reduction.

In addition, comparing resource utilization of the ResNet-56 and its different BSG-augmented variations on the underlying FPGA, provided in Table 5.6, shows that fewer number of BRAMs and FFs are always utilized, mainly because of lower number of parameters used on-chip.

Table 5.7: Accuracy and the model size of MobileNetV2 and its BSG augmented version for different sizes of code vectors

	Original	$n_c = 32$	$n_c = 64$	$n_c = 128$
Accuracy	93.33%	91.29%	91.67%	92.19%
Param. Size. (Train)	2.30MB	1.73MB	1.92MB	2.30MB
Param. Size. (Inference)	2.30	1.56	1.57	1.59
Comp. Ratio (Inference)	$1\times$	$1.48\times$	$1.46\times$	$1.44\times$

More number of LUTs for the cases with vector size of 1024 and 2048 are used compared to the original model, due to requiring more resources to implement their BSG.

BSG for MobileNetV2

Here, in this section, we apply our method to MobileNetV2, which is a compact model based on using low-rank filters, i.e., separable point-wise and depth-wise filters. We integrate BSG into the MobileNetV2 architecture by modifying the first convolutional layers of each block that are point-wise convolutions. The reason why we choose to modify only point-wise convolutional layers is that they account for a large fraction of parameters and operations among the convolutional layers in that model. Since MobileNetV2 is sensitive to the approximation in its parameters due to the compact form of its convolutional filters and layers, we only apply BSG on the first point-wise layer in each block to maintain the accuracy within 2%. Each slice of the convolution parameters for the BSG-augmented point-wise layers is selected as a tensor of shape $(16, 16, 1, 1)$, each of which is generated by BSG network. To generate these slices, the shape of BSG matrix (A_{BSG}) for these layers is set to $(n_c, 16, 16, 1, 1)$, which is represented by one base vector of n_c binary elements, and the efficiency for different sizes of code vectors (i.e., 16, 32, 64, 128) are evaluated.

Parameter Reduction and Accuracy- The test accuracy of these models are summarized

Table 5.8: Latency reported by Vivado HLS tool and the corresponding on-chip energy consumption on FGPA, energy improvement of DRAM accesses and resource utilization on a Xilinx Zynq-7000 series FPGA (xc7z020-clg484) for BSG-augmented MobileNetV2 compared to the original model for different sizes of code vectors (n_c)

	Original	$n_c = 32$	$n_c = 64$	$n_c = 128$
Latency on FPGA	200.4ms	189.23ms	190.2ms	191.3ms
Energy Cons. on FPGA (mJ)	358.71mJ	323.59mJ	336.65mJ	340.51mJ
Energy Saving of DRAM	N/A	44.24%	43.87%	43.13%
BRAM	131 (46%)	130 (46%)	130 (46%)	130 (46%)
DSP	35 (15%)	35 (15%)	35 (15%)	35 (15%)
FF	16209 (15%)	14399 (13%)	14645 (13%)	15295 (14%)
LUT	45219 (84%)	44154 (82%)	44234 (83%)	45103 (84%)

in Table 5.7. The accuracy of the aforementioned BSG-augmented version of MobileNetV2 with the code vector size of 128 is dropped by $\approx 1\%$ with $1.44\times$ compression ratio during inference time, while with the vector size of 32, $\approx 2\%$ accuracy drop with about only 4% higher compression ratio is observed, which indicates the sensitivity of separable filters to this approach.

Hardware Efficiency- The hardware efficiency of different variations of BSG-augmented MobileNetV2 based on different values of n_c is summarized in Table 5.8. The reported on-chip latency by Vivado HLS tool shows the latency improvement of 4%-6% in the mentioned variation of MobileNetV2. On-chip latency reduction is relatively low due to the dominance of convolution operations compared to on-chip memory accesses. The corresponding on-chip energy consumption obtained by the reported latency and XPE tool, is improved by $\approx 5\% - 10\%$ depending on the level of approximation and the accuracy drop. Comparing resource utilization of FPGA shows fewer number of BRAMs and FFs required for BSG-augmented version due to lower number of parameters to represent the original parameters of the impacted layers. In addition, fewer number of LUTs in the BSG-augmented ones shows that the required resources for implementing BSG is compensated by less resources required for storing and interfacing fewer number of parameters in the BSG-augmented one.

Based on the implementation of MobileNetV2 and its BSG-augmented version on the accelerator architecture described in Section 5.6.2, the size of parameters transferred between FPGA and main memory are dominant compared to the size of feature maps; therefore, similar energy improvement of off-chip memory accesses ($43\% - 44\%$) to the compression ratio of the model size is achieved for different code vector sizes.

