Low Power Approximate Hardware Design for Multimedia and Neural Network Applications

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Farhana Sharmin Snigdha

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Sachin S. Sapatnekar, Advisor

June, 2019

Acknowledgements

The writing of this note of thanks is the finishing touch of the long five-year journey of my Ph.D. graduate life. It has been a period of intense learning for me, not only in the scientific arena but also on a personal level. The journey of being a Ph.D. has lots of ups and downs, struggles that helped me shape into a humble person. I would like to reflect on the people who have supported and helped me throughout this process.

First of all, I want to thank my advisor, Prof. Sachin S. Sapatnekar, for his support throughout my graduate studies. It has been an honor working with a great researcher like him. I would not be able to complete this thesis without his guidance as well as his encouragement all along these years. He has been an incredible mentor, helping me organize my thoughts, guiding me to push through the challenges, and teaching me to pay attention to details. He has helped me to visualize the big picture of any work and improve my technical writing, which I lacked completely at the beginning of my graduate studies. He provided me guidance at every step of my graduation process with patience and support. The time I have worked with him has been the most productive five years of my life, and I am grateful for that.

I want to thank my committee members at the University of Minnesota: Prof. Kia Bazargan, Prof. Ulya Karpuzcu, and Prof. Pen-Chung Yew, for their constructive feedback about my research. I would like to thank Prof. Jiang Hu from Texas A&M University and Prof. Chris Kim for the valuable support, guidance and collaboration, which helped me enrich the horizon of my knowledge. I am grateful to the National Science Foundation (NSF), and ECE Department at the University of Minnesota, for the financial support towards my thesis projects. I want to express my gratitude to the ECE staff, especially Carlos Soria and Chimai Nguyen, for their help and support.

I am grateful to my teachers at BUET for encouraging me to pursue a Ph.D. I am

also thankful to my BUET friends for making life at BUET a memorable experience and supporting me through various ups and downs in my life.

I want to thank Dr. Deepashree Sengupta, Dr. Vivek Mishra, Dr. Zhaoxin Liang and Meghna Mankalale for their help during the starting years of my graduate school with respect to navigating through the various CAD tools needed for my projects, insightful inputs about my research, and fun-filled de-stressing discussions. I want to thank Susmita and Tonmoy, my undergraduate friends and now labmates, for all the help, thoughtful discussions and support. I would also like to thank Tengtao, Qianqian, Masoud, Vidya, Kunal, and Shohel for being wonderful labmates.

I would not be at this stage of my life without the support of my family. I want to thank my parents, Fatema Banu and Khurshid Ahmed for their belief in my capabilities and encouraging me to keep pushing through the boundaries with determinations. I also want to thank my ever-supporting and enthusiastic husband, and best friend Ibrahim Ahmed, without whose constant support, encouragement and help, I could not have finished my Ph.D. He deserves almost as much credit as my advisor in terms of providing constructive criticisms, by helping me deal with failures and rejections and giving a shoulder to cry. He has also been a tremendous support throughout the whole Ph.D. journey to keeping me sane. I would like to thank my mother-in-law and father-in-law for all the encouragement and prayers. I want to convey my thanks to my elder brothers Rumman Ahmed and Dr. Istiaque Ahmed, and my sister-in-laws Nahia Islam and Dr. Nadira Sharmin, for their good wishes. My young nephew, Ryan, has been a constant source of my joy and refreshment.

Last but not least, I want to thank my friends in Minneapolis, and I feel lucky to have a group to hang out with, outside work. Staying away from home and loved ones are always tough. The Bangladeshi community and the Bangladesh Student Association (BDSA) had been a constant source of support and warmth. Thanks to Dr. Debaroti Ghosh, Dr. Anindya Saha, Dr. Aminul Mehedi, Taskin Hoque, Khandakar Wahedur Rahman, Dr. Tanjila Nawshin and Zamshed Iqbal for all the get-togethers, long talks, laughter, amazing foods and helps to make the graduate school a joyful experience.

Lastly, I am thankful for belonging to such a beautiful campus of the University of Minnesota, which never failed to amaze me with her scenic beauty in all seasons.

Dedication

To my wonderful parents.

Abstract

In today's data- and computation-driven society, day-to-day life depends on devices such as smartphones, laptops, smart watches, and biosensors/image sensors connected to computational engines. The computationally intensive applications that run on these devices incur high levels of chip power dissipation, and must operate under stringent power constraints due to thermal or battery life limitations. On future hardware platforms, a large fraction of computation power will be spent on error-tolerant multimedia applications such as signal processing tasks (on audio, video, or images) and artificial intelligence (AI) algorithms for recognizing voice and image data. For such error-tolerant applications, approximate computation has emerged as a new paradigm that provides a pragmatic approach for trading off energy/power for computational accuracy. A powerful method for implementing approximate computing is by performing logic-level or architecture-level hardware modifications. The effectiveness of an approximate system depends on identifying potential modes of approximation, accurate modeling of injected error as a function of the approximation, and optimization of the system to maximize energy savings for user-defined quality constraints. However, current approaches to approximate computation involve *ad hoc* trial-and-error based methods that do not consider the effect of approximations on system-level quality metrics. Additionally, prior methods for approximate computation have provided little or no scope for modulating the design based on user- and application-specific error budgets.

This thesis proposes adaptive frameworks for energy-efficient approximate computing, leveraging the target application characteristics, system architecture, and input information to build fast, power-efficient approximate circuits under a user-defined error budget. The work is focused on two well-established, widely-used, and computationally intensive applications: multimedia and artificial intelligence. For multimedia systems, where minor errors in audio, image, and video are imperceptible to the human senses, approximate computations can be very effective in saving energy without significant loss in the quality of results. AI applications are also good candidates for approximation as they have inherent error-resilience feedback mechanisms embedded into their computations. This thesis demonstrates methodologies for approximate computing on

representative platforms from the multimedia and AI domains, namely, the widely used JPEG architecture, and various architectures for deep learning.

The first part of the thesis develops a methodology for designing approximate hardware for JPEG that is input-independent, i.e., it aims to meet the specified error budgets for any inputs. The error sensitivities of various arithmetic units within the JPEG architecture with respect to the quality of the output image are first modeled, and a novel optimization problem is then formulated, using the error sensitivity model, to maximize power savings under an error budget. The optimized solution provides 1.5×–2.0× power savings over the original accurate design, with negligible image quality degradation. However, the degree of approximation in this approach must necessarily be chosen conservatively to stay within the error budget over all possible input images.

The second part of the thesis designs an image-dependent approximate computation process that uses image-specific input statistics to dynamically increase the approximation level over the image-independent approach, thereby reducing its conservatism. This approach must overcome several challenges: circuitry for real-time extraction of input image statistics must be inexpensive in terms of both power and computation time, and schemes for translating abstracted image information into dynamically chosen approximation levels in hardware must be devised. The approach devises a simplified heuristic to estimate the input data distribution. Based on this distribution, a dynamic approximate architecture is developed, altering the approximation levels for input images in real-time. Over a set of benchmarks, the input-dependent approximation provides an average of 31% additional power improvement, as compared to the input-independent approximation process.

The final part of the thesis addresses the use of approximate computing for convolutional neural networks (CNNs), which have achieved unprecedented accuracy on many modern AI applications. The inherent error-resilience and large computation requirements imply that CNN hardware implementations are excellent candidates for approximate computation. A systematic framework is developed to dynamically reduce the computation in the CNN based on its inputs. The approach is motivated by the observation that for a specific input class, during both the training and testing phases, some features tend to be activated together while others are unlikely to be activated. A dynamic selective feature activation framework, SeFAct, is proposed for energy-efficient

CNN hardware accelerators to early predict an input class and only perform necessary computations. For various state-of-the-art neural networks, the results show that energy savings of $20\% - 25\%$ are achievable, after accounting for all implementation overheads, with small loss in accuracy. Moreover, a trade-off between accuracy and energy savings may be characterized using the proposed approach.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern portable electronic appliances such as cellphones, laptops, and tablets have become ubiquitous and an integral part of our daily lives. These devices have progressively become capable of performing versatile and increasingly complex functionalities, such as processing media (audio, video, images), recognizing voice/image data using artificial intelligence (AI) algorithms, and data mining [1]. The diverse functionalities are enabling innovative applications in the fields of mobile healthcare, communications, navigation, and personal entertainment, and are transforming society. However, many of these contemporary applications are computationally intensive and power-hungry.

The demand for low-power solutions for integrated systems has driven the progress of process technology for many decades, in accordance with the well-known Moore's law [2]. Moore's law predicted the doubling of transistors in the integrated circuits every two years, which enables increasing the energy efficiency of computation. However, as Moore's law has slowed down, researchers have been pursuing various alternative software and hardware solutions for better energy efficiency.

The traditional model of computing has been based on scientific computing applications (e.g., computing space rocket trajectories), where achieving the highest possible accuracy is of paramount importance. However, many emerging applications show great levels of error-tolerance. For multimedia systems, minor errors in audio, image, and video are imperceptible to human senses, e.g., when a pixel color in an image is encoded as a number from 0 to 255, an error of a few units in the value is often not noticeable by the end-user. Similarly, AI algorithms that recognize a feature (e.g., a pet in an image,

or a pedestrian in a video) are intrinsically insensitive to minor "noise," including computing errors. The traditional computing models used in chips that implement these applications can be replaced with implementations based on a new paradigm, approximate computing, based on approximate designs that introduce controlled levels of errors to improve chip power, and thus enhance battery life in mobile embedded systems, or mitigate thermal constraints in desktop or datacenter applications.

1.1 The Approximate Computing Paradigm

Approximate computing [3] is a new design paradigm that leverages the inherent error-tolerance of human senses, and can potentially achieve large efficiencies in the design by deliberately introducing errors in computation by either simplifying a circuit logic or architecture or by modifying its operating conditions (supply voltage and frequency). Hence, such intentional unreliability in computation can reduce hardware costs, in terms of energy, power and area.

The vital ingredients of any methodology based on approximate design are quantifying the distribution of error injected into a system by an approximation scheme, and controlling the introduced error into the system based on user-specific preferences [4]. However, to date, the prevailing practice is to introduce approximation by an *ad hoc* trial-and-error method that does not guarantee that the system-level error lies within acceptable bounds (e.g., through numerical guarantees on the image quality in a multimedia application).

The objective of this thesis is to aid both circuit design engineers and end users to apply adaptive techniques, based on application and input characteristics, to ensure high-quality chip operation that optimizes the trade-off between error and chip power with error characterization and analytical optimization of approximate circuits. The thesis focuses on JPEG and neural networks as representative applications of multimedia and artificial intelligence, respectively.

1.1.1 Hardware for JPEG Compression

JPEG [5], a widely used image compression algorithm, is a key building block in many multimedia applications: for example, this format is used almost universally in digital

cameras. A JPEG design can be considered to illustrates a representative error-resilient computational block on which approximate computing schemes are developed; similar concepts can easily be applied to other error-tolerant computations such as video and voice processing. A core part for JPEG compression is the discrete cosine transform (DCT) architecture, which requires a significant amount of computation and power. For a moderate-sized image with a pixel dimension of 512×512 (256KB), the DCT requires about one million multiplications and two million additions. The large number of computation and the inherent error-tolerance of JPEG worked as a motivation to explore approximate computing. A few prior works have explored approximations in JPEG compression through ad hoc optimizations in general architectures, such as uniform approximation [6] and dynamic approximation with limited choices [7], without exploiting the structure of JPEG.

The first part of this thesis designs an optimized image-independent JPEG architecture. An error model is developed considering both JPEG architecture and structure of its constituent approximate arithmetic units. The JPEG hardware is then optimized under user-specified error budgets using the error model. While the image-independent JPEG architecture provides good energy savings, this approach is conservative by design to ensure that the approximation level does not violate the error budget for any input image.

The second part of the thesis focuses on designing an image-dependent approximation framework by characterizing image statistics and dynamically adjusting the approximation level to maximize energy savings. Even though the real-time adjustment of approximation level requires peripheral decision circuitry that accrues an energy overhead, it will be shown that image-dependent approximation can achieve substantial energy savings that easily pays for this overhead. The approximate JPEG architecture enhances circuit performance by increasing speed as well as extending battery life, while ensuring reliable circuit operation [8–10].

1.1.2 Neural Network Hardware

Neural networks (NNs) have achieved unprecedented accuracy on many modern AI applications such as computer vision, speech recognition, robotics, etc [11]. The architecture of a neural network, inspired by the human brain, consists of several layers

of interconnected neurons [12]. The neurons perform a weighted sum of inputs from previous layers and deliver activated output signals. The NN undergoes two highly computation-intensive but error-tolerant operation phases: training and testing. During the training phase, a large input data set from a few predefined classes is used to train the network, and a feedback path is provided to alter the weights in the network based on the classification errors. A different set of data is used to validate the network classification accuracy in the testing phase. Testing is also error-resilient, as a small error in the neuron can still produce an activated output signal. Hence, approximate computation techniques have significant prospects for designing efficient NN hardware architectures to enable the wide deployment of AI systems in everyday life.

Deep neural network (DNN) architectures [13, 14] have multiple layers consisting of artificial neurons, where each layer is trained to recognize various features of the input, with deeper layers uncovering more complex features. In recent years, convolutional neural networks (CNNs) have emerged as a very prominent branch of DNNs, and have been shown to solve increasingly complex problems with remarkably high accuracy.

The third part of this thesis focuses on designing low-power hardware accelerators for CNNs. It has been observed that during both training and testing phases of a neural network, specific features tend to be activated by the recognition of specific classes. Some paths through the network are unlikely to be activated because particular sets of neurons are seldom activated together. We propose a dynamic selective feature activation approach, SeFAct, for energy-efficient CNN hardware accelerators [15]. The proposed framework is an adaptive data-dependent scheme that selectively activates a subset of all computations, by narrowing down the possible activated classes. The reduction of class prediction helps to reduce computation and memory access and thus save significant energy.

1.2 Thesis organization

The thesis is organized as follows:

- Chapter 2 provides a brief introduction to the approximate computing paradigm and a basic architectural description of JPEG and neural network hardware. These

are the two applications on which approximate computation is demonstrated in this thesis.

- Chapter 3 presents an image-independent static approximation scheme for JPEG hardware. The approach considers the sensitivity of the error at the output for the approximations at the arithmetic units within the JPEG architecture. A nonlinear optimization problem is presented that maximizes the power savings under user-specified error constraints.

- Chapter 4 expands the approximation scope in JPEG through exploiting input image patterns. An image-dependent dynamic approximation framework is described in this chapter that optimizes approximate hardware with variable approximate bit-widths for a user-specified error budget. The dynamic approximation is implemented over the static approximation of JPEG. The proposed method can dynamically adjust the extent of approximation in the system depending on the pixel values of the input image, thus leveraging the inherent sparsity of specific images. The input distributions of different images are estimated in real time to determine the level of approximations.

- Chapter 5 identifies the scope of approximation in CNN and presents a dynamic energy reduction framework, SeFAct, for neural network hardware accelerators. The framework of SeFAct identifies the pattern of activated neurons for different classes, and groups neurons with similar activations across multiple classes into clusters. These clusters are then used to approximate a large section of the neural network that is unlikely to be activated for a given input image. The chapter presents the algorithm and analytical equations that are used to systematically and automatically explore the design space for finding optimal solutions.

- Chapter 6 concludes the thesis.

Chapter 2

Background

Many computation-intensive and power-hungry applications, such as media processing (audio, video, image), or recognition of voice/image data using artificial intelligence (AI) algorithms, are error-resilient in nature. The demand for energy efficient hardware and algorithm has driven researchers to find new ways [3,16] to optimize computation to leverage the inherent error-resilience of these applications. Approximate computation is one such paradigm where simplified, lower power hardware deliberately introduces a controlled amount of error in the system using logic or architecture modifications [6,17–21], voltage overscaling [22], time starvation [23], etc.

Approximate computation is beneficial for real-time image processing in portable devices as the results of many image processing applications interact directly with the human eye, which cannot discern discrepancies in an image beyond a certain level. Hence, these computations may be performed approximately to save memory and energy, without affecting the user experience. A typical target for approximation is in the domain of digital signal processing (DSP), which is a key functionality in multimedia processors [24]. Typical DSP operations used in image and video processing units include image compression, filtering, and reconstruction.

AI applications based on neural networks are another potential avenue for approximate computation. In recent years, the trend in CNN hardware has been to use a large number of layers in neural networks. While this increases the classification accuracy, it also results in an increase in the power and memory footprint requirements [14]. Energy reduction of neural networks is of paramount importance in both datacenters and

mobile platforms. The notion of exploiting error resilience in neural networks to reduce energy has been proposed in several works in the recent past, and various methods for reducing computations have been proposed, including data parallelization using multiple cores [25], specialized hardware [26–28], and approximate computation [29–33] based on simplified architecture and arithmetic.

In this thesis, we focus on approximate computing with logic and hardware simplification and analysis of error behavior of the system architecture to introduce controlled error for energy savings. We implement our concept on a standard JPEG architecture and several neural network topologies, covering two widely used classes of error-resilient applications: multimedia and artificial intelligence, respectively. In this chapter, we first discuss the background of approximate computation based on logic simplification is discussed. Next, an overview of various blocks of JPEG architecture is provided. Finally, we discuss the fundamentals structures of various neural network architectures.

2.1 Approximate Computing

The data (images, sound, video) used in multimedia and AI applications are represented by arithmetic numbers in hardware and are processed through a sequence of computer arithmetic operations such as addition and multiplications. For such computations, adders and multipliers are implemented as arrays of full adders (FAs). A basic building block is the conventional mirror adder, a widely used full adder (FA) that requires 24 transistors for exact computation. Switching such an adder involves charging/discharging a considerable amount of capacitance, and incurs significant propagation delay as well as high dynamic power.

Approximate implementations of an FA can be achieved by replacing some of the exact FAs in the arithmetic operations by approximate versions with a reduced number of transistors per FA. For an n-bit operation, a certain number of the least significant bits (LSBs) can be approximated with a controlled loss in accuracy. The approximation incurs an error, but the reduced transistor count implies that power and delay are also reduced. Five transistor-level approximate FA designs, named *appx1–appx5*, as proposed in [6], are described by the truth tables in in Table 2.1. Their transistor counts are listed in Table 2.2.

Table 2.1: Truth table for an exact FA and several approximate FAs.

Inputs			Outputs											
			Exact		*appx1*		*appx2*		*appx3*		*appx4*		*appx5*	
A	B	C_{in}	Sum	C_{out}	Sum_1	$Cout_1$	Sum_2	$Cout_2$	Sum_3	$Cout_3$	Sum_4	$Cout_4$	Sum_5	$Cout_5$
0	0	0	0	0	0 ✓	0 ✓	1 ×	0 ✓	1 ×	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓
0	0	1	1	0	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	0 ×	0 ✓
0	1	0	1	0	0 ×	1 ×	1 ✓	0 ✓	0 ×	1 ×	0 ×	0 ✓	1 ✓	0 ✓
0	1	1	0	1	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓	1 ×	0 ×	1 ×	0 ×
1	0	0	1	0	0 ×	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	0 ×	1 ×	0 ×	1 ×
1	0	1	0	1	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓
1	1	0	0	1	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓	0 ✓	1 ✓	1 ×	1 ✓
1	1	1	1	1	1 ✓	1 ✓	0 ×	1 ✓	0 ×	1 ✓	1 ✓	1 ✓	1 ✓	1 ✓

Table 2.2: Transistor counts for the exact FA and various approximate FAs.

	Exact FA	Approximate FA				
		appx1	*appx2*	*appx3*	*appx4*	*appx5*
Transistor count	24	16	14	11	11	4

Of these, the *appx5* adder has the simplest circuit design, and is the easiest to implement. From Tables 2.2 and 2.1, it can be observed that *appx5* has the lowest number of transistors, but has the largest number of incorrect entries (four out of eight) in the truth table. However, in a larger system, the probability of an erroneous result for a single approximate FA also depends on the way it is interconnected to other elements. In this case, the outputs of *appx5* only depends on the inputs A and B [6], and are independent of the input carry. As a result, the errors of *appx5* are independent of errors in the FAs in other bit positions, and at the system level, *appx5* shows a greater benefit over other approximate adders by containing the error.

Let us consider a large system where N_a and N_m are the bitwidths of adder, a, and the multiplier, m, respectively. We implement addition using N_a-bit ripple carry adders and multiplication using a $4:2$ compressed carry save array (CSA) based N_m-bit lower triangle multiplier. For various numbers of approximated bits, the power requirements and corresponding error variance for the five FA designs, as compared to the accurate multiplier, are shown in Fig. 2.1.

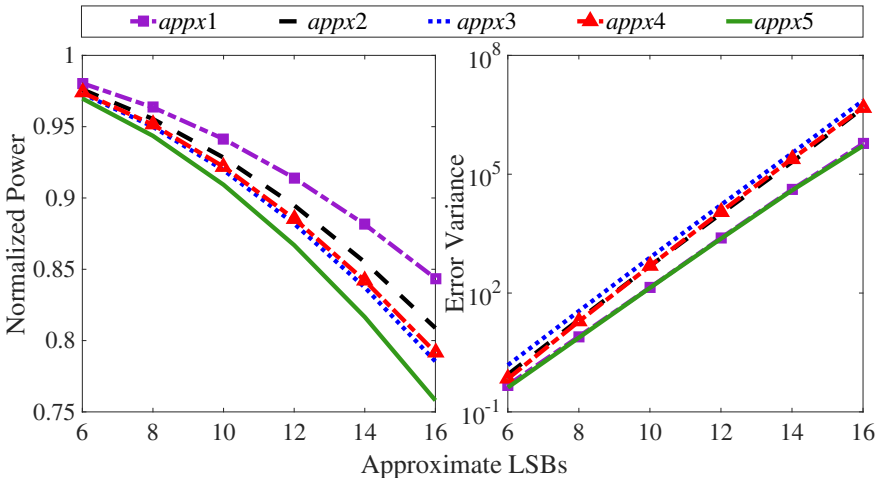


Figure 2.1: Power and error variance comparison for various approximate multipliers.

The area and delay improvements show the same trends as power (not shown). For all levels of approximations, it can be seen that the *appx5* design provides the lowest error levels and power relative to the other options.

2.2 JPEG Decomposition

JPEG is a commonly-used lossy compression method for digital images [5, 34]. This compression scheme is based on a 2-dimensional (2D) frequency-domain DCT [35], followed by a quantization step, as illustrated in Figure 2.2. The method operates by dividing an image, I_{image}, into 8×8 pixel blocks and compressing each block separately. Each such block constitutes the 8×8 initial matrix, I, which will be referred to as one *frame*. For example, a typical image of size 512×512 pixel consists of 4096 frames. Each element of the matrix, I, has a value in the range of 0 to 255, but the DCT operation is performed on a shifted matrix, M, obtained from I by subtracting 128 from each entry in I. The result of compressing M is an 8×8 matrix, C. The JPEG compression sequence has two major steps, described in the rest of this section.

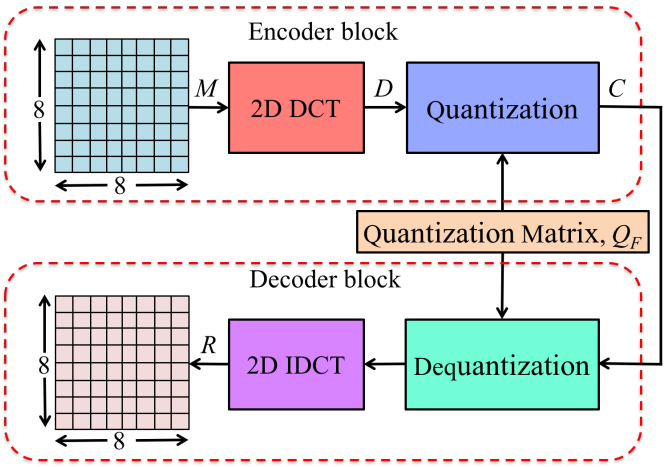


Figure 2.2: JPEG encoder block diagram.

2.2.1 The DCT Stage

Functionality

Each frame, represented by M, undergoes an 8-point 2D DCT. The DCT coefficient matrix, D, is given by

$$D(i,j) = \frac{1}{4}U(i)U(j)\sum_{x=0}^{7}\sum_{y=0}^{7}M(x,y)\times\cos\left(\frac{(2x+1)i\pi}{16}\right) \times\cos\left(\frac{(2y+1)j\pi}{16}\right),\ 0\le i,j\le 7 \tag{2.1}$$

where i [j] represent the horizontal [vertical] spatial frequencies. The scaling factor, $U(k)$, is given by

$$U(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{, if } k=0 \\ 1 & \text{, if } k>0 \end{cases} \tag{2.2}$$

Hardware Implementation

The separable property of the DCT allows the 2D operation to be decomposed in two sequential 1D DCT operations along the row and column of M, as shown in Figure 2.3. We denote the first and second sets of 1D DCT blocks as Layer 1 and Layer 2, respectively. Figure 2.4 illustrates the internal structure of Layer 1 and Layer 2 of 2D DCT.

Each layer contains eight individual 1D DCT blocks, so that over the two layers, 16 such 1D DCT blocks are required. For each block, i, in the first layer, its input comes from the i^{th} column of M, denoted by M_i, $i \in \{0, \cdots, 7\}$. The outputs of the first layer are fed as inputs to the second layer after a transpose operation.

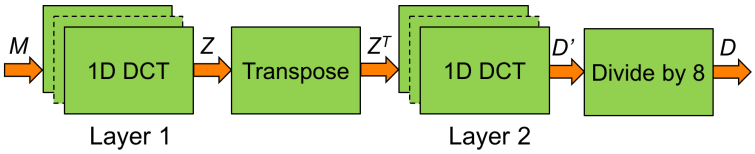


Figure 2.3: Decomposition of 2D DCT architecture into 1D DCT structures.

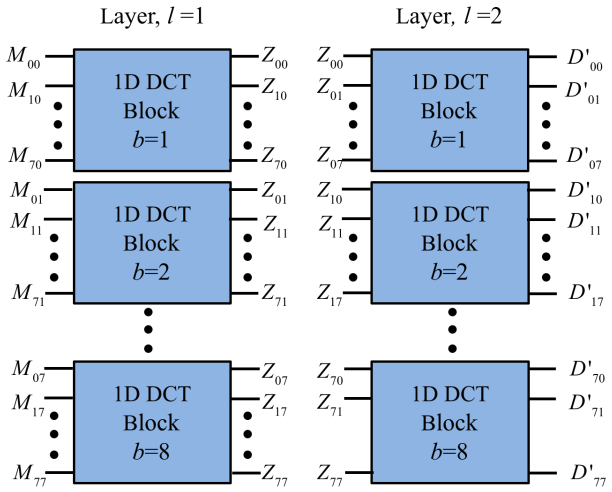


Figure 2.4: The 2D DCT using 16 1D DCT blocks.

The periodic structure of cosine terms in (2.1) can be exploited to efficiently implement fast 1D DCT operations by combining several additions and multiplications. A number of fast 1D DCT algorithms have been proposed [36–38]. Of these, the Loeffler 1D DCT algorithm [38], which requires 11 multiplications and 29 additions, achieves the theoretical lower bound for multiplications in 8-point DCT, and we have used it as a basis for our work. The output Z_i of each 1D DCT block using Loeffler's algorithm is:

$$Z_i = \sqrt{2} \sum_{x=0}^{7} U(i) M_i \cos \frac{(2x+1) i\pi}{16} \tag{2.3}$$

where M_i is as defined above. An extra $\sqrt{2}$ term is incorporated to obtain Z_0 and Z_4 without any multiplication. When this implementation is used for the 1D DCT in each layer in Figure 2.3, a final divide-by-8 stage is required, as shown in the figure, in order to translate the output D' of Layer 2 to an output D consistent with (2.1). Algorithm 1 shows the process of computing JPEG compressed matrix for a single frame using the Loeffler fast DCT technique, and compression corresponds to steps 1 through 8.

Algorithm 1 Finding the JPEG Compression Matrix, C

INPUT: 8×8 Image pixel matrix, M
OUTPUT: 8×8 JPEG compressed matrix, C.
METHOD:

1: **for** $i \leftarrow 1$ to 8 **do** ▷ Row operation
2: $\quad Z(i,:) = f_{1D,DCT}(M_i)$
3: **end for**
4: **transpose**
5: **for** $i \leftarrow 1$ to 8 **do** ▷ Column operation
6: $\quad D(i,:) = f_{1D,DCT}(Z_i)$
7: **end for**
8: $D(i,j) >> 3$ ▷ Divide by 8
9: **return** $C(i,j) = \text{round}\left[\frac{D(i,j)}{Q_F(i,j)}\right]$ ▷ Quantization step

The Loeffler 1D DCT can be represented as the directed acyclic graph (DAG) structure shown in Fig. 2.5. Each node in the DAG represents arithmetic operations such as add/subtract or multiply. There are total of eight stages, $s(i), i = 1, \cdots, 8$, between the input and output. There are 11 multiplications and 29 additions in each 1D DCT block. The net computation over the 16 1D DCT blocks in the 2D DCT block requires 176 multiplications and 464 additions per frame.

It is important to note that all multipliers perform constant signed multiplications, where the fractional constant inputs are stored as fixed-bit integers by multiplying them by 2^7; this scaling is reversed at the output through shift operations.

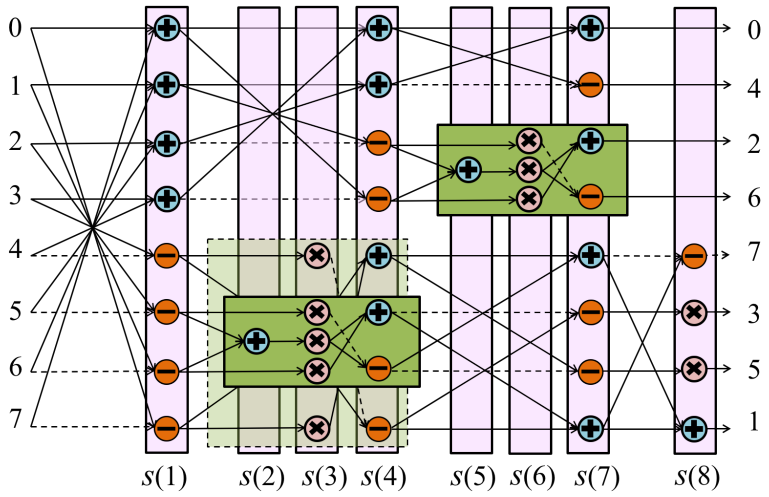


Figure 2.5: A DAG model of the Loeffler DCT.

2.2.2 The Quantization Stage

As illustrated in Algorithm 1, the DCT coefficient matrix, D, undergoes a lossy compression step in the final step in line 9 as it performs the operation,

$$C(i,j) = \text{round}\left[\frac{D(i,j)}{Q_F(i,j)}\right] \tag{2.4}$$

i.e., each element in the DCT coefficient matrix, $D(i,j)$, is individually divided by the corresponding element of a quantization matrix, $Q_F(i,j)$, and rounded to the nearest integer to form the quantized result, $C(i,j)$, of JPEG compression.

The subscript F of the quantization matrix, Q_F, is a quality factor ranging from 1 to 100, where 1 corresponds to the lowest quality. Commonly used values of F are 50

and 90 and the corresponding quantization matrices are:

$$Q_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} Q_{90} = \begin{bmatrix} 3 & 2 & 2 & 3 & 5 & 8 & 10 & 12 \\ 2 & 2 & 3 & 4 & 5 & 12 & 12 & 11 \\ 3 & 3 & 3 & 5 & 8 & 11 & 14 & 11 \\ 3 & 3 & 4 & 6 & 10 & 17 & 16 & 12 \\ 4 & 4 & 7 & 11 & 14 & 22 & 21 & 15 \\ 5 & 7 & 11 & 13 & 16 & 21 & 23 & 18 \\ 10 & 13 & 16 & 17 & 21 & 24 & 24 & 20 \\ 14 & 18 & 19 & 20 & 22 & 20 & 21 & 20 \end{bmatrix}$$

The rounding of the results is a key step where some information of the image is discarded. At a lower quality factor, F, the divisors are higher, implying that higher compression ratio is achieved [5]. Further, the key DCT coefficients are close to $D(0,0)$, and the elements of Q_F progressively increase as we move away from the $(0,0)$ element.

2.2.3 Image Reconstruction

The reconstruction of the compressed image shown in Fig. 2.2 follows a procedure analogous to compression, except that the inverse DCT (IDCT) technique is used instead of the DCT to reconstruct the image matrix, R. The IDCT can also be implemented using a butterfly, and incurs similar computation as the compression step.

The difference between the initial image, I, and the reconstructed image, R, determines the compression quality, measured in terms of its peak signal to noise ratio (PSNR). The PSNR of the JPEG compressed image largely depends on the quantization matrix, Q_F. If I_x and R_x are the x^{th} frame of the original and reconstructed image matrices, respectively, then in an image with N frames, the $PSNR$ is:

$$PSNR = 10 \log \left(\frac{\max \left(I_{image}^2 \right)}{MSE} \right) \tag{2.5}$$

$$\text{where,} \quad MSE = \frac{1}{N} \sum_{x=1}^{N} (I_x - R_x)^2$$

2.3 Convolution Neural Network (CNN) Architecture

State-of-the-art CNNs employ a deep hierarchy of layers to achieve very high accuracy. Each layer recognizes unique features of the input, and the deeper layers uncover more complex features that enable the solution of increasingly challenging problems. This high depth implies that deep neural networks require very large computational energy. The third part of the thesis focuses on designing energy-efficient CNN hardware.

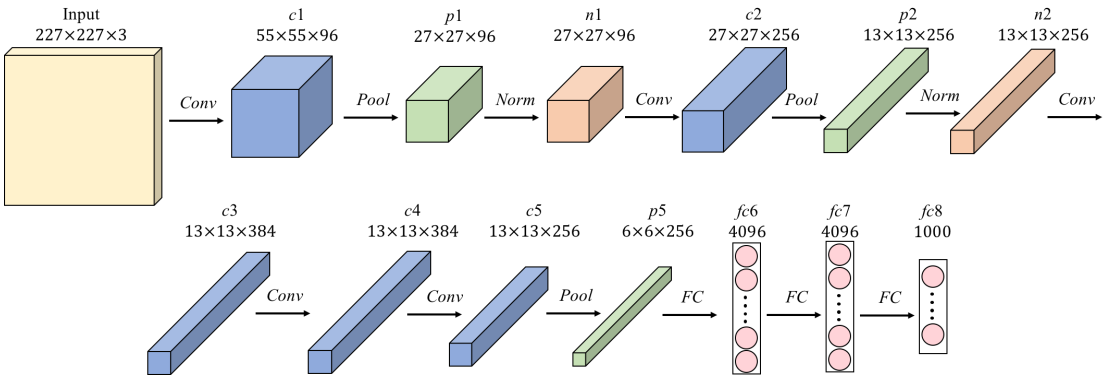


Figure 2.6: The architecture of AlexNet.