5.8 Summary of the Chapter

We presented a novel and easy to implement method to reduce the number of unnecessary parameters of convolutional layers during both training and inference by representing them in a low dimensional space through the use of a simple auxiliary neural network without significantly compromising the accuracy or tangibly adding to the processing burden. In addition, we presented an accelerator architecture to implement CNN models augmented by a Binarized Slice Generator (BSG). This architecture reduces latency and memory accesses due to two reasons: compact representation of model parameters through a smaller sizes of binary code vectors and binarized CSG matrix, called BSG matrix; and simplified BSG network and longer reuse distance of parameters loaded on the on-chip buffers.

The experimental results on CIFAR-10 dataset show that on show 5%-24% on-chip energy efficiency for ResNet-56. In addition, due to lowering off-chip memory accesses by reducing the size of the parameters in inference phase by $\approx 16\times$-$77\times$, BSG improves the overall energy of DRAM accesses in ResNet-56 by $\approx 22\times$-$35\times$ with $1\% - 5\%$ drop in classification accuracy. We note that the on-chip energy and latency savings are significantly less than the parameter compression would suggest. The gains in energy and latency are over and above all optimizations that are already implemented in commercial synthesis tools. We also apply our proposed method to one third of the layers in MobileNetV2, which is an already compact network compressed by other methods, in particular, separable filters, to show the effectiveness of our method on compressed networks. The evaluation shows that $4\% - 6\%$ of on-chip energy consumption and $43\% - 44\%$ of energy required for off-chip memory accesses are saved, while the accuracy drop is $\approx 1\% - 2\%$. We expect higher latency improvement for larger sizes of images such as images from ImageNet-1K dataset, due to larger number of input partitions, thus longer reuse distance of BSG parameters.

There are still several directions that can be pursued in future. The use of this method for other tasks, especially other than vision related tasks, such as natural language processing, etc. needs to be assessed.The combination of this method with efficient computation and compression methods mentioned in this chapter for distributed machine learning and machine learning acceleration for edge devices need to be explored further. Additionally, the use of more than one CSG for different classes of filters or the use of non-linear and/or multi-layer CSGs remains to be investigated. Tuning hyper-parameters for training the CSG-augmented networks is omitted in this work, which can be considered as a possible direction to improve the results.

Chapter 5 contains the materials of Vahideh Akhlaghi, Hamed Omidvar, Massimo Franceschetti, and Rajesh K. Gupta, “Parameter Approximation of CNNs for Improved Inference on FPGA”, *submitted for publication in Design Automation Conference (DAC)*, 2021, of which this dissertation author is the primary author, and Hamed Omidvar, Vahideh Akhlaghi, Hao Su, Massimo Franceschetti, and Rajesh K. Gupta, “Associative Convolutional Layers”, *submitted for publication in International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2021, of which this dissertation author is the primary investigator.

Chapter 6

Conclusion

In this dissertation, we have provided algorithm-hardware optimizations of DNNs to improve their computation costs in order to pave the way for their deployment on edge devices, which helps to maintain users' data privacy and provide fast highly accurate responses to their requests. Through considering the optimization opportunities at both algorithm and hardware level and the limitations imposed by the hardware such as limited resources and model performance such as accuracy of the results, it has been shown that computation costs of DNNs can be significantly improved on the small sizes of hardware platforms that are suitable for edge devices without compromising the accuracy of DNN models below the acceptable threshold.

Low-cost hash-based function approximation of main costly operations in CNNs through approximate computation reuse is one of the approaches explored in this dissertation that optimizes the computation and hardware implementation of CNNs with acceptable accuracy. We also show that dynamic network pruning decreases the number of massively occurring operations at runtime by exploiting the algorithmic structure of CNNs and activating the operations of activation functions in CNNs earlier based on lightweight yet optimal threshold of computation at runtime. In addition, exploiting the error tolerance of different layers of CNNs to reduced precision at algorithm and hardware level and capability of underlying platforms to support such precision reduction, optimal platform-aware algorithm-hardware approximation to optimize the computation of CNNs based on a set of constraints related to accuracy of models, resource limitations of underlying computing platforms and dataflow of the underlying hardware architecture. Finally, reducing the size of model parameters, thus reducing the required storage to store these models and reducing the off-chip and on-chip memory accesses, has been achieved through linear approximation of parameters of CNNs with a smaller number of auxiliary (sometimes, binary) parameters.

In conclusion, this dissertation shows that simultaneous optimization of algorithms of DNN considering the hardware limitations, and optimization of hardware implementation of DNNs exploiting the opportunities at their algorithms is a promising direction to improve the computation costs of DNNs in order to improve their deployment for executing edge applications.

Bibliography

[AFL97] Mir Azam, Paul Franzon, and Wentai Liu. Low power data processing by elimination of redundant computations. In *Proceedings of the International Symposium on Low Power Electronics and Design, 1997*, pages 259–264, Monterey, CA, USA, 1997. ACM.

[AHS17] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):32, 2017.

[AJH$^+$16] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodtand Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *ISCA*, 2016.

[AKAKP17] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. Rap-cla: A reconfigurable approximate carry look-ahead adder. *IEEE Transactions on Circuits and Systems II: Express Briefs*, PP(99):1–1, 2017.

[AKX$^+$19] Karan Ahuja, Dohyun Kim, Franceska Xhakaj, Virag Varga, Anne Xie, Stanley Zhang, Jay Eric Townsend, Chris Harrison, Amy Ogan, and Yuvraj Agarwal. Edusense: Practical classroom sensing at scale. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(3):1–26, 2019.

[ALPBP17] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary Neural Networks for Resource-efficient AI Applications. In *IJCNN*, 2017.

[AMD] Amd app sdk v2.5.

[ARG16] Vahideh Akhlaghi, Abbas Rahimi, and Rajesh K Gupta. Resistive bloom filters: from approximate membership to approximate computing with bounded errors. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, 2016.

[ASKM18] Tahmid Abtahi, Colin Shea, Amey Kulkarni, and Tinoosh Mohsenin. Accelerating convolutional neural network with FFT on embedded hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(9):1737–1749, 2018.

[aut] Autograd. Available at: https://github.com/HIPS/autograd.

[AZLS19] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *International Conference on Machine Learning*, 2019.

[Bar06] Mauro Barni. *Document and Image Compression*. CRC Press, 2006.

[BC10] Woongki Baek and Trishul M. Chilimbi. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. In *PLDI*, 2010.

[BM13] Joan Bruna and Stéphane Mallat. Invariant scattering convolution networks. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1872–1886, 2013.

[BMP$^+$06] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 315–326, 2006.

[Cal] Caltech 101.

[CCY$^+$17] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. In *Advances in Neural Information Processing Systems*, pages 742–751, 2017.

[CDS90] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990.

[CDS$^+$14a] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[CDS$^+$14b] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *ASPLOS*, 2014.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[CGW+14] Karthik Chandrasekar, Sven Goossens, Christian Weis, Martijn Koedam, Benny Akesson, Norbert Wehn, and Kees Goossens. Exploiting expendable process-margins in drams for run-time performance optimization. In *Proceedings of the IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014.

[CLL+14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[CPK+17] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[CW16] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999, 2016.

[CWB+11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[CWL+18] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power energy estimation tool. 2018.

[CWZZ18] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 35(1):126–136, 2018.

[CYG+17] Kevin K. Chang, Abdullah Giray Yaglikci, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Onur Mutlu. Understanding reduced-voltage operation in modern dram devices: Experimental characterization, analysis, and mechanisms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2017.

[CZZ19] Xingyu Chen, Lei Zou, and Bo Zhao. Detecting climate change deniers on twitter using a deep neural network. In *Proceedings of the 2019 11th International Conference on Machine Learning and Computing*, 2019.

[DDP11] Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.

[DFC$^+$15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *ISCA*, 2015.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[Die13] Kenneth Diest. *Numerical Methods for Metamaterial Design*. Springer, 2013.

[DKSL03] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel bloom filters. In *Proceedings of IEEE Symposium on High Performance Interconnects*, HotI'03, pages 44–51, 2003.

[DLC$^+$15] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna V Palem, Olivier Temam, and Chengyong Wu. Leveraging the error resilience of neural networks for designing highly energy efficient accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1223–1235, 2015.

[DLW$^+$17] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–408. ACM, 2017.

[EVGW$^+$10] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[FA91] William T. Freeman and Edward H Adelson. The design and use of steerable filters. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 9:891–906, 1991.

[FC19] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

[Flo] Flopoco: Floating-point cores generator.

[FMC$^+$11] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision. In *CVPR Workshops*, 2011.