2.3.1 Various Layers of a CNN

A CNN consists of various combinations of convolutional (*Conv*), fully connected (*FC*), pooling (*Pool*) as well as normalization (*Norm*) layers. Fig. 2.6 shows the architecture of the AlexNet [39] NN architecture that is built from a combination of these layer types. The computation in the CNN is primarily dominated by *Conv* and *FC* layers.

Conv Layer

Each of the *Conv* layers in CNNs is primarily composed of high-dimensional convolutions. As illustrated in Fig. 2.7, there are three types of data associated with a *Conv* layer:

- **ifmap**, the input feature map, $F^{if} \in \mathbb{R}^{K^- \times H^- \times W^-}$, which comes from the computations in layer L_{i-1}.

- **filter**, the filter weights, $F^{filter} \in \mathbb{R}^{K \times K^- \times d \times d}$, i.e., K 3D filters with each feature dimension $d \times d$ are applied to the ifmap, and
- **ofmap**, the output feature map, $F^{of} \in \mathbb{R}^{K \times H \times W}$, which acts as the ifmap for layer L_{i+1}.

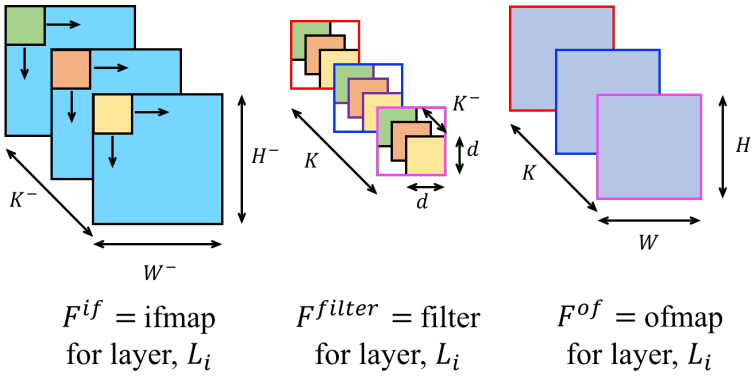


Figure 2.7: Illustration of ifmap, filter, and ofmap in a *Conv* layer.

For a stride size U under bias $F^{bias}[u]$, the computation in layer L_i is given by [14]:

$$F^{of}[u][x][y] = \text{ReLU}\Bigg(F^{bias}[u] + \sum_{k=1}^{K^-}\sum_{i=1}^{d}\sum_{j=1}^{d} F^{filter}[u][k][i][j] \times F^{if}[k][Ux+i][Uy+j]\Bigg) \tag{2.6}$$

where, $1 \le u \le K$, $1 \le x \le H$, $1 \le y \le W$

The k^{th} feature of F^{if} is represented by the 2D feature plane, $F^{if}_k \in \mathbb{R}^{H^- \times W^-}, 1 \le k \le K^-$. The rectified linear unit (ReLU), a nonlinear activation function, introduces nonlinearity into the CNN using $\text{ReLU}(x) = \max(0, x)$. It is typically applied after each *Conv* or *FC* layer. ReLU [40] is the most widely used nonlinear activation function in CNN due to its simplicity and its ability to enable fast training. The ReLU operator introduces a significant amount of sparsity in the data that enables computation reduction in future layers. Numerous other nonlinear activation functions such as traditional sigmoid or hyperbolic tangent as well as several variations of ReLU, such as leaky ReLU, parametric ReLU, and exponential LU, that have also been explored for improved accuracy [14].

FC Layer

An *FC* layer also applies filters on the *ifmaps* as in the *Conv* layers, but the filters are of the same size as the *ifmaps*. Equation (2.6) still holds for the computation of *FC* layers with additional constraints on the shape parameters: $H^- = W^- = d, H = W = 1$, and $U = 1$.

Pool Layer

A *Pool* layer is typically applied after a *Conv* layer to reduce the spatial size of the feature map representation. A *Pool* layer combines a set of values in its receptive field [41] into a smaller number of values using max or average operations. Thus, the numbers of parameters and computation in the network are reduced. This also facilitates the network to be robust and invariant to small shifts and distortions. The pooling operation is applied to each 2D feature plane separately.

Norm Layer

In a normalization (*Norm*) layer, the distribution of the input features are normalized such that the data has a zero mean and a unit standard deviation. Normalizing the input distribution across layers can help significantly speed up training and improve accuracy. Two types of normalizations have been most commonly used in a CNN: local response normalization (LRN) [39], and batch normalization (BN) [42]. However, batch normalization (BN) is now considered standard practice in the design of CNNs.

2.3.2 Network in Network

Conventional CNN topologies implicitly make the assumption that the extracted features of an object are linearly separable and hence linear filters are used to perform convolution and feature extraction [43]. However, the features for the same object often lie on a nonlinear manifold; therefore the representations that capture these features are generally a highly nonlinear function of the input. The use of a richer nonlinear abstraction can serve as a better feature extractor.

The concept of a *network in network* (NIN) has been introduced in [43], which places a mini-neural-network in place of a convolution filter to accommodate higher nonlinear

abstraction. This mini-network increases the representational power of CNNs by introducing additional depth in the network. It is also compatible with the backpropagation algorithm of neural nets; thus this fits well into existing CNN architectures.

GoogLeNet [44], a state-of-the-art network, proposed a new NIN-inspired module *inception* (Fig. 2.8) to improve accuracy. An inception module is composed of multiple parallel connections, unlike a conventional CNN (i.e. LeNet [41], AlexNet [39]), where only a single serial connection exists. In the inception module, three filters of different sizes (i.e., 1×1, 3×3, 5×5) and a 3 x 3 max-pooling are used. The output feature planes, K_1, K_2, K_3, and K_4, of these four paths are concatenated to generate the module output. The use of multiple filter sizes has the effect of processing the input at multiple scales. In addition, intermediate layers with K_1^I and K_2^I number of 1×1 filters are applied before 3 x 3 and 5 x 5 filters, respectively, to reduce the number of computations as well as to increase the depth. The idea of processing the input at multiple scales and using a higher depth is seen to improve the CNN accuracy significantly.

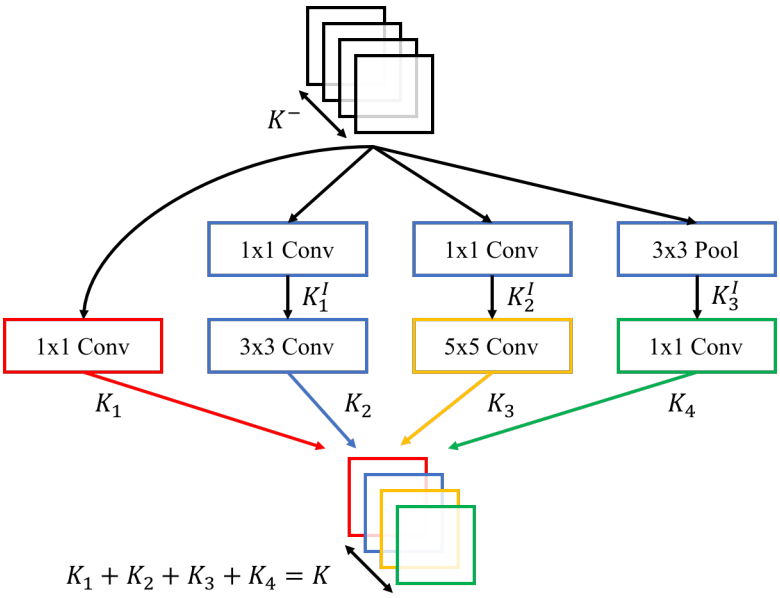


Figure 2.8: Illustration of inception layer in GoogLeNet.

Chapter 3

Static Approximation of JPEG Hardware

A core building block for JPEG compression is the discrete cosine transform (DCT), as discussed in Section 2.2. The amount of computation and power dissipation in a DCT is significant: for a moderate-sized 512×512 (256K) image with 4096 frames, the fast DCT requires nearly 10^6 multiplications and 2×10^6 additions. Several algorithms for implementing the DCT have been proposed to reduce the number of computations without approximations by altering the architecture or/and logic simplifications [36–38] by exploiting the periodic symmetries of cosine terms in the DCT flow graph. However, even for a moderate-size image, the number of computations is large [45]. A few prior methods have explored approximations in JPEG, but perform *ad hoc* optimizations. Some approaches use dynamic bitwidth [7] and dynamic range reduction [46], while others uniformly approximate all adders in the DCT [6]. Other approaches [47, 48] target general RTL structures and do not exploit the structure of JPEG.

We address the problem of optimizing JPEG hardware by approximating the arithmetic units within the JPEG block to reduce the power consumption under an user-specified error budget. This is a worthwhile target for optimization for two reasons. First, JPEG compression is used widely enough for the results to be useful, and second, the concepts developed here could potentially be extended to optimize other structures such as FFTs, MPEGs. We explore two classes of approximations in the DCT block:

(1) *Static* approximation chooses the degree of approximation for various arithmetic units of the DCT block once during system optimization, and maintains the same approximation for all input images. The choice must be conservative to function correctly for any input image. The concept, formulation and results of static approximation in JPEG hardware will be discussed in this chapter.
(2) *Dynamic* approximation depends on the data in a specific image, and generates image-dependent approximations chosen on the fly, depending of the error-resilience of the image. We will discuss dynamic approximation in JPEG hardware in Chapter 4.

3.1 Concept of Static Approximation

We have developed a static optimization framework to optimize the approximation level for each computational unit within the Loeffler network of the DCT architecture (Section 2.2.1). The optimization is carried out over the two layers in Fig. 2.3, each containing eight 1D Loeffler DCT structures (Figure 2.4), each of which processes one column of the M matrix. The full set of optimization variables corresponds to the approximation levels in the 29 adders and 11 multipliers in each of the eight blocks and both levels of the 2D DCT. We denote the full list of adder and multiplier optimization variables by the sets θ_a and θ_m, respectively. This leads to a total of $(29+11)\times 8\times 2 = 640$ variables. However, such an optimization could imply that each 1D DCT block could have a different structure since it may be optimized differently. Since this surrenders the major advantage of regularity that is exploited in layout optimization of DSP blocks, we choose to restrict the optimization so that within each layer, all eight 1D DCT blocks are identical, allowing for easier design and layout. This leads to a reduced number of variables, θ'_a and θ'_m, so that the total number of variables is now $(29 + 11) \times 2 = 80$, thus also reducing the size of the optimization problem.

We consider the impact of using approximate multipliers on the accumulated error at the output of the JPEG compression engine, and its effect on the quality of the resultant image. The error at each node within the DAG has a different contribution to the total error at the final output. We propose a scheme based on the output error sensitivity of each node to optimize the hardware, subject to a user-specified error budget that is typically application-dependent. We proceed in three steps:

- **Computing the error statistics of each unit module:** We characterize the error mean and variance for arithmetic units (adders and multipliers) under static approximation criteria. We assume that the input patterns are uniformly distributed, and this is reasonable over a large number of frames.
- **Sensitivity analysis of the DCT block:** We calculate the sensitivity of each node in a 1D DCT block to the error at the output stage. We use the error sensitivity of each node as an input to the optimization engine.
- **Nonlinear optimization:** We formulate and solve a nonlinear optimization problem to obtain the optimal number of approximated bits for each adder and multiplier.

3.2 Computing Error Statistics

Errors in Adders: For the approximate FA design *appx5*, the error x from approximating α LSBs can take on 2^{α} different values from the set $\{0, \pm 1, \pm 2, \cdots, \pm\left(2^{\alpha-1}-1\right), -2^{\alpha-1}\}$. The approximation scheme in the adder is concentrated in the lower LSBs. Over the large number of frames processed by an adder, the lower LSBs of the input have equal probability of ones and zeros. Using this uniform distribution of error values, i.e., $p_x = 1/2^{\alpha}$, we find the error mean and variance:

$$\mu_a(\alpha) = -0.5; \qquad \sigma_a^2(\alpha) = \left(4^{\alpha}-1\right)/12 \tag{3.1}$$

The mean is independent of number of approximated bits. Even for a 1 LSB approximation, 3σ is 0.75 units and exceeds the mean. The variance is larger for more approximated bits, and the mean can be reasonably approximated as zero.

Errors in Multipliers: In the JPEG computation, each multiplier has one constant and one variable node, as described in Section 2.2. Assuming a uniform input distribution, the error statistics of multipliers are analytically modeled using the MATLAB curve fitting toolbox as a function of the α approximated LSBs as:

$$\sigma_m^2(\alpha) = 0.6135(4^{\alpha}) \tag{3.2}$$

As before, the error mean is essentially zero.

3.3 Sensitivity Analysis of the DCT Block

The computation at each node of a 1D DCT block is a linear combination of the results of its immediate predecessor stage node. The result at node n of stage s, $T_{n,s}$, is:

$$T_{n,s} = \sum_{k=1}^{8} W_{k,n,(s-1)} \times T_{k,(s-1)}, \quad 1 \leq n, s \leq 8 \tag{3.3}$$

where $W_{k,n,(s-1)}$ is the weight of node k of the previous stage, $(s-1)$ for node n of stage s. The value of $W_{k,n,(s-1)}$ can be obtained from the structure of the network, shown in Fig. 2.5. We use $\zeta_{n,s,k}$ to denote the sensitivity of node k of stage $(s-1)$, at node n of stage s. From (3.3),

$$\zeta_{n,s,k} = \frac{\partial T_{n,s}}{\partial T_{k,(s-1)}} = W_{k,n,(s-1)} \qquad \forall n, s, k \in \{1, \cdots, 8\} \tag{3.4}$$

The error variance at a node can be obtained by weighting the variances of predecessor nodes by the sensitivity, and adding the error generated due to an approximation at the node. The error variance, $\sigma^2_{n,s,b,l}$, for node n, stage s, block b, layer l is:

$$\sigma^2_{n,s,b,l} = \sum_{k=1}^{8} \zeta^2_{n,s,k} \times \sigma^2_{k,(s-1),b,l} + \sigma^2_{op,nsl} \tag{3.5}$$

where

$\sigma^2_{op,nsl}$ = Generated variance at node n, stage s, layer l

$\sigma^2_{n,0,b,l}$ = Input variance for block, b, of layer, l

The values of $\sigma^2_{op,nsl}$ can be obtained from (3.1) or (3.2). We propagate the error variance up to the D' node of Fig. 2.2(b) using (3.5) and match it with the error variance budget, $\sigma^2_{budget,D'}$. For a quality factor, F, and a maximum user-specified error budget, δ_R, an empirical relation between $\sigma^2_{budget,D'}$ and $\sigma^2_{budget,R}$ at node R in Fig. 2.2(a), is as follows:

$$\sigma^2_{budget,D'} = \tau \times \sigma^2_{budget,R} \tag{3.6}$$

$$\text{where} \qquad \tau = \frac{F}{100} \times \left[-2.94 \times 10^5 \delta_R^{-2.25} + 250\right]$$

This formula captures the factors that contribute to the difference between $\sigma^2_{budget,D'}$ and $\sigma^2_{budget,R}$, namely, (a) a lower F incurs a larger quality degradation in quantization, and (b) the inputs to the DCT nodes in Fig. 2.5 are not independent, as assumed, due to correlations.

To obtain $\sigma^2_{budget,R}$, we begin with the maximum error δ_R, specified by the user at the 3σ point of the error at R, based on the application requirement. This can be written as

$$\delta_R = \mu_{budget,R} + 3\sigma_{budget,R} \approx 3\sigma_{budget,R} \tag{3.7}$$

where $\mu_{budget,R} \approx 0$ and $\sigma_{budget,R}$ are the mean and standard deviation of the maximum error budget, respectively. Thus,

$$\sigma^2_{budget,R} = (\delta_R/3)^2 \tag{3.8}$$

3.4 The Nonlinear Optimization Formulation

For a reasonable k approximate output LSBs in an array multiplier, the number of FAs is quadratic in k. For an adder, each approximate output bit translates to one approximate FA. Thus, the number of approximate FAs in an adder, $f_a(k)$, and multiplier, $f_m(k)$, are given by:

$$f_a(k) = k \; ; \; f_m(k) = k(k-1)/2 \tag{3.9}$$

Therefore, the total number of FAs that are approximated can be represented as a function of the number of approximated bits associated with each adder or multiplier, as follows:

$$\lambda_{appx} = 8 \times \left(\sum_{a_x \in \theta'_a} f_a(\alpha_{a_x}) + \sum_{m_y \in \theta'_m} f_m(\alpha_{m_y}) \right) \tag{3.10}$$

Here, θ'_a and θ'_m correspond to the reduced sets of adders, a_x, and multipliers, m_y, for the 2D DCT block. As discussed earlier, the same approximation is used within each of the 8 1D DCT blocks in layer, $l = 1, 2$, to preserve layout regularity. The multiplicative factor of 8 captures this notion.

We define a metric, "Percentage FA savings" (PFA), to quantify the percentage savings with respect to the the total FAs, as follows:

$$PFA = \left(\frac{\lambda_{appx}}{\lambda_{total}} \right) \times 100 \tag{3.11}$$

where, λ_{total} is the total number of FAs in the system.