[FMJS19] Mohamed Amine Ferrag, Leandros Maglaras, Helge Janicke, and Richard Smith. Deep learning techniques for cyber security intrusion detection : A detailed analysis. In *6th International Symposium for ICS SCADA Cyber Security Research 2019 (ICS-CSR)*, 2019.

[FOP+18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, 2018.

[FSTN16] Masayoshi Fujii, Yuuki Sato, Tomoaki Tsumura, and Yasuhiko Nakashima. Exploiting bloom filters for saving power consumption of auto-memoization processor. In *Computing and Networking (CANDAR), 2016 Fourth International Symposium on*, pages 354–360. IEEE, 2016.

[Gal12] Sameh Galal. *Energy Efficient Floating-Point Unit Design*. PhD thesis, The Department of Electrical Engineering of Stanford University, 2012.

[GLL19] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7036–7045, 2019.

[GMG16] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *ArXiv*, abs/1604.03168, 2016.

[GMP+11] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. Impact: imprecise adders for low-power approximate computing. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 409–414. IEEE Press, 2011.

[GMRR13] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-power digital signal processing using approximate adders. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(1):124–137, 2013.

[GPY+17] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *ASPLOS*, 2017.

[HAM07] S Himavathi, D Anitha, and A Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 4114–4122, Red Hook, NY, USA, 2016. Curran Associates Inc.

[HCS+17] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[HDY+12] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[HLM+16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, 2016.

[HLVDMW17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[HMD16] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding. In *ICLR*, 2016.

[HMS+09] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. Technical Report MIT-CSAIL-TR-2009-042, MIT, 2009.

[HP89] Stephen Jose Hanson and Lorien Y. Pratt. Comparing biases for minimal network construction with back-propagation. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 177–185. Morgan-Kaufmann, 1989.

[HPTD15a] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[HPTD15b] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, 2015.

[HRH+19] Awni Y. Hannun, Pranav Rajpurkar, Masoumeh Haghpanahi, Geoffrey H. Tison, Codie Bourn, Mintu P. Turakhia, and Andrew Y. Ng. Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network. In *Nature Medicine*, 2019.

[HSW93] Babak Hassibi, David G. Stork, and Gregory Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pages 293–299 vol.1, 1993.

[HZC+17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[HZRS16a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[HZRS16b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[HZS17] Yihui He, Xiangyu Zhang, and Jian Sun. Channel Pruning for Accelerating Very Deep Neural Networks. *arXiv preprint arXiv:1707.06168*, 2017.

[IHM+16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and¡0.5 MB Model Size. *arXiv preprint arXiv:1602.07360*, 2016.

[IPK+17] Mohsen Imani, Daniel Peroni, Yeseong Kim, Abbas Rahimi, and Tajana Rosing. Efficient neural network acceleration on gpgpu using content addressable memory. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1026–1031. IEEE, 2017.

[IPR16] Mohsen Imani, Shruti Patil, and Tajana Rosing. Approximate computing using multiple-access single-charge associative memory. *IEEE Transactions on Emerging Topics in Computing*, 2016.

[JAH+16] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial Deep Neural Network Computing. In *MICRO*, 2016.

[JKC+18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[JLL+17] Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. A review, classification, and comparative evaluation of approximate arithmetic circuits. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(4):60, 2017.

[JSD$^+$14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[JvGLS16] Jorn-Henrik Jacobsen, Jan van Gemert, Zhongyu Lou, and Arnold WM Smeulders. Structured receptive fields in CNNs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2610–2619, 2016.

[JVZ14] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference. BMVA Press*, 2014.

[JYP$^+$17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of International Symposium on Computer Architecture*, 2017.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[KK12] Andrew B Kahng and Seokhyeong Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Proceedings of the 49th Annual Design Automation Conference*, pages 820–825. ACM, 2012.

[KKC$^+$16] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *ISCA*, 2016.

[KMY$^+$16] Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[KOY^{+}19] S. Koppula, L. Orosa, A. G. Yaglıkc, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. Eden: Enabling energy-efficient, high-performance deep neural network inference using approximate dram. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KWW^{+}17] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J Pai, and Naveen Rao. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in Neural Information Processing Systems 30*, pages 1742–1752. Curran Associates, Inc., 2017.

[LAP^{+}14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[LBG^{+}15] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*, 2015.

[LCA^{+}11] 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.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[LCL^{+}15] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. PuDianNao: A Polyvalent Machine Learning Accelerator. In *ASPLOS*, 2015.

[LG16] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[LGR^{+}15] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan V. Oseledets, and Victor S. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *CoRR*, abs/1412.6553, 2015.

[LHM+18] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[LJVM12] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture*, 2012.

[LKD+16] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[LKP+15] Donghyuk Lee, Yoongu Kim, Gennady Pekhimenko, Samira Khan ; Vivek Seshadri, Kevin Chang, and Onur Mutlu. Adaptive-latency dram: Optimizing dram timing for the common-case. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[LLH+16] Seogoo Lee, Dongwook Lee, Kyungtae Han, Emily Shriver, Lizy K. John, and Andreas Gerstlauer. Statistical quality modeling of approximate hardware. In *International Symposium on Quality Electronic Design (ISQED)*, pages 163–168, March 2016.

[LLZ+17] David Liu, Nathan Lay, Shaohua Kevin Zhou, Jan Kretschmer, Hien Nguyen, Vivek Kumar Singh, Yefeng Zheng, Bogdan Georgescu, and Dorin Comaniciu. Method and system for approxmating deep neural networks for anatomical object detection. In *Siemens Healthcare GmbH, US Patent*, 2017.

[LMC+16] Edward H. Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S. Simon Wong. Lognet: Energy-efficient neural networks using logrithmic computations. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016.

[LPMZ11] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

[LSKS17] Yingyan Lin, Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. PredictiveNet: An Energy-efficient Convolutional Neural Network via Zero Prediction. In *ISCAS*, 2017.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ASPLOS*, 1996.

[Mal12] Stéphane Mallat. Group invariant scattering. *Communications on Pure and Applied Mathematics*, 65(10):1331–1398, 2012.

[MARAM18] Roberto Fernandez Molanes, Kasun Amarasinghe, Juan Rodriguez-Andina, and Milos Manic. Deep learning and reconfigurable platforms in the internet of things: Challenges and opportunities in algorithms and hardware. In *IEEE Industrial Electronics Magazine*, 2018.

[MG12] Franck Mamalet and Christophe Garcia. Simplifying convnets for fast learning. In *International Conference on Artificial Neural Networks*, pages 58–65. Springer, 2012.

[MHP$^+$17] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. Exploring the Regularity of Sparse Structure in Convolutional Neural Networks. *CoRR*, 2017.

[mica] https://www.micron.com/media/documents/products/technical-note/dram/tn4612.pdf.

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

[MPA$^+$16] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE HPCA*, pages 14–26, March 2016.

[MRR11] Sasa Misailovic, Daniel M Roy, and Martin C Rinard. Probabilistically Accurate Program Transformations. In *SAS*, 2011.

[MSC$^+$16] Yufei Ma, Naveen Suda, Yu Cao, Jae sun Seo, and Sarma Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[MSHR10] Sasa Misailovic, Stelios Sidiroglou, Hank Hoffman, and Martin Rinard. Quality of Service Profiling. In *ICSE*, 2010.

[MSS$^+$16] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *2016 International Conference On Computer Aided Design (ICCAD)(prijato)*, page 7, 2016.

[Mul] Multi2sim: A heterogeneous system simulator.

[NBA$^+$18] Roman Novak, Yasaman Bahri, Daniel A Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: an empirical study. *arXiv preprint arXiv:1802.08760*, 2018.

[NHKL20] Duy Thanh Nguyen, Nguyen Huy Hung, Hyun Kim, and Hyuk-Jae Lee. An approximate memory architecture for energy saving in deep learning applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[OYSO17] Fumiya Okubo, Takayoshi Yamashita, Atsushi Shimada, and Hiroaki Ogata. A neural network approach for students' performance prediction. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, pages 598–599, 2017.

[PGC$^+$] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zeming Lin Zachary DeVito, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *ArXiv*.

[PRM$^+$17] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*, 2017.

[RBG13] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 60(12):847–851, Dec 2013.

[RDS$^+$15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 2015.

[res] Proper resnet implementation for cifar10/cifar100 in pytorch. In *https://github.com/akamaster/pytorch_resnet_cifar10*.

[RGLM$^+$14] Abbas Rahimi, Amirali Ghofrani, Miguel Angel Lastras-Montano, Kwang-Ting Cheng, Luca Benini, and Rajesh K Gupta. Energy-efficient gpgpu architectures via collaborative compilation and memristive memory-based computing. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[Rin06] Martin Rinard. Probabilistic Accuracy Bounds for Fault-tolerant Computations that Discard Tasks. In *ICS*, 2006.

[Rin07] Martin C. Rinard. Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points. In *OOPSLA*, 2007.

[RORF16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[RSLF13] Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. Learning separable filters. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2754–2761, 2013.