Based on the relations in Sections 3.2 and 3.3, we formulate an optimization problem that maximizes the savings due to approximation, subject to a specified bound on the error introduced at D' node for Fig. 2.2(b). The precise formulation of the optimization problem is:

$$\begin{aligned}
&\max \quad \textstyle\sum_{a_x \in \theta'_a} \alpha_{a_x} + \sum_{m_y \in \theta'_m} \frac{\alpha_{m_y}(\alpha_{m_y}-1)}{2} \qquad (3.12)\\
&\text{s. t. (a) } \forall n, s, b \in \{1, \cdots, 8\}, l \in \{1, 2\},\\
&\qquad \sigma^2_{n,s,b,l} = \sum_{k=1}^{8} \zeta^2_{n,s,k} \times \sigma^2_{k,(s-1),b,1} + \sigma^2_{op,nsl}\\
&\qquad \text{(b) } \sigma^2_{n,0,b,1} = 0 \quad \forall n, b \in \{1, \cdots, 8\}\\
&\qquad \text{(c) } \sigma^2_{n,0,b,2} = \sigma^2_{n,8,b,1} \quad \forall n, b \in \{1, \cdots, 8\}\\
&\qquad \text{(d) } \sigma^2_{n,8,b,2} \leq \sigma^2_{budget,D'} \quad \forall n, b \in \{1, \cdots, 8\}
\end{aligned}$$

The objective function maximizes λ_{appx}, representing the total hardware savings. The constraints represent limits on the error variances at every stage. Constraint (a) is the relationship (3.5) between the error variance from one stage to the next in each layer of the network. Constraints (b) and (c), respectively, state that there is no error at the input of layer 1 of the DCT on the compression side when M is presented, and that the error of stage 8 of layer 1 is passed on to layer 2. The variance budget constraint (d) is obtained from (3.7) and (3.8).

Since the error variance equations of adder and multiplier are nonlinear, as shown in (3.1) and (3.2), both the objective and constraint functions are nonlinear. The decision variables, α_{a_x} and α_{m_y}, for this optimization are integer-valued. Although a mixed integer linear programming problem can be computationally expensive, (a) the number of variables is small enough that a solution is realistic, and (b) since this is an one-time solution at design time, computation time is not critical. In practice, we find that the problem is solved in about 1 hour.

3.5 Results

We synthesized the RTL of the JPEG hardware using the 45nm Nangate library [49] in Synopsys Design Compiler [50] and all simulations have been performed at the typical process corner with V_{dd} = 1.1V and temperature, T = 25^{o}C. All calculations are

performed using 32-bit signed arithmetic. The JPEG circuit was exercised on the benchmark images from [51, 52].

We employ the mixed integer nonlinear problem solver, KNITRO [53], to solve the static nonlinear optimization problem derived in (3.12). The CPU times for optimization were around 1 hour on a 2.6 GHz Intel Core i5 CPU with 8Gb RAM and running the 64-bit OS X. This computation time is reasonable since this is an one-time optimization. The optimized values of α_{a_x} and α_{m_y} obtained from KNITRO were hard-coded into the RTL and synthesized to obtain the area, power and delay of the approximate hardware.

We obtained the PSNR of the JPEG compressed images using a MATLAB simulation according to Eq. (2.5). The base case for PSNR uses quality factor, $F = 90$ with no approximation. Relative to the base case, we compared the PSNR difference, $\Delta PSNR$, for several images, for various F, with or without approximations.

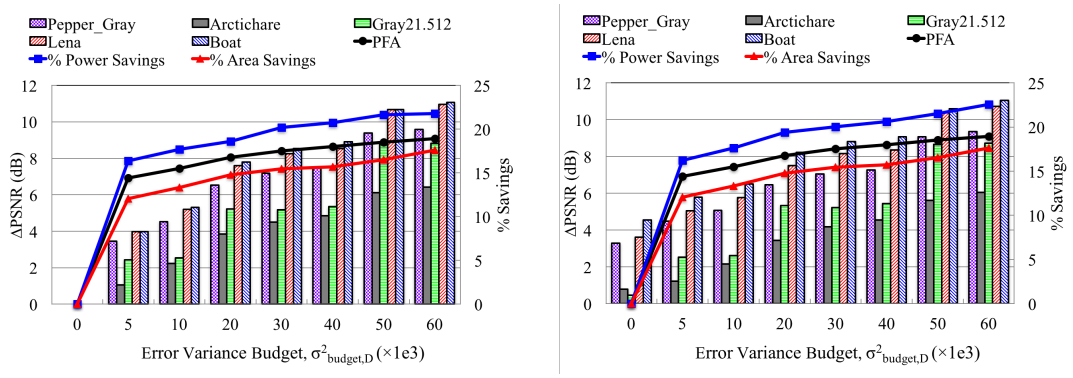


Figure 3.1: PSNR degradation, $\Delta PSNR$, PFA, area and power savings vs. $\sigma^2_{budget,D}$, for quality factor, (a) $F = 90$ and (b) $F = 50$.

We plot the PFA as mentioned in (3.11), power and area savings obtained from a synthesized hardware implementation along with $\Delta PSNR$ vs. error variance budget, $\sigma^2_{budget,D}$, for two values of quality factors, $F = 90$ and $F = 50$ in Fig. 3.1(a) and 3.1(b), respectively. The $\Delta PSNR$ values are plotted on the left axis using bars while the percentage savings for various quantities (area, power and PFA) are plotted on the right axis using points in each graph. The graphs of percentage area savings and PFA correlate well due to the strong correspondence between FA savings and area savings.

A good estimation for percentage change in switching capacitance, C_{sw}, can also be obtained from PFA.

At $\sigma^2_{budget,D} = 0$, no PSNR degradation is observed for $F = 90$, as this is the base case criteria, whereas for $F = 50$, $\Delta PSNR$ is nearly 4 dB. This means a significant amount of error is incorporated in the image without any approximations to achieve higher compression ratio. We can use this PSNR degradation of $\sigma^2_{budget,D} = 0$ for various quality factors, F, as an advantage for achieving power savings with some increased storage requirement. Appropriate F and $\sigma^2_{budget,D}$ are typically provided as inputs based on the application. For example, if $F = 50$ and the acceptable degradation is 4 dB, we cannot approximate any further as 4 dB error is already incorporated due to the quality degradation factor, F. As seen from Figure 3.1(b), the variance budget for this criteria will be zero and there will be no area and power savings. For $F = 90$, $\Delta PSNR = 4$dB budget can be translated to $\sigma^2_{budget,D} = 5,000$. The corresponding area and power savings are 12.0% an 16.2%, respectively as shown in Figure 3.1(a). This implies higher area and power savings are possible using higher F.

Qualitatively, high precision applications should choose higher quality factor, F, with very smaller error budget, $\sigma^2_{budget,D}$, which will still provide around $12 - 14\%$ savings in both area and power. On the other hand, error tolerant applications should choose higher F with higher error budget and save up to 20% in area and power.

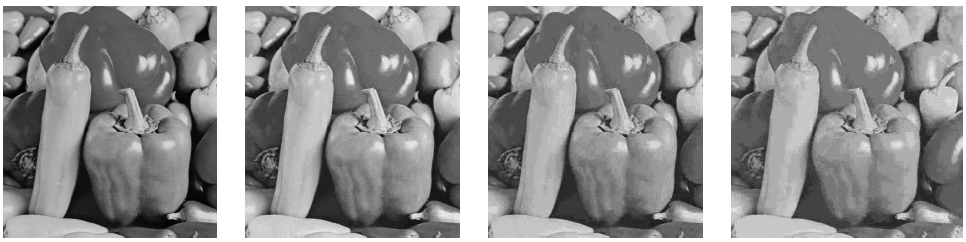
We observe 15% delay improvement for the approximate hardware which can be translated to reduced supply voltage, V'_{dd}. The reduced power, P', for such a voltage-scaled system depends on the global V_{dd}, global clock period, T_c as well as reduced switching capacitance, C'_{sw} as,

$$P' = \frac{C'_{sw} V'^2_{dd}}{2T_c} \tag{3.13}$$

where $C'_{sw} = C_{sw}\,(1 - PFA)$, $V'_{dd} = V_{dd}\left(1 - \frac{\Delta T}{T_c}\right)$ and ΔT is the delay change (slack) due to the approximate hardware. From (3.13), we infer that $\sim 40\%$ power reduction is achievable with voltage scaling with our approximation scheme.

To provide a visual illustration, compressed images with $F = 90$ for four different specified $\sigma^2_{budget,D}$ are shown in Figure 3.2. The PSNR for the base case, shown in Figure 3.2(a), is 35.29 dB. Image qualities with three different error variances are shown in the figure. In spite of the significant approximation very little difference in image

quality is visible.



(a) (b) (c) (d)

Figure 3.2: Image quality for peppers_gray ($F = 90$) with (a) $\sigma^2_{budget,D}$=0, PSNR=35.29dB (b) $\sigma^2_{budget,D}$=1.e4, PSNR=30.76dB (c) $\sigma^2_{budget,D}$=3.e4, PSNR=28.11dB (d) $\sigma^2_{budget,D}$=6.e4, PSNR=25.70dB.

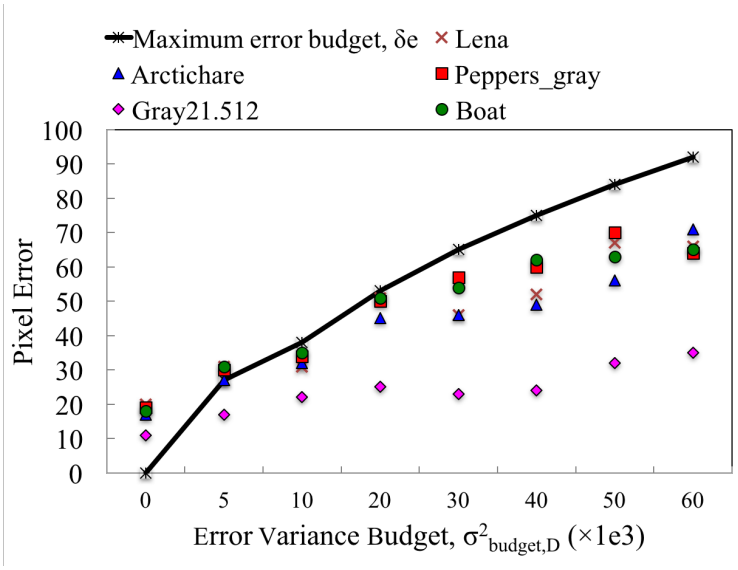


Figure 3.3: Maximum error budget line, δ_R, and the induced maximum pixel error for various images for $F = 90$.

The scatter points in Figure 3.3 represent the maximum pixel error generated for different images incorporating different $\sigma^2_{budget,D}$. For comparison, the relation between $\sigma^2_{budget,D}$ and maximum allowable pixel error, δ_R, shown in (3.8), is also represented in the figure with a line. It is observed that, the maximum error generally remains below the maximum error budget. The error generated at $\sigma^2_{budget,D} = 0$ (without any

approximations) is the lossy JPEG compression error, which is not captured in (3.8).

Finally, we compare the benefit of our approximation scheme, where 80 nodes in the 1D DCT are allowed to have independent approximation levels, versus the case where a fixed number of bits are approximated for all nodes in the DCT. The latter case incorporates higher amount of error than approximating variable bit size error generation, i.e., the optimization uses only one variable.

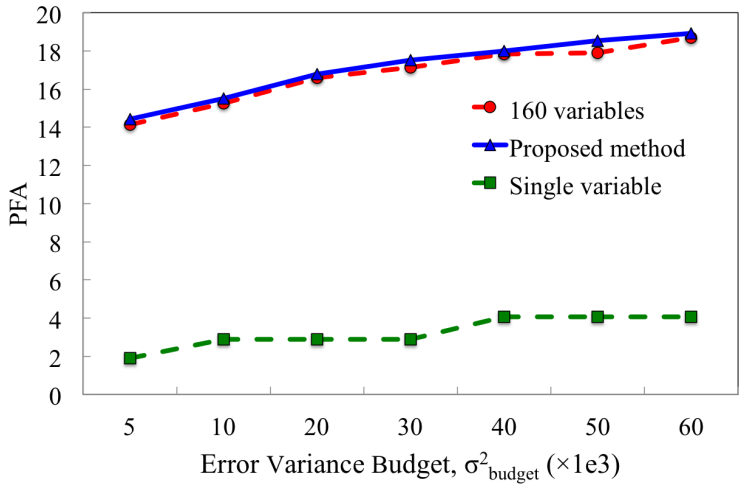


Figure 3.4: Comparison among proposed method, 160 variables and single variable optimization.

As shown in Figure 3.1, *PFA* provides a good estimation of area and power savings, and therefore we use this metric to compare the 80-variable and 160-variable optimization with the single-variable optimization in Figure 3.4. We find that our scheme provides 10% more savings, while the simpler scheme only provides marginal benefits. This is because the sensitivity of many addition and subtraction operations in the DAG is high, and approximation errors are magnified. Our sensitivity-based scheme captures this effect and chooses a lower number of approximate LSBs for high-sensitivity nodes, and more approximate LSBs for low-sensitivity nodes. The result for the 160-variable optimization problem only provides a feasible solution after iterating over 200,000 steps for about three hours. The results of both the 80-variable and 160-variable optimizations provide similar improvements in *PFA*. Increasing the problem space in non-linear optimization leads the solution further away from the optimal point and also increases

the CPU time.

3.6 Conclusion

We have presented an approach that solves a small integer nonlinear program to optimize the power dissipation of a JPEG compression unit. We formulate the problem using the sensitivity of nodes in the DAG that represents the computation and enable a solution that approximates more (fewer) LSBs at low (high)-sensitivity nodes.

Chapter 4

Dynamic Approximation of JPEG Hardware

The image-independent static approximation, presented in Chapter 3, considers the sensitivity of the error at the output for the approximations at the arithmetic units within the block. An optimization problem is solved to maximize power savings while ensuring that the output error lies within a user-specified budget. However, such a scheme is necessarily conservative since it provides a single approximation scheme that is safe for all input images. The differences in the pixel patterns of images imply that the distributions of input values differ from image to image, and the level of approximation can be dynamically modified to be image-specific. This dynamic approximation approach reduces the conservatism of the static approach by tailoring the degree of approximation to the input image. We propose an input-dependent dynamic approximation scheme that is implemented over the static approximation process. A few prior works have explored input-dependent approximations without characterizing the input pixel variations [54, 55]. In our work, the input distributions of different images are estimated in real-time to determine the level of approximations. Run-time dynamic approximation is enabled by the addition of decision circuitry.

In this chapter, we will first discuss the concept of dynamic approximation and its implementation process. The heuristics to simplify the dynamic architecture are later discussed. The result section compares the power, area, delay and quality for various

error targets and images using our framework.

4.1 Concept of Dynamic Approximation

The starting point of dynamic approximation is the static approximation framework developed in Chapter 3. The error variance equations (3.1) and (3.2) assume uniform input probabilities for all adders and multipliers, respectively, but in practice, this approximation provides pessimistic error estimates. Using image-specific input statistics, dynamic approximation can increase the approximation level significantly. However, the extraction of real-time data is a costly operation in terms of power and time as it requires additional circuits to measure the data in real-time. Moreover, translating measurements of image information into increased approximation level in circuits is a nontrivial task. We propose an approximate dynamic architecture that alters the approximation levels in real-time for individual images. The peripheral circuitries, probability distribution block and optimization block help to calculate the input data distribution and optimize the approximate levels, respectively. The following sections discuss the selection of nodes for dynamic approximation, proposed modifications for the simplifications of the dynamic approximation hardware and the power modeling.

4.1.1 Selecting Nodes for Dynamic Approximation

The DCT block has two node types: adders (or subtractors) and multipliers. Considerations for approximation include:

- **Number of image-dependent inputs at the node:** The adder nodes have two image-dependent inputs, while multiplier nodes have one constant and one image-dependent input. Therefore, the data-dependent approximation circuitry is required on both adder inputs, but only one multiplier input, resulting in lower hardware overhead for multipliers.
- **Node error sensitivity to the outputs:** In the DCT DAG, adder nodes are followed by constant multiplier nodes. Thus, an error introduced at the adder node is amplified. No such amplification occurs for approximations in multiplier nodes. As a result, the error sensitivities of the multiplier nodes are always smaller than those of the adder nodes.

- **Number of dynamic FA blocks at the node:** To achieve the maximum dynamic savings within the limited error budget, the number of approximated FA bits must be maximized. According to (3.9), incrementing the approximation level by one bit for an adder and a multiplier implies approximating $f_a(k+1)-f_a(k) = 1, f_m(k+1)-f_m(k) = k$ FAs, respectively. Therefore, more FAs are available, per bit of approximation, for multipliers than adders.

All of these factors imply that the dynamic approximation of multiplier nodes is more effective than adder nodes.

We now choose an approximate FA design from *appx1–appx5* [6] for dynamic approximation. We find that *appx5* is the best choice for dynamic approximation because:

- From Fig. 2.1, *appx5* is the most power-efficient option.
- The outputs of *appx5* only depend on the inputs a and b, not on the carry-in bit, c_{in} [6]. Therefore, the scope of propagated errors is limited since an error generated at the carry-out at a node is not sent to the next bit.
- The *appx5* design provides accurate results for two crucial combinations, $(a, b, c_{in}) =$ (000) and (111). This is particularly important because many inputs do not use the entire bit width, but use sign-extension for their most significant bits (MSBs). The correct computation of these two combinations ensures that no errors are generated in processing sign-extension MSBs. The computation of multiplication involves taking the summation of the partial products after proper shift operation. If we only take summations of non-zero partial products, the sign extensions of all non-zero partial products are the same and thus according to the property of *appx5*, no error will be generated. If Booth's multiplication is used (which is not the case in this thesis), then the signs will be mixed and this property will not hold.

Let us consider two n-bit multiplier inputs where the lower x and y bits, respectively, contain useful data, and the MSBs are sign-extension bits. If we approximate $\alpha \geq x + y$ bits using *appx5*, the error in the multiplier output is limited to an $(x + y)$-bit approximation, rather than an α-bit approximation, since the MSBs are correctly computed. For $\alpha < x + y$, the maximum error corresponds to α bits. In contrast, for *appx1–appx4*, the correctness of sign-extension does not hold, either because an error is generated for the input bit combinations and/or because an error in an input carry is propagated to the sign-extension MSBs. Hence, the maximum error corresponds to α bits regardless

of $x + y$. This is summarized in Table 4.1.

Table 4.1: Error characteristics of approximate multipliers.

	Number of potentially erroneous bits	
	appx5 based multiplier	*appx1–appx4* based multiplier
$\alpha \geq x + y$	$x + y$	α
$\alpha < x + y$	α	α

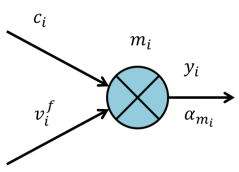
4.1.2 Modification of Multiplier Error Variance Formulation

For a multiplier node, m_i, in Fig. 4.1, let y_i be its output, c_i and v_i^f its constant input and its variable input in frame f, respectively, α_{m_i} the number of bits used for static approximation, and $\sigma_m^2(\alpha_{m_i})$ its error variance, obtained from (3.2). When the variable input of the *appx5* based multiplier satisfies

$$\alpha_{m_i} \geq \log_2(|v_i^f|) + \log_2(c_i), \tag{4.1}$$

its error characteristics, shown in Table 4.1, reveal two insights:

- Full approximation of all bits of the multiplier keeps the error unchanged, while saving considerable power.
- The true error variance, $\sigma_{m,true}^2(\alpha_{m_i})$, due to the static approximation is smaller than $\sigma_m^2(\alpha_{m_i})$, and, for both the static and dynamic cases, is given by $\sigma_m^2(\lceil \log_2(|v_i^f|) + \log_2(c_i) \rceil)$.



Multiplier node	m_i
Constant input	c_i
Variable input	v_i^f
Multiplier output	y_i
Static approximation bit-width	α_{m_i}

Figure 4.1: A typical multiplier processing data in frame f.

To obtain a criterion based purely on v_i^f, we relax the criterion of (4.1) to define the bound $\mathcal{N}_i$:

$$\alpha_{m_i} \geq \lceil \log_2(|v_i^f|) \rceil + \lceil \log_2(c_i) \rceil \tag{4.2}$$

$$\text{i.e., } \lceil \log_2(|v_i^f|) \rceil \leq \mathcal{N}_i = (\alpha_{m_i} - \lceil \log_2(c_i) \rceil) \tag{4.3}$$

Let us assume, α_{m_i} and $z_i \in \mathbb{Z}_{\geq 0}$ are the static and additional dynamic approximate bits, respectively, for the multiplier node m_i. We can model the true introduced error variance, $\sigma^2_{m,true}(\alpha_{m_i} + z_i)$, as the weighted sum of the error variances of various approximate bits, where the weights are the probabilities of various input ranges:

$$\sigma^2_{m,true}(\alpha_{m_i} + z_i) = \gamma_1(z_i) + \gamma_2(z_i) \tag{4.4}$$

$$\begin{aligned}
\text{where } \gamma_1(z_i) &= \textstyle\sum_{k=1}^{\mathcal{N}_i+z_i-1} P_k \sigma^2_m(k + \lceil \log_2(c_i) \rceil) \\
\gamma_2(z_i) &= \left[1 - \textstyle\sum_{k=0}^{\mathcal{N}_i+z_i-1} P_k\right] \sigma^2_m(\alpha_{m_i} + z_i) \\
P_k &= \begin{cases} \text{Probability of } |v_i^f| = 0, & k = 0 \\ \text{Probability of } 2^{k-1} \leq |v_i^f| < 2^k, & k \in \mathbb{N} \end{cases}
\end{aligned}$$

The exact evaluation of these probabilities is computationally prohibitive since it requires the entire image data to be processed. However, good estimates of the probability can be obtained by sampling the frames at a rate, $1 : R_{sample}$, that provides an appropriate balance between accuracy and computation. We compared the original probability distribution (without sampling) of the multiplier with the distribution achieved from various sample rates. We empirically find that $R_{sample} = 16$ provides a good match between the original and sampled distributions, while also providing a significant reduction in computation. The sampling framework is implemented by randomizing all N_f image frames and the probabilities are computed using first N_f/R_{sample} frames that act as representative of remaining frames. To obtain P_k probability values, we have designed a probability calculation block, which we will discuss shortly.

4.1.3 Dynamic Approximation Hardware

The dynamic approximate architecture for each multiplier node, m_i, consists of following three blocks:

1. the dynamic multiplier block
2. the probability calculation block
3. the optimization block

We will discuss about the hardware description of the dynamic multiplier block and the probability calculation block in this section. The optimization block will be discussed in Section 4.2.

The multiplier m_i is designed based on a CSA-based configuration, as mentioned in Section 2.1, and we use only the lower triangle of this architecture. If the bit width of the multiplier is N_{m_i}, then it has N_{m_i} columns of FAs, and the x^{th} of these columns contains x FAs. The total number of FAs in the multiplier is:

$$\lambda_{m_i} = \sum_{x=0}^{N_{m_i}-1} x \tag{4.5}$$

In static approximation, the lower α_{m_i} columns use approximate FA blocks and rest of the columns employ accurate FAs. For dynamic approximation, the accurate FA columns are replaced with the columns of dynamic select FA (DSFA) blocks. An additional absolute comparator block is added to each dynamic multiplier to compare between v_i^f and $2^{\mathcal{N}_i}$.

Based on sampled image-specific data obtained from the probability calculation block, the optimization block chooses an appropriate z_i. The absolute comparator block checks whether for the specific image, $|v_i^f| \leq 2^{\mathcal{N}_i+z_i}$. If so, it asserts the signal, $\mathcal{S}_i^f(z_i)$ and triggers dynamic full bit-width approximation. This full bit-width approximation increases the error variance to $\sigma^2_{m,true}(\alpha_{m_i} + z_i)$ as mentioned in (4.1). The formulation to obtain selector signal per frame, $\mathcal{S}_i^f(z_i)$, is

$$\mathcal{S}_i^f(z_i) = \begin{cases} 1, & \text{if } |v_i^f| < 2^{(\mathcal{N}_i+z_i)} \\ 0, & \text{otherwise} \end{cases} \tag{4.6}$$

The capability of dynamically changing the level of approximation is enabled through the use of the DSFA block, shown in Fig. 4.2(a), consisting of an accurate FA, an approximate FA, and a MUX that chooses between the two, based on the selector signal, $\mathcal{S}_i^f(z_i)$ [55]. When the approximate FA is in operation, the accurate FA block is power-gated to reduce the power. An optimized DSFA design has been proposed, shown in Fig. 4.2(b), that skips the MUX delay of the accurate mode signals and thus reduces the critical delay as well as the total power of the DSFA design.

During dynamic approximation, the selector signal, $\mathcal{S}_i^f(z_i)$, decides between two operating modes: static approximation and full bit-width approximation. If $\lambda_{m_i,s}$ and

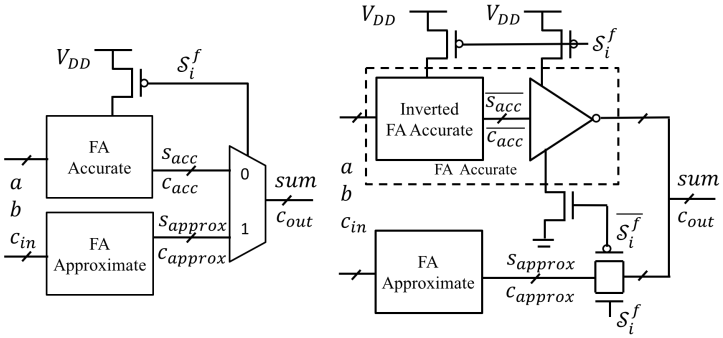


Figure 4.2: (a) Preliminary and (b) optimized structure of a dynamic select FA (DSFA).

$\lambda_{m_i,d}$ are the number of FAs selected for static and dynamic approximation alone, respectively, then for an α_{m_i}-bit approximation,

$$\lambda_{m_i,s} = \sum_{x=0}^{\alpha_{m_i}-1} x \quad , \quad \lambda_{m_i,d} = \sum_{x=\alpha_{m_i}}^{N_{m_i}-1} x \tag{4.7}$$

For each N_{m_i}-bit dynamic multiplier node, m_i, we have designed a probability calculation block. The block has total N_{m_i} counters with the bitwidth $\log_2(N_f/R_{sample})$. Each k^{th} counter is enabled by a logic circuit that detects whether the input signal lies in the range $2^{k-1} \leq |v_i^f| < 2^k$.

4.1.4 Power Modeling

A list of variables used to model the power dissipation of various blocks under a set of static and dynamic approximation schemes is provided in Table 4.2. For the accurate and statically approximated multiplier, the power expressions are:

$$\mathcal{P}_{m_i}^1 = \sum_f p_1 \cdot \lambda_{m_i} = \sum_f p_1[\lambda_{m_i,s} + \lambda_{m_i,d}] \tag{4.8}$$

$$\mathcal{P}_{m_i}^2 = \sum_f p_2 \cdot \lambda_{m_i,s} \tag{4.9}$$

To compute the multiplier power for the dynamic approximation case, we begin with modeling the DFSA. The DFSA operates in the accurate mode except when the selector signal, $\mathcal{S}_i^f(z_i)$, is asserted. The power dissipation in the f^{th} frame, $\phi_{m_i}^f(z_i)$, for the DSFA

Table 4.2: A list of notations for various power terms.

Notation	Explanation
$\mathcal{P}^x_{a_i}$ $x = \{1, 2\}$	Power of an adder at node a_i, operating under the following modes $x = 1$: Accurate N_{a_i}-bit adder $x = 2$: Static approximate α_{a_i}-bit adder
$\mathcal{P}^x_{m_i}$ $x = \{1, 2, 3\}$	Power of a multiplier at node m_i, operating under the following modes $x = 1$: Accurate N_{m_i}-bit multiplier $x = 2$: Static approximate α_{m_i}-bit multiplier $x = 3$: Dynamic approximate $(N_{m_i} - \alpha_{m_i})$-bit multiplier
p_x $x = \{1, 2, 3, 4\}$	Power of an FA operating under the following modes $x = 1$: Accurate-mode FA $x = 3$: Accurate-mode DSFA $x = 2$: Approximate-mode FA $x = 4$: Approximate-mode DSFA
$\Delta\mathcal{P}^x_{m_i}$ $x = \{2, 3\}$	Power savings for various approximation modes implementations at multiplier node m_i $x = 2$: Static approximation $x = 3$: Dynamic approximation
$\Delta\mathcal{P}^x_{DCT}$ $x = \{2, 3\}$	Power savings for various approximation modes for 2D DCT block over all frames $x = 2$: Static approximation $x = 3$: Dynamic approximation
ϕ^f_i	Power of the DSFA in multiplier m_i for frame f
$\mathcal{P}_{abscomp}$	Power of the absolute comparator block
$\mathcal{P}_{opt}$	Power of the optimization block
$\mathcal{P}_{prob}$	Power of probability calculation block
$\Delta\mathcal{P}_{a_i}$	Total power savings for adder node, a_i
$\Delta\mathcal{P}_{m_i}$	Total power savings for multiplier node, m_i

is given by:

$$\phi^f_{m_i} = \Big(1 - \mathcal{S}^f_i(z_i)\Big) \cdot p_3 + \mathcal{S}^f_i(z_i) \cdot p_4 \tag{4.10}$$

The larger the time $\mathcal{S}^f_i = 1$, the lower is the power for DSFA block, which depends on larger z_i, according to (4.6). The power for a dynamically approximated multiplier is:

$$\mathcal{P}^3_{m_i} = \sum_f \Big[\phi^f_{m_i}(z_i) \cdot \lambda_{m_i,d} + \mathcal{P}_{abscomp}\Big] + \mathcal{P}_{overhead} \tag{4.11}$$

where $\mathcal{P}_{abscomp}$ is the power dissipation of the absolute comparator. The last term is the overhead of dynamic optimization:

$$\mathcal{P}_{overhead} = \left(\frac{N_f}{R_{sample}}\right) \cdot \mathcal{P}_{prob} + \mathcal{P}_{opt} \tag{4.12}$$

where $\mathcal{P}_{prob}$ and $\mathcal{P}_{opt}$ are, respectively, the power dissipation of the probability calculation block and the optimization block, discussed in Section 4.2. In (4.11) and (4.12), note that the dynamic multiplier block operates in all N_f frames of the image, the probability calculation block on (N_f/R_{sample}) frames, and the optimization block only once on the entire image.

We launch dynamic approximation from the static approximation case. The power savings over the accurate case are:

$$\begin{aligned} &\Delta\mathcal{P}_{m_i} = \mathcal{P}^1_{m_i} - [\mathcal{P}^2_{m_i} + \mathcal{P}^3_{m_i}] = \Delta\mathcal{P}^2_{m_i} + \Delta\mathcal{P}^3_{m_i} \\ &\text{where} \quad \Delta\mathcal{P}^2_{m_i} = \textstyle\sum_f \lambda_{m_i,s} \cdot (p_1 - p_2) \\ &\qquad\quad \Delta\mathcal{P}^3_{m_i} = \textstyle\sum_f \left[\lambda_{m_i,d} \cdot \left(p_1 - \phi^f_{m_i}(z_i)\right) - \mathcal{P}_{abscomp}\right] - \mathcal{P}_{overhead} \end{aligned} \tag{4.13}$$

The terms $\Delta\mathcal{P}^2_{m_i}$ and $\Delta\mathcal{P}^3_{m_i}$ indicate, respectively, the power savings of static and dynamic approximation. The former is constant over all frames, but the latter varies with each frame.

The total dynamic power savings in the DCT block is the sum of power savings over all dynamic multiplier blocks:

$$\Delta\mathcal{P}^3_{DCT} = \textstyle\sum_{m_i \in \theta_m} \Delta\mathcal{P}^3_{m_i} \tag{4.14}$$

where θ_m is the set of all multipliers in the 2D DCT block.

4.2 Optimized Dynamic Approximation

4.2.1 Formulation of the Optimization Problem

The goal of optimized dynamic approximation is to perform a set of inexpensive optimizations in real time. As a result, it is important for the optimizations to be simple. We achieve this by *locally* maximizing the power savings, $\Delta\mathcal{P}^3_{m_i}(z_i)$, at each node of the DCT operation. The optimization operates the dynamic multiplier nodes with the highest power savings subject to error constraints that guarantee output quality. At each multiplier node m_i, we solve the problem:

$$\begin{aligned} &\max_{z_i} \quad \Delta\mathcal{P}^3_{m_i}(z_i) \\ &\text{subject to} \quad \sigma^2_{m,true}(\alpha_{m_i} + z_i) \leq \sigma^2_m(\alpha_{m_i}) \end{aligned} \tag{4.15}$$

From (4.13), the power savings, $\Delta\mathcal{P}^3_{m_i}(z_i)$, are maximized when $\phi^f_{m_i}(z_i)$ is as small as possible, and from (4.6) and (4.10), this occurs when z_i is maximized. However, the value of z_i is limited by the error variance budget, $\sigma^2_m(\alpha_{m_i})$. Here, the true error variance, $\sigma^2_{m,true}(\alpha_{m_i}+z_i)$, is defined by (4.4). Therefore, we simplify the optimization problem to:

$$\max \quad z_i, \quad \text{s.t.} \quad \sigma^2_{m_i,true}(\alpha_{m_i}+z_i) \leq \sigma^2_{m_i}(\alpha_{m_i}) \tag{4.16}$$

4.2.2 Design Space Reduction for the Optimization Problem

A Motivating Example

A key question relates to the choice of the error budget, $\sigma^2_m(\alpha_{m_i})$, at each multiplier in (4.16). One way to set the budget is to choose the value from static optimization, allowing each node to be dynamically configurable, but this requires substantial overhead circuitry at each node. Instead, we limit configurability to the nodes with the largest potential for power savings, and increase their error budgets to utilize the unused error budget from the other nodes.

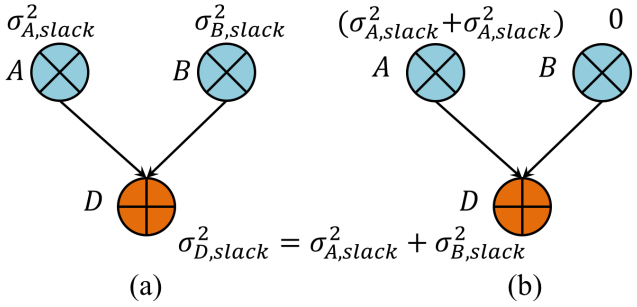


Figure 4.3: The output error variance slack (a) without and (b) with the benefit of error variance stacking.

To understand the relation between error at multiple nodes in a DAG, consider the example DAG where two multiplier nodes, A and B, converge at an exact adder node D. We define the error slack variance, $\sigma^2_{m_i,slack}$, as:

$$\sigma^2_{m_i,slack} = \sigma^2_m(\alpha_{m_i}) - \sigma^2_{m,true}(\alpha_{m_i}) \tag{4.17}$$

$$= (1-\eta_{m_i})\,\sigma^2_m(\alpha_{m_i}) \tag{4.18}$$

where $\eta_{m_i} = \sigma^2_{m,true}(\alpha_{m_i}) / \sigma^2_m(\alpha_{m_i})$. The error slack variance at nodes A and B from the static approximation budget (Fig. 4.3(a)) could be insufficient to allow an extra approximate bit at either node. However, the total error variance at the output is the sum of the error variances at all nodes in the DAG, and we could stack the error variances by allocating the entire slack budget to node A, as in Fig. 4.3(b), i.e., A would be the only dynamically configurable node, and the error slack of B would be transferred to A. This has following benefits:

- It may allow A to further approximate one or more bits, saving power while remaining within the slack budget.
- The hardware overhead required for dynamic reconfigurability is halved as only A is made reconfigurable, and B is approximated statically without the overhead.

A Framework for Redistributing Slack Budgets

We have implemented this slack redistribution idea for the DAG model of the 1D DCT structure shown in Fig. 2.5. There are 11 multiplier nodes, m_1 through m_{11}, and eight outputs, x_0 through x_7, in the 1D DCT structure. The candidates for dynamic approximation are all the multiplier nodes. We denote the sensitivity of the error at node x_j for the error at m_i as S_{ij} and the corresponding sensitivity of the variance of the error as S^2_{ij}. For each (multiplier, output) pair, (m_i, x_j), of the DCT structure, the error variance sensitivity, S^2_{ij}, is shown in Table 4.3. A blank entry at (m_i, x_j) implies no connection between m_i and x_j. Outputs x_0 and x_4 have no multipliers in their fan-in cone and are not shown. The three distinct sensitivity values, s_1, s_2 and s_3, are listed below the table. Let us use $\mathcal{M}_j$ to denote the set of multiplier nodes that are on a path to output x_j. The total error variance introduced by the static approximation process at x_j is:

$$\begin{aligned} \sigma^2_{x_j,stat} &= \textstyle\sum_{i \in \mathcal{M}_j} S^2_{i,j} \cdot \sigma^2_m(\alpha_{m_i}) \\ &= \textstyle\sum_{i \in \mathcal{M}_j} S^2_{i,j} \cdot \left[\sigma^2_{m,true}(\alpha_{m_i}) + \sigma^2_{m_i,slack}\right] \end{aligned} \tag{4.19}$$

where the second equality follows from (4.17). As indicated in the example above, we may use more approximation and also save on the overhead of dynamic reconfiguration by making only a few nodes dynamically reconfigurable and stacking the slacks of the

Table 4.3: Variance sensitivities for multipliers in 1D DCT block.

$S^2_{i,j}$	x_1	x_2	x_3	x_5	x_6	x_7
m_1	s_1		s_2			s_1
m_2	s_1			s_2		s_1
m_3	s_1		s_2			s_1
m_4	s_1			s_2		s_1
m_5	$2s_1$		s_2	s_2		$2s_1$
m_6	$2s_1$		s_2	s_2		$2s_1$
m_7					s_1	
m_8		s_1				
m_9		s_1			s_1	
m_{10}			s_3			
m_{11}				s_3		

$s_1 = \frac{1}{2^{14}}, s_2 = \frac{181^2}{2^{28}}$ and $s_3 = \frac{1}{2^{28}}$

other nodes on to these nodes. We therefore partition the elements of $\mathcal{M}_j$ into a set of nodes that are selected for dynamically approximation, $\mathcal{M}_{j,d}$, and a set of nodes, $\mathcal{M}_{j,s}$, that are left at their static approximation levels.

As we redistribute the slacks of the static nodes to the dynamic nodes, we associate each dynamic node, $m_i \in \mathcal{M}_{j,d}$, with a set of static nodes, $\mathcal{Y}_{m_i}$. As we stack the slack from the static nodes to the dynamic nodes, the variance at output x_j becomes:

$$\begin{aligned}
\sigma^2_{x_j,dyn} &= \sum_{i\in\mathcal{M}_{j,d}} \left[S^2_{i,j}\cdot\sigma^2_m(\alpha_{m_i}) + \sum_{m_k\in\mathcal{Y}_{m_i}} S^2_{k,j}\cdot\sigma^2_{m_k,slack} \right] \\
&\quad + \sum_{i\in\mathcal{M}_{j,s}} S^2_{i,j}\cdot\sigma^2_{m,true}(\alpha_{m_i}) \\
&= \sum_{i\in\mathcal{M}_{j,d}} S^2_{i,j}\cdot \left[\sigma^2_m(\alpha_{m_i}) + \sum_{m_k\in\mathcal{Y}_{m_i}} \chi_{m_i}[m_k]\cdot\sigma^2_{m_k,slack} \right] \\
&\quad + \sum_{i\in\mathcal{M}_{j,s}} S^2_{i,j}\cdot\sigma^2_{m,true}(\alpha_{m_i})
\end{aligned} \tag{4.20}$$

where the weights $\chi_{m_i}[m_k] = S^2_{k,j}/S^2_{i,j}$.

The above equation provides a new error budget, $\sigma^2_{m_i,NB}$, for nodes $m_i \in \mathcal{M}_{j,d}$,

which can be simplified using the approach provided in Appendix A:

$$\sigma^2_{m_i,NB} = \tilde{\mathcal{C}}_{m_i} - \tilde{\mathcal{D}}_{m_i} \times \sigma^2_{m,true}(\alpha_{m_i}) \tag{4.21}$$

Here, $\tilde{\mathcal{C}}_{m_i}$ and $\tilde{\mathcal{D}}_{m_i}$ are error-budget-dependent constants for node, m_i. The new budget, $\sigma^2_{m_i,NB}$, is used to update the optimization problem from (4.16) to:

$$\max \quad z_i, \quad \text{s.t.} \quad \sigma^2_{m,true}(\alpha_{m_i} + z_i) \leq \sigma^2_{m_i,NB} \tag{4.22}$$

We propose two following algorithms to reduce the design space of the optimization problem:

- A heuristic described in Algorithm 2 selects a set of dynamically approximated nodes, Λ.
- For each multiplier $m_i \in \Lambda$, Algorithm 3 identifies the set of non-dynamic nodes, $\mathcal{Y}_{m_i}$, whose slacks are stacked on to dynamic node m_i, and the set of weights, χ_{m_i}, associated with stacking, as described in (A.2) of Appendix A.

An Algorithm for Selecting Dynamic Multiplier Nodes

The criteria for selecting a set of dynamic nodes ensure that:

1. the chosen nodes cover all outputs, i.e., the propagated errors reach all outputs (with the exception of x_0 and x_4, which cannot be reached by any multiplier).
2. a goodness metric, ω_{PB}, explained in Appendix B, ensures high approximation levels with low error, is minimized.
3. the error at the outputs of the 1D DCT block is within the total error budget, thus ensuring that dynamic approximation saves power while staying within error specifications.

The inputs to Algorithm 2 are all the multiplier nodes in the 1D DCT network and their corresponding dynamic error variance sensitivities defined in Appendix B. If two nodes have identical row entries in Table 4.3, we discard one of them from consideration for Λ, as shown in Line 3–5 of Algorithm 2. For example, since the rows corresponding to m_1 and m_3 nodes are identical, m_1 could be pruned out. The rationale for this is

Algorithm 2 Selection of Dynamic Multiplier Nodes, Λ

INPUT: Multiplier nodes in the 1D DCT network, $\{m_1,\ m_2, \cdots,\ m_{11}\}$ and the corresponding dynamic error variance sensitivities: $\{\omega_1,\ \omega_2,\ \cdots,\ \omega_{11}\}$.

OUTPUT: A set, Λ, of dynamically approximated nodes.

$\Lambda = \emptyset$
$S_m = \{m_1,\ m_2, \cdots,\ m_{11}\}$
if $m_i, m_j \in S_m, i \neq j$ have identical rows in Table 4.3 **then**
    $S_m = S_m \setminus m_j$
end if
Sort S_m, arranging m_is in non-ascending order of ω_i.
for each multiplier node $m_i \in S_m$ **do**
    if Discarding m_i loses control on any output **then**
        $\Lambda = \Lambda \cup m_i$
    end if
end for
$\omega_{min} \leftarrow \infty$
for $\nu = |S_m|$ to 1 **do**
    Calculate $\omega_{PB} = \left(\sum_{k=1}^{\nu} \omega_{S_m(k)}\right)/\nu$
    if $\omega_{PB} < (1+\epsilon) \times \omega_{min}$ **then**
        $\omega_{min} = \omega_{PB}$, $\Lambda = \Lambda \cup S_m(\nu)$
    end if
end for
return

based on the large overhead of creating a DSFA to allow reconfigurability: even though the error increases exponentially with z_i, for real test cases, the break-even point where the benefits of approximating more multipliers under an error budget overcomes this overhead is well beyond the range of the eventually selected values of z_i.

Line 6 sorts the elements m_i of set S_m in non-ascending order of ω_i. Next, in Lines 7–11, we identify any nodes that can uniquely influence some output x_j, i.e., if these nodes are not included in Λ, x_j cannot be reached. Based on Criterion 1) above, such nodes are added to Λ. After this step, we greedily select from the remaining nodes in non-decreasing order of ω_i. In Lines 13–17, we add a node to Λ if it reduces ω_{PB}, using a factor of ϵ to allow temporary cost increases that permit a level of hill-climbing during optimization. The dominating operation of this algorithm is to check the influence of each candidate multiplier node to all the output nodes. Hence, the complexity of this algorithm is $\mathrm{O}(\mathcal{T}_x \cdot \mathcal{T}_m)$. Here, $\mathcal{T}_x$ and $\mathcal{T}_m$ are the total number of output and multiplier nodes in a DAG, respectively.

An Algorithm for Redistributing Error Slacks

Algorithm 2 yields $\Lambda = \{m_3, m_4, m_7, m_8, m_{10}, m_{11}\}$, i.e., 6 of 11 multiplier nodes are chosen for dynamic approximation.

Further practical considerations are used to prune the number of candidate multiplier nodes over the entire JPEG hardware. As shown in Fig. 2.3, there are two layers of 1D DCT blocks in the 2D DCT, and the constant multiplications in the second layer amplify any error introduced by the first layer of 1D DCT block. This implies that there is limited advantage in implementing dynamic approximation for the multiplier nodes of Layer 1, and therefore, we only choose multiplier nodes from Layer 2 for dynamic approximation. In summary, we choose a total of $6 \times 8 = 48$ multiplier nodes, listed in a set, θ''_m, for dynamic approximation out of $11 \times 16 = 176$ available multiplier nodes.

The inputs to the Algorithm 3, $\mathcal{M}_{j,d}$, $\mathcal{M}_{j,s}$, and $\mathcal{O}_{m_i}$, and the dynamic node set, Λ, can be obtained from Table 4.3 and Algorithm 2. After the initialization in Line 1, Line 2 creates $\mathcal{I}_{m_i}$, the set of nodes that are candidates for stacking on m_i: these are nodes that lie on every output path, $\mathcal{O}_{m_i}$, that m_i can reach. The rationale for this choice can be illustrated by an example: node $m_3 \in \Lambda$ reaches $\mathcal{O}_{m_3} = \{x_1, x_3, x_7\}$, and the non-dynamic nodes that lie on every path to nodes in this set are $\mathcal{I}_{m_3} = \{m_1, m_5, m_6\}$.

Algorithm 3 Algorithm to obtain $\mathcal{Y}_{m_i}$ and χ_{m_i}

INPUT: Λ = Set of selected multiplier nodes,
$\mathcal{M}_{j,d}$ = Set of nodes $m_i \in \Lambda$ that have a path to output x_j
$\mathcal{M}_{j,s}$ = Set of nodes $m_i \notin \Lambda$ with a path to output x_j
$\mathcal{O}_{m_i}$ = Set of outputs x such that $m_i \in \Lambda$ has a path to x
OUTPUT: $\mathcal{Y}_{m_i}$= Nodes whose error slacks are stacked to $m_i \in \Lambda$
$\chi_{m_i} = \text{map}\left\langle \text{key} = m_k, \text{value} = \frac{S^2_{k,j}}{S^2_{i,j}} \right\rangle, m_k \in \mathcal{Y}_{m_i}, m_i \in \Lambda$

Initialize $\mathcal{Y}_{m_i} \leftarrow \varnothing, \chi_{m_i} \leftarrow \varnothing, V \leftarrow \varnothing \ \ \forall m_i \in \Lambda$
Create $\mathcal{I}_{m_i} = \bigcap_{x_j \in \mathcal{O}_{m_i}} \mathcal{M}_{j,s}, \quad \forall m_i \in \Lambda$
for each $x_j \quad j \leftarrow \{0, 1, \cdots, \mathcal{T}_x - 1\}$ **do**
    Calculate $\sigma^2_{x_j,dyn}$ using (4.20)
    if $\sigma^2_{x_j,dyn} > \sigma^2_{x_j,stat}$ **then**
        $\chi_{m_i}[m_k] = \min\left(\chi_{m_i}[m_k], \frac{S^2_{k,j}}{S^2_{i,j}}\right),$
                $\forall m_k \in \mathcal{Y}_{m_i}, m_i \in \mathcal{M}_{j,d} \cap V$
    end if
    $\mathcal{M} = \mathcal{M}_{j,d} \setminus V$
    $\mathcal{I}_{m_i} = \mathcal{I}_{m_i} \setminus \cup_{m_i \in \mathcal{M}} \mathcal{Y}_{m_i}, \quad \forall m_i \in \mathcal{M}$
    Calculate $\mathcal{Q}_j = \bigcap_{m_i \in \mathcal{M}} \mathcal{I}_{m_i}$
    Assign $\mathcal{Y}_{m_i} \leftarrow \mathcal{I}_{m_i} \setminus \mathcal{Q}_j, \quad \forall m_i \in \mathcal{M}$
    $Y[m_i] = \mathcal{Y}_{m_i}, \quad \forall m_i \in \mathcal{M}$
    $Y = \text{NodeDist}(Y, \mathcal{Q}_j, \mathcal{M})$
    $\mathcal{Y}_{m_i} = Y[m_i], \quad \forall m_i \in \mathcal{M}$
    $\chi_{m_i}[m_k] = S^2_{k,j}/S^2_{i,j}, \forall m_k \in \mathcal{Y}_{m_i}, \forall m_i \in \mathcal{M}$
    $V \leftarrow V \cup \mathcal{M}$
end for
return

Algorithm 4 Y =NodeDist$(Y, \mathcal{Q}_j, \mathcal{M})$

if $\mathcal{Q}_j \neq \varnothing$ **then**
    Find $\sigma^2_{x_j,m_k}$, $m_k \in (\cup_{m_i \in \mathcal{M}} Y[m_i]) \cup \mathcal{Q}_j$, using (B.1)
    $TS[m_i] = \sum_{m_k \in Y[m_i]} \sigma^2_{x_j,m_k}, \forall m_i \in \mathcal{M}$
    Sort $\mathcal{Q}_j$ in descending order of $\sigma^2_{x_j,m_k}, m_k \in \mathcal{Q}_j$
    for $m_k \in \mathcal{Q}_j$ **do**
        $m_l \leftarrow \text{argmin(TS)}$
        $Y[m_l] \leftarrow Y[m_l] \cup \{m_k\}$
        $TS[m_l]$ += $\sigma^2_{x_j,m_k}$
    end for
end if
return Y

Hence, stacking the error slacks of m_1, m_5, or m_6 to m_3 is merely a redistribution of slacks that will not result any error constraint violation. Now consider node $m_2 \notin \mathcal{I}_{m_3}$, which lies on the path to x_1 and x_7, but not x_3. Therefore, stacking the error slacks of m_2 to m_3 will add a new error on the path to x_3, potentially violating the error budget at x_3. Hence, m_2 is not a stacking candidate for m_3.

Next, we consider paths to each output x_j in an iterative loop. The present value of the dynamic budget for output x_j, $\sigma^2_{x_j,dyn}$, is first computed in Line 4. If this value exceeds the variance budget from static optimization, then the slacks are adjusted to ensure adherence to the budget by updating the χ_{m_i} values of the visited node set V in accordance with (A.2).

We then consider the set $\mathcal{M}$ of dynamic nodes that have not been visited so far on Line 9. The nodes in the set $\mathcal{Y}_{m_i}$ are removed from further stacking consideration on Line 10 as they have already been stacked to m_i. These nodes can no longer share their slacks, and we update a narrower set of candidates, $\mathcal{I}_{m_i}$. We then compute the set $\mathcal{Q}_j$ in Line 11, which contains non-dynamic nodes on every path from every element of $\mathcal{M}$ to an output. This set, $\mathcal{Q}_j$, helps to identify the unique terms in $\mathcal{I}_{m_i}$ that are directly assigned to $\mathcal{Y}_{m_i}$ in Line 12. For example, for output x_1, $\mathcal{M}_{1,s} = \{m_3, m_4\}$, and $\mathcal{I}_{m_3} = \{m_1, m_5, m_6\}$, $\mathcal{I}_{m_4} = \{m_2, m_5, m_6\}$, so that $\mathcal{Q}_1 = \{m_5, m_6\}$. The unique elements of $\mathcal{I}_{m_3}$ and $\mathcal{I}_{m_4}$, m_1 and m_2, are directly assigned to $\mathcal{Y}_{m_3}$ and $\mathcal{Y}_{m_4}$, respectively.

On Line 13, we assemble Y, a set of all sets $\mathcal{Y}_{m_i}$, which is updated in the function call to NodeDist on Line 14, where the slacks of the nodes in $\mathcal{Q}_j$ are distributed among $Y[m_i]$ (i.e., $\mathcal{Y}_{m_i}$). After the function returns the updated value of Y, the values are written back into the $\mathcal{Y}_{m_i}$ variables. Finally, Line 16 updates the list χ_{m_i} for nodes $m_i \in \mathcal{M}$, and Line 17 marks the nodes in $\mathcal{M}$ as visited.

The NodeDist routine is described in Algorithm 4. It first computes the slacks of all nodes belongs to the Y and $\mathcal{Q}_j$ sets on Line 2. The total slacks of the stacked nodes in $Y[m_i]$ for all $m_i \in \mathcal{M}$ are calculated in Line 3. The sorted nodes in $\mathcal{Q}_j$ are then greedily stacked to the dynamic nodes in Lines 5–9. The complexity of this algorithm is $O\left(|\mathcal{M}| \cdot |\mathcal{Q}_j|\right)$. Since the cardinality of both $\mathcal{M}$ and $\mathcal{Q}_j$ can be in the order of total number of multiplier nodes, $\mathcal{T}_m$, of a DAG, the complexity becomes: $O\left(\mathcal{T}_m^2\right)$. The complexity of Algorithm 3 is dominated by the operation of NodeDist routine for all outputs, $\mathcal{T}_x$. Hence, the overall complexity of this algorithm is $O\left(\mathcal{T}_x \cdot \mathcal{T}_m^2\right)$.

The results of Algorithm 3 are shown in Table 4.4, which lists the stacked nodes, $m_k \in \mathcal{Y}_{m_i}$, and the corresponding weights, $\chi_{m_i}[m_k]$, for each $m_i \in \Lambda$. The blank entries of $\mathcal{Y}_{m_i}$ and χ_{m_i} in the table imply that no stacking is possible for the corresponding node, and the budget, $\sigma^2_{m_i,NB}$, remains unchanged from its static budget, $\sigma^2_m(\alpha_{m_i})$.

Table 4.4: The set of stacked nodes, $\mathcal{Y}_{m_i}$, and the corresponding weights χ_{m_i} for all selected nodes $m_i \in \Lambda$.

$m_i \in \Lambda$	m_3	m_4	m_7	m_8	m_{10}	m_{11}
$m_k \in \mathcal{Y}_{m_i}$	m_1, m_5	m_2, m_6	m_9	m_9	–	–
$\chi_{m_i}[m_k]$	$1, 2$	$1, 2$	1	1	–	–

4.2.3 Real-Time Optimization of Dynamic Approximation

The optimization problem (4.22) must be solved in real time to obtain the result of dynamic approximation level, z_i. To control the area and power overhead associated with dynamic approximation, it is essential to simplify the framework described above, addressing both hardware and runtime overheads.

Reducing the Number of Iterations during Optimization

The number of dynamic approximate bits, z_i, is a non-negative integer valued number and could be increased in steps of one. The use of a coarser resolution, $\kappa > 1$, can reduce the number of iterations, potentially at the cost of optimality. Practically, $\kappa = 2$ provides a good balance between the two.

Approximating Algorithm and Hardware

We introduce several heuristics to reduce the computation.

Approximation of the Error Variance Formula The computations in the optimization problem involve several additions and multiplications, which would consume significant power if implemented as precise floating point additions and multiplications. To mitigate this, we have approximated the multiplier error variance equation (3.2) as:

$$\hat{\sigma}^2_m(\alpha) = 0.5 \times 4^{\alpha} = 2^{(2\alpha-1)} \tag{4.23}$$

This approximation converts the multiplication operation to a shift, thus saving a notable amount of power. We show in Section VII-B2 that under this approximation, we obtain significant area and power savings at the system level.

Approximation of the Hardware As stated in Section 4.1.3, an absolute comparator is required in the dynamic multiplier block. The calculation of the absolute value of an n-bit number requires n XOR gates that XOR the sign bit with the data bits, and an n-bit adder that adds the sign bit to the result to obtain the two's complement of a negative number. To reduce the hardware cost, we omit the adder block: as a result, the absolute value of a negative number is off by one, which causes no significant error in our estimation.

Formulating the Iterations Incrementally In the optimization problem (4.22), the left hand side of the constraint is given by (4.4) and involves the computation of $\gamma_1(z_i)$ and $\gamma_2(z_i)$, The optimization procedure iteratively increments the values of some z_is, and we devise recursive expressions that update these values between iterations incrementally as:

$$\gamma_1(z_i) = \gamma_1(z_i - \kappa) + \mathcal{F}_1(\kappa, z_i) \tag{4.24}$$

$$\gamma_2(z_i) = 4^{\kappa} \cdot [\gamma_2(z_i - \kappa) - \mathcal{F}_2(\kappa, z_i)] \tag{4.25}$$

$$\mathcal{F}_1(\kappa, z_i) = \sum_{v=\mathcal{N}_i+z_i-\kappa}^{\mathcal{N}_i+z_i-1} P_v \cdot \hat{\sigma}_m^2(v + \lceil \log_2(c_i) \rceil) \tag{4.26}$$

$$\mathcal{F}_2(\kappa, z_i) = \hat{\sigma}_m^2(\alpha_{m_i} + z_i - \kappa) \cdot \sum_{v=\mathcal{N}_i+z_i-\kappa}^{\mathcal{N}_i+z_i-1} P_v \tag{4.27}$$

where the base cases, $\gamma_1(0)$ and $\gamma_2(0)$ are obtained from (4.4). Note that the updated formula uses the simplification in (4.23).

The operations for updating γ_1 and γ_2 are implemented in hardware as a combinational block. From (4.23), the $\hat{\sigma}_m^2$ term is a power of 2, and therefore $\mathcal{F}_1$ and γ_1 can be implemented using adders and shifters, without expensive multiplications. Similarly, $\mathcal{F}_2$ and γ_2 can also be implemented using adders and shifters, and therefore the hardware overhead associated with implementing these updates is low.

The overall dynamic multiplier architecture for multiplier node, m_i, is shown in Fig. 4.4. For an input image, the probability values, P_k, are calculated in the probability calculation block using reduced number of frames. Using these P_k values, the optimization block first calculates the base case values of γ_1 and γ_2 using (4.4) and later computes the new budget, $\sigma^2_{m_i,NB}$ from (4.21). For each z_i, in steps of κ, the error variance is calculated using (4.24)-(4.27) and compared against the new budget, $\sigma^2_{m_i,NB}$, in parallel. The encoder block detects the maximum feasible value of z_i and passes the value to the dynamic multiplier block.

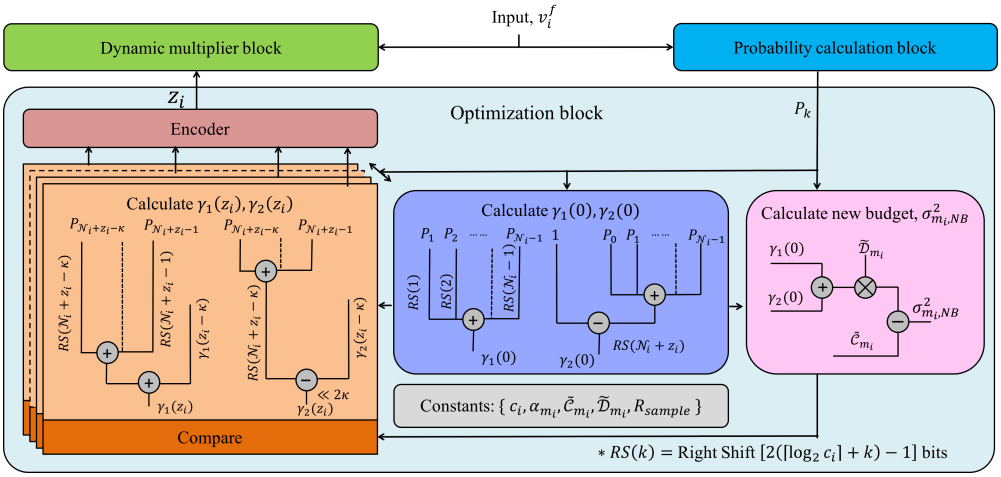


Figure 4.4: Dynamic multiplier architecture for node, m_i.

The percentage power savings for the DCT block under static and dynamic approximation condition incorporating all the heuristics is given by:

$$\frac{\Delta\mathcal{P}_{DCT}}{\mathcal{P}_{DCT}} = \frac{\Delta\mathcal{P}^2_{DCT} + \Delta\mathcal{P}^3_{DCT}}{\sum_{m_i\in\theta_m}\mathcal{P}^1_{m_i} + \sum_{a_i\in\theta_a}\mathcal{P}^1_{a_i}} \tag{4.28}$$

$$\text{where } \Delta\mathcal{P}^2_{DCT} = \sum_{m_i\in\theta_m}\mathcal{P}^2_{m_i} + \sum_{a_i\in\theta_a}\Delta\mathcal{P}_{a_i}$$

$$\Delta\mathcal{P}^3_{DCT} = \sum_{m_i\in\theta''_m}\mathcal{P}^3_{m_i}$$

$$\mathcal{P}^1_{a_i} = \sum_f N_{a_i} \times p_1,\ \Delta\mathcal{P}_{a_i} = \sum_f \alpha_{a_i} \times (p_1 - p_2)$$

where θ_a and θ_m are the sets of all adders and multipliers, respectively, and θ''_m is the set of all dynamic nodes in the 2D DCT block. Using (4.8) and (4.13), the power savings

of static approximation can be reduced to

$$\frac{\Delta\mathcal{P}^2_{DCT}}{\mathcal{P}_{DCT}} = (\frac{p_1 - p_2}{p_1}) \cdot \left[\frac{\sum_{m_i \in \theta_m} \lambda_{m_i,s} + \sum_{a_i \in \theta_a} \alpha_{a_i}}{\sum_{m_i \in \theta_m} \lambda_{m_i} + \sum_{a_i \in \theta_a} N_{a_i}} \right] \tag{4.29}$$

The first term, $(\frac{p_1-p_2}{p_1})$, is the power savings for the approximate FA w.r.t. the accurate FA, and the second term is the number of approximate FAs compared to the total number of FAs in the 2D DCT block. The power savings for static approximation is constant for all images for a specified δ_R. The required area, power, and delay of the dynamic approximation process including the overheads are studied in details in Section 4.3.

4.3 Results

We have implemented our approximate DCT framework using the 45nm NanGate Open Cell Library. The absolute comparator, probability calculation, and optimization block have each been synthesized from an RTL description using Synopsys Design Vision to obtain the power, delay, and area associated with these blocks. The FA blocks (accurate, approximate, DSFA) were designed using Cadence Virtuoso and simulated with HSPICE to obtain their power and delay, and the areas were estimated by drawing the layouts of these FA blocks. All simulations have been performed at the typical process corner with $V_{DD} = 1.1$V and temperature, T = 25°C. Table 4.5 lists the power requirements for the building blocks. The DCT architecture varies the computations bit-widths with the dynamic range of the node. All calculations are performed using variable bit-width signed arithmetic. The JPEG circuit was exercised on the benchmark images from [51, 52].

We first implement static nonlinear optimization problem to obtain optimized approximate DCT hardware (Chapter 3). Next, we implement dynamic approximation to the 48 multipliers identified in Section 4.2.2. Depending on the input image, these multipliers can be approximated to various degrees. We have used the granularity level, $\kappa = 2$ with all proposed heuristics described in Section 4.2.3 for this work. The probability distributions at the multiplier inputs, the switching rates of various blocks as well as the static approximate solution were used in (4.28) to find the total static and dynamic power savings.

Table 4.5: Power dissipation of various blocks in dynamic approximation.

Blocks	Delay (ns)	Power (μW)
Accurate FA	0.16	5.49
Approximate FA	0.00	0.00
DSFA (Accurate)	0.20	6.94
DSFA (Approximate)	0.08	1.13
Absolute (XOR) comparator	1.28	49.09
Probability calculation block	0.10	129.04
Optimization block	6.15	5.6×10^3

Power savings for a set of input images

Table 4.6 shows the power savings comparison between static and dynamic approximation. The first two columns of Table 4.6 show power savings from static approximation for various values of user-specified maximum pixel error, δ_R (defined in (3.7)), at node, R, of Fig. 2.2(a). As expected, the magnitude of savings increases when the error specification is relaxed. It is also worth noting that the magnitude of power savings is image-independent, motivating our work on dynamic approximation.

Table 4.6: Power savings for a set of input images using static and dynamic approximation.

Max. pixel error, δ_R	%Power savings						
	Static	Dynamic					Theoretical limit
		Baboon	Peppers	Lena	Arctichare	Gray21	
30	34.8	32.5	38.0	46.0	46.0	50.3	50.3
40	38.8	37.5	45.7	49.5	50.5	53.5	53.5
50	41.2	41.6	49.0	52.8	52.9	55.8	55.8
60	43.3	45.3	52.7	55.8	55.1	57.2	57.2
70	44.7	47.4	55.4	57.4	56.8	58.2	58.2
80	45.8	49.4	56.7	58.6	58.2	59.3	59.3
90	46.8	51.0	57.8	58.7	58.7	59.8	59.8

For five representative images, Columns 3–7 of Table 4.6 show the overall power

savings, incorporating the cost of the overhead circuitry. The last column shows a theoretical limit on the best achievable power savings, corresponding to the case where all bits of the 48 dynamic multiplier nodes are approximated. The Gray21 image achieves this limit in all cases, and the other images require a certain number of precise bits and deliver power savings that are below the error limit.

The dynamic approximation level variations for an error budget, $\delta_R = 30$, are shown in Fig. 4.5 for image (a) Lena and (b) Baboon. Each plot depicts the level of introduced error at the 64 outputs under static and dynamic approximation. We analyze the results for each image below.

Lena: Moving from static to dynamic approximation increases the error but keeps it within specifications.

Baboon: Static approximation alone brings the error to the constrained limit, and no further gains are achieved through dynamic approximation. In fact, as shown in Table 4.6, the power gains from static approximation are somewhat lost due to the overhead of the dynamic approximation circuitry. At higher error specifications (e.g., $\delta_R = 60$), dynamic approximation provides gains over static approximation.

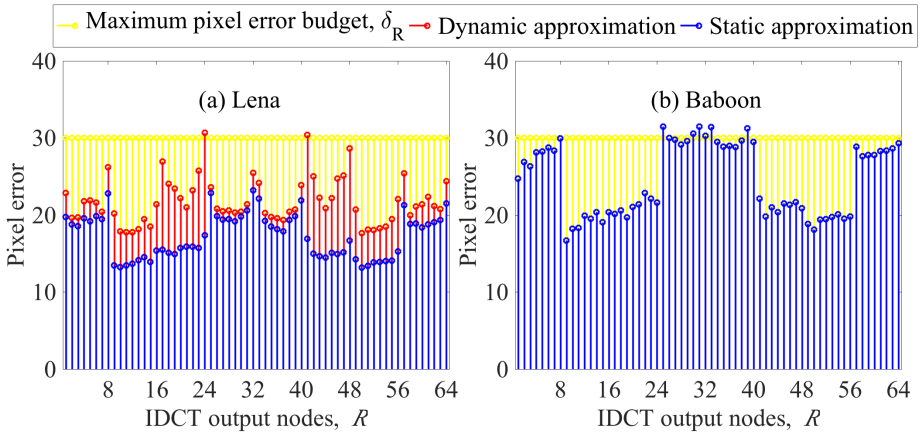


Figure 4.5: The pixel error at each output for (a) Lena and (b) Baboon ($\delta_R = 30, F = 90$).

The dynamic power savings varies with different images which can be attributed to various pixel contrasts of the images. The pixel contrast can be defined as the difference in pixel values among the adjacent image coordinates. As shown in Fig. 2.5, the inputs

to the multipliers of 1D DCT block come from subtractor outputs, and we focus on differences of pairwise pixel inputs to stage, $s(1)$ of 1D DCT for each column of matrix, M, as defined in Section 2.2. We take the cumulative distribution function, CDF, of 10 bins of the pixel difference values, and define the contrast for an image as the index of the bin which contains the 95 percentile data. For example, the image contrast level for Arctichare image is 2 as the second bin contains the 95 percentile data as shown in Fig. 4.6(a). On the other hand, Baboon image has larger contrast level (shown in Fig. 4.6(b)) than Arctichare as more bins are required to reach 95 percentile data.

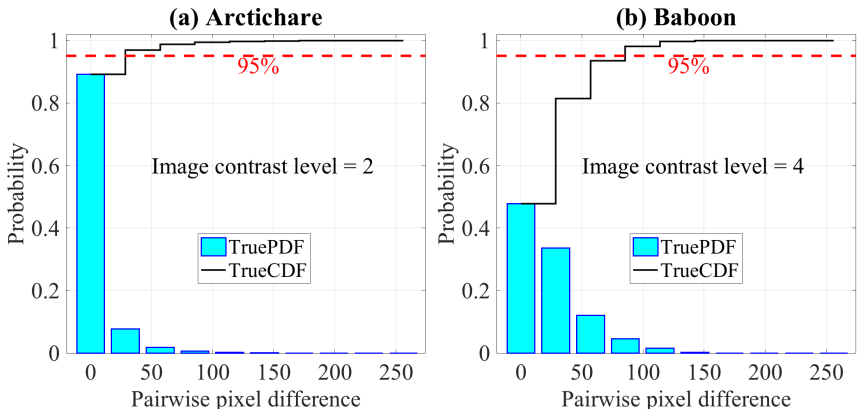


Figure 4.6: Distribution of pairwise pixel differences and image contrast level calculation of (a) Arctichare and (b) Baboon.

The heatmaps in Fig. 4.7 show average of percentage dynamic power savings, $\Delta\mathcal{P}^3_{DCT}/\mathcal{P}_{DCT}$, and average of image quality degradation from the base case, ΔPSNR, for 30 standard images [51, 52] with different contrast levels and error budgets, δ_R. The base case corresponds to a quality factor, $F = 90$ with no approximation, and the $PSNR$ values are computed using Eq. (2.5). For $\delta_R = 0$, the dynamic approximation process was implemented without any static approximation. Hence, the power savings are negative due to the overhead of additional circuitry. For the same image contrast level, as the level of static approximation increases, the overhead becomes progressively easier to overcome which improves the dynamic power savings but degrades PSNR.

On the other hand, increased image contrast level translates to higher probabilities of higher input ranges to multipliers and introduces larger error (Eq. (4.4)). As a result, for the same error budget, increased image contrast degrades both dynamic power savings

and PSNR. We can create a lookup table using this heatmap. For a known maximum image contrast level of a particular application, the required minimum error budget for dynamic approximation to outperform static approximation can be found. If the average PSNR degradation for the minimum error budget is acceptable, then the JPEG hardware can be designed with dynamic approximation capabilities.

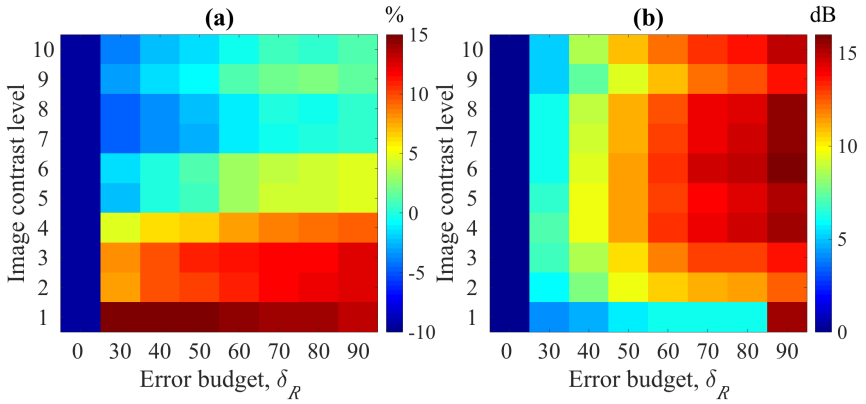


Figure 4.7: Heatmap of average (a) percentage dynamic power savings, $\Delta\mathcal{P}^3_{DCT}/\mathcal{P}_{DCT}$, and (b) $PSNR$ degradation for various image contrast levels and error budgets, δ_R.

Impact of hardware choices

We evaluate the effectiveness of our heuristics in Section 4.2.3 that simplify the hardware implementations of the absolute comparator, the error variance formula in (4.23), and the choice of the granularity level, κ, of the number of dynamic approximate bits, z_i in Table 4.7. For several choices of κ and under various heuristics, we compare the power and area savings and the $PSNR$. We evaluate these on the Baboon image, which shows the lowest power savings, i.e., the highest overhead (Table 4.6). Table 4.7 shows, for the same value of κ, the introduction of heuristics show significant area savings and some power savings, with negligible change in the $PSNR$. The use of coarser granularity of z_i, shown in the third row, cuts into the power savings and $PSNR$ slightly, but uses noticeably less area due to the reduction in the optimization circuitry.

Table 4.7: Comparison of power and area savings, and PSNR for various hardware choices (Baboon and $\delta_R = 70$).

Hardware choice		% Power	% Area	$PSNR$ (dB)
Heuristics	κ	savings	overhead	
No	1	46.69%	27.96%	22.01
Yes	1	47.81%	5.42%	22.02
Yes	2	47.41%	1.73%	22.12

Impact of approximation choices

To quantify the gains of specific approaches, we focus on the image Gray21, which is seen to achieve the maximum theoretical savings. We compare the following three approaches to achieve iso-quality degradation ($\delta_R = 30$) for a specific image, Gray21:

1. No approximation with a small quality factor, $F = 50$
2. Static approximation with a large quality factor, $F = 90$
3. Dynamic approximation with $F = 90$

We assume the base case to be the no-approximation JPEG computation with the quality factor, $F = 90$. For a maximum pixel error, $\delta_R = 30$, the compression ratio, power, delay and area comparison for various approaches, with respect to the base case, are listed in Table 4.8. The compression ratio is inversely related to F, and is calculated from the image storage sizes of the original image and JPEG compressed image. The base case uses a $9:1$ compression ratio.

Approach 1) uses a small quality factor, $F = 50$ to achieve the target δ_R. The storage requirement is reduced by $(15/9)\times$ compared to the baseline. However, this approach does not yield any area, power, and delay savings as there is no approximation. On the other hand, Approach 2) employs static approximation with $F = 90$ and reduces the power and area by 34.8% and delay by 46.1% over the base case.

Approach 3) uses $F = 90$, as in Approach 2), but implements dynamic approximation. A simple implementation of this scheme requires a large amount of overhead circuitry, and therefore we explore tradeoffs that will reduce this overhead. We consider the structure of the dynamic multiplier node, as shown in Fig. 4.4. The optimization block has the largest area overhead, and we consider two choices:

Table 4.8: Comparison of memory requirement, power, delay, area savings and normalized power-delay product w.r. to dynamic approximation (Gray and $\delta_R = 30$).

	Reduced Quality	Static approximation	Dynamic approximation	
			(a)	(b)
Quality factor, F	50	90	90	90
Relative compression ratio	15:9	1:1	1:1	1:1
% power savings	0.0%	34.8%	50.1%	50.3%
% delay savings	0.0%	46.1%	41.1%	41.0%
% area savings	0.0%	34.8%	–91.3%	–12.2%
Normalized power-delay product	3.4×	1.2×	1.0×	1.0×

(a) Each dynamic multiplier node is provided with its own optimization block.

(b) The dynamic aproximation was implemented in each of the eight 1D DCT blocks in Layer 2 of the 2D DCT block of Fig. 2.3. Since the multiplier nodes of these eight blocks use the same constant multiplier values and the same static approximation levels, we use a single optimization block for all 1D DCT blocks.

The results for both cases are shown in the last two columns of Table 4.8. Case (a) incurs a large area overhead of over 91% over the base case, but Case (b) reduces this area requirement significantly to 12%. The optimization block computes the dynamic approximation level, z_i, for all the dynamic multipliers. For case (a), all calculations are performed in parallel and requires only one additional clock cycle, but for case (b), the shared optimization block finds the optimal z_i values for each of the eight blocks serially and thus requires eight additional clock cycles. This incurs a small delay penalty, but this additional overhead is negligible in comparison with the delay and power of the DCT block computations. The power gain for this approach is 50.3% over the baseline, which translates to $\left(\frac{65.2-49.7}{65.2}\right) \times 100\% = 23.8\%$ lower power requirement than the static approximation alone. However, dynamic approximation process requires additional overhead circuitry and hence the delay requirement for this approach is increased by $\left(\frac{59.0-53.9}{53.9}\right) \times 100\% = 9.5\%$ than the static approximation. As the power improvement is more dominating than the delay increment, the static approximation process

requires $\left(\frac{65.2\times53.9}{49.7\times59.0}\right) = 1.2\times$ larger power-delay product compared to the dynamic approximation process. Whereas, the power-delay product for approach 1) is significantly larger than both static and dynamic approximation process.

4.4 Conclusion

We have proposed an optimized JPEG compression unit framework for a user-specified error budget, consisting of both static design-time optimization and dynamic run-time approximation. Depending on the input image characteristics, the approximation level changes to meet the user specified error budget, translating into overall power savings.

Chapter 5

SeFAct: Selective Feature Activation and Early Classification for CNNs

As DNNs have grown deeper, adding more layers and features [14], their computational requirements have rapidly increased. As a result, platforms that implement DNNs in hardware are power-hungry and require large memory footprints. Therefore, energy reduction methods for DNN hardware are of paramount importance.

Prior works have proposed several methods for reducing computations [56], including data parallelization using multiple cores, specialized hardware [26–28], and approximate computation [30–33, 57] based on simplified architecture and arithmetic. In this work, we identify specific DNN properties related to how features are mapped to neurons in various layers that can provide further improvements in computational efficiency. We make two observations regarding the feature activation patterns in CNNs. First, during both training and testing, specific features tend to be activated by the recognition of specific classes. Second, deeper layers have higher neuron activation sparsity, i.e., fewer neurons are activated per layer [58]. Some paths through the network are unlikely to be activated because specific sets of neurons are seldom activated together. These properties can be leveraged to switch off unnecessary computations to save energy, and we propose a selective feature activation approach, SeFAct, to develop quantitative

metrics for identifying such scenarios, which are used to drive runtime energy-reduction optimizations. Specifically, during training, we prepare for selective feature activation by grouping similarly activated classes into clusters across multiple classes. During testing, we go beyond well-known static pruning methods [30,59,60] to perform dynamic pruning: we use the clusters to dynamically approximate a large section of the CNN that does not help interpret the input. This optimization is particularly beneficial in the later layers of a DNN. The energy requirement of a neural network comes from two parts, data access from various hierarchies of memory, and computation of neuron activations. The memory access is the dominant part of the energy consumption in a neural network. Our proposed method achieves large energy savings by reducing both the number of memory accesses and the computations. We couple these gains with reduced precision in earlier layers of the DNN. Reduced precision refers to reducing the number of bitwidths for data representation. Together, these methods provide excellent energy savings without significantly affecting accuracy.

5.1 Overview of the Selective Feature Activation

5.1.1 Motivating Example

We illustrate our key idea on a simplified neural network for letter recognition with three fully connected layers, L_1, L_2, and L_3, shown in Fig. 5.1. We later implement this idea on larger, standard CNN topologies. Each neuron represents one feature, and the outputs map to six classes, corresponding to the letters A through F. Two feature activation patterns are shown in Fig. 5.1, where the activated neurons for Classes B and D are highlighted and the unactivated neurons are white.

It can be seen that classes B and D have 75% or more common activated features in both layer L_1 and L_2. As the activation patterns of the classes B and D are very similar, we can cluster them together in training phase. More generally, we identify the common feature activation pattern for all classes in each cluster, which we refer to as its "stamp". During the testing phase, if the feature activation pattern of an input matches with a particular stamp, then the input is said to belong to one of the classes from the corresponding cluster. This clustering helps predict classes in early layers, and can be used to reduce both the neuron computation and memory access energy in the

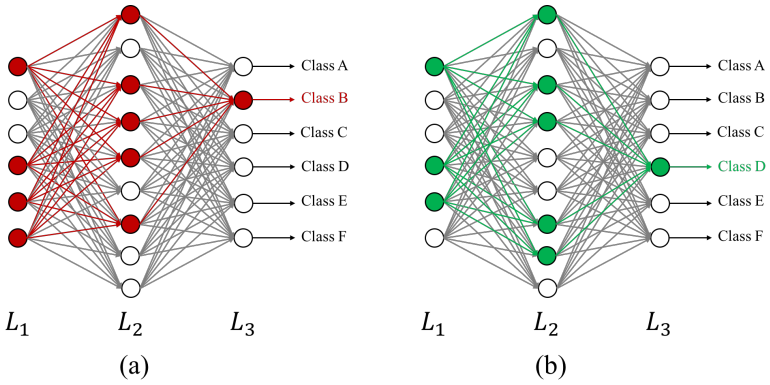


Figure 5.1: Activation pattern of features for (a) Class B and (b) Class D in layers, L_1 to L_3 of an example neural net.

subsequent layers. By grouping class B and D in one cluster in layer L_2, we can narrow down the classification options from six to two, and skip computations and data loads for other classes in layer L_3.

We build a framework for identifying and using *important features* to selectively activate neurons. We work in two steps:

1. *Cluster learning phase* We augment the traditional training phase with a step that identifies clusters. Based on the trained weights and training data, we

   (a) identify the important features for each class

   (b) cluster classes with similar feature activation patterns

   (c) prepare stamps and cluster data for the testing phase

2. *Testing phase* The traditional testing phase is modified to use the cluster learning results. At each CNN level, we

   (a) compare feature activation patterns with the cluster stamps and identify the activated cluster(s)

   (b) load and compute data only for the predicted class(es)

   (c) propagate the predicted class sets to the next layer

Thus, cluster learning is a preprocessing step associated with training. Changes to the testing phase incur overheads, but also generate savings, and are implemented in

real-time.

Notation 1. *We frequently refer to a variable x in multiple layers. We denote the variable x in the current layer, L_i, without a superscript; the variable in the previous layer, L_{i-1}, and the next layer, L_{i+1}, as x^- and x^+, respectively.*

5.1.2 Cluster Learning Phase

The Concept of Important Features

The input feature map, i.e., *ifmap* (F^{if}), shown in Fig. 2.7, is used to detect important features. In this section, we explain how we identify the important features for a single input image, using the notation for the DNN defined in Section 2.3.

For the ifmap, the summation of all data in the k^{th} feature plane, $\tilde{F}_k^{if} = \sum_{h=1}^{H^-} \sum_{w=1}^{W^-} F_{khw}^{if}$, is a "signature" for the feature. We calibrate the importance of a feature in a layer based on the relative magnitude of $\tilde{F}_k^{if}$ with respect to the average of all feature plane data, $\tilde{F}_{avg}^{if} = \sum_{k=1}^{K^-} \tilde{F}_k^{if} / K^-$.

Definition 1. *The indicator function $1\{\cdot\}$ is defined as $1\{true\} = 1$, and $1\{false\} = 0$.*

Definition 2. *Feature k is considered important if it satisfies*

$$I_k = 1\left\{\tilde{F}_k^{if} \geq T_1 \times \tilde{F}_{avg}^{if}\right\} = 1\left\{\tilde{F}_k^{if} \geq T_I\right\} \tag{5.1}$$

where T_1 is a tunable threshold parameter.

Computational simplification: In simple terms, Eq. (5.1) states that an ifmap feature is important if it is within a multiplicative factor T_1 of the average of all feature plane data. While this equation can be applied relatively easily to an *FC* layer where $H^- = W^- = 1$ and the volume of data is moderate, *Conv* layers must handle a much larger volume of data. In a realistic hardware implementation, the 3D data processed by a *Conv* layer of the CNN is fetched through a memory hierarchy. It is computationally expensive to fetch the data and compute $\tilde{F}_{avg}^{if}$ for use in Eq. (5.1). Moreover, ifmap data is very sparse, i.e., numerous elements are zero.

We simplify the computation of $\tilde{F}_{avg}^{if}$ based on mean and sparsity information of the *ifmap*, pre-calculated during the cluster learning phase. The simplification helps to reduce computation without hurting the effectiveness of our method.

Theorem 1. *Under the above simplified assumption, feature k in a Conv layer is important, as defined in Definition 2, if*

$$I_k = 1\left\{(1 - S_k^{if})\mu_k^{if} \geq T_I\right\}, \tag{5.2}$$

where $T_I = T_1 \times (1 - S^{if}) \times \mu^{if}$. *We assume, the overall sparsity,* $S^{if} = \sum_{k=1}^{K^-} S_k^{if}/K^-$, *and the average of all nonzero data,* μ^{if}, *are constants for an* ifmap *layer.*

Proof. The number of nonzero elements associated with the k^{th} feature plane is $H^-W^-(1-S_k^{if})$. If the average of all nonzero data in the feature plane is μ_k^{if}, we can write, $\tilde{F}_k^{if} = H^-W^-(1 - S_k^{if})\mu_k^{if}$. The Eq. (5.1) for a *Conv* layer then becomes:

$$\begin{aligned} I_k &= 1\left\{H^-W^-(1 - S_k^{if})\mu_k^{if} \geq T_1 \times \frac{\sum_{k=1}^{K^-} H^-W^-(1 - S_k^{if})\mu_k^{if}}{K^-}\right\} \\ I_k &\approx 1\left\{(1 - S_k^{if})\mu_k^{if} \geq T_1 \times (1 - S^{if}) \times \mu^{if}\right\} = 1\left\{(1 - S_k^{if})\mu_k^{if} \geq T_I\right\} \end{aligned} \tag{5.3}$$

The latter approximation operates under the assumption that the average of all feature plane data, $\tilde{F}_{avg}^{if} = (1 - S^{if}) \times \mu^{if} H^-W^-$, is *constant*, as H^-, W^-, S^{if} and μ^{if} are considered constants for a given *ifmap* layer. □

Important Feature Detection for Each Class

The relation in (5.1) shows how the k^{th} element of the class-based importance feature vector, $I \in \mathbb{R}^{K^-}$, is used to develop the importance criterion for the features of an *individual* input image in layer, L_i. However, during the cluster training phase, we work with *multiple* images in multiple classes to identify important features for the cluster.

In the current layer, let us say that we apply (5.1) to determine the importance feature vector I' of the q^{th} image of one particular