[RVPR13] Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Adithya Parandhaman, and Anand Raghunathan. Relax-and-retime: A methodology for energy-efficient recovery based design. In *Design Automation Conference (DAC)*, pages 1–6. IEEE, 2013.

[RWA^{+}16] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 267–278. IEEE Press, 2016.

[SCYE17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint*, 2017.

[SDMHR11] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *FSE*, 2011.

[SHZ^{+}18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[SJLS14] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng. Lee, and Mahlke Scott. Paraprox: pattern-based approximation for data parallel applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, ASPLOS 14, pages 35–50, 2014.

[SLJ^{+}15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *CVPR*, 2015.

[SM13] Laurent Sifre and Stéphane Mallat. Rotation, scaling and deformation invariant scattering for texture discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1233–1240, 2013.

[SM19] Sebastian Scher and Gabriele Messori. Weather and climate forecasting with neural networks: using general circulation models (gcms) with different complexity as a study ground. In *Geoscientific Model Development*, 2019.

[SPM^{+}16] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From High-Level Deep Neural Models to FPGAs. In *MICRO*, 2016.

[SSAL16] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, pages 2217–2225, 2016.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[SZW^{+}18] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint*, 2018.

[tin] tiny-dnn: https://github.com/tiny-dnn/tiny-dnn.

[Tor30] TorchVision. tourchvision.models – PyTorch Master Documentation. https://pytorch.org/docs/stable/torchvision/models.html, Accessed: 2020-01-30.

[TXZ^{+}15] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, and Weinan E. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

[Vel10] Todd Veldhuizen. Measures of image quality. *CVonline: The Evolving, Distributed, Non-Proprietary, On-Line Compendium of Computer Vision*, 2010.

[VKAKP17] Shaghayegh Vahdata, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. Letam: A low energy truncation-based approximate multiplier. *Computers Electrical Engineering*, 63(Supplement C):1 – 17, 2017.

[VRRR14] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Axnn: energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 27–32. ACM, 2014.

[Wat94] Andrew B Watson. Image compression using the discrete cosine transform. *Mathematica journal*, 4(1):81, 1994.

[WBC^{+}19] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiaov, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[Wel99] Stephen T Welstead. *Fractal and wavelet image compression techniques*. SPIE Optical Engineering Press Bellingham, Washington, 1999.

[WH19] Yueh-Chi Wu and Chih-Tsun Huang. Efficient dynamic fixed-point quantization of cnn inference accelerators for edge devices. In *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2019.

[Win80] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

[WLF17] Min Wang, Baoyuan Liu, and Hassan Foroosh. Factorized convolutional neural networks. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 545–553, 2017.

[WLL$^+$19] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. pages 8604–8612, 06 2019.

[WSL$^+$18] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In *Advances in Neural Information Processing Systems*, pages 9850–9861, 2018.

[WWLZ18] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1299–1309. Curran Associates, Inc., 2018.

[WWW$^+$16] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2082–2090, Red Hook, NY, USA, 2016. Curran Associates Inc.

[XGD$^+$17] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995, 2017.

[XYP$^+$17] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[YA18] Min Ye and Emmanuel Abbe. Communication-computation efficient gradient coding. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of*

Machine Learning Research, pages 5610–5619, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[YBU19] Chika Yinka-Banjo and Ogban-Asuquo Ugot. A review of generative adversarial networks and its application in cybersecurity. In *Artificial Intelligence Review*, 2019.

[YJZ$^+$06] Hongmei Yan, Yingtao Jiang, Jun Zheng, Chenglin Peng, and Qinghui Li. A multilayer perceptron-based medical decision support system for heart disease diagnosis. *Expert Systems with Applications*, 30(2):272–281, 2006.

[YPT$^+$15] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. RFVP: Rollback-Free Value Prediction with Safe to Approximate Loads. In *TACO*, 2015.

[YYX$^+$19] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[ZDZ$^+$16] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An Accelerator for Sparse Neural Networks. In *MICRO*, 2016.

[ZJ13] Mahmoud Zangeneh and Ajay Joshi. Design and optimization of nonvolatile multibit 1t1r resistive ram. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(8):1815–1828, Sept 2013.

[ZLS$^+$15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*, 2015.

[ZPL14] Hang Zhang, Mateja Putic, and John Lach. Low power gpgpu computation with imprecise hardware. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 1–6, 2014.

[ZWT$^+$15] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: an approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 701–706, 2015.

[ZZLS18] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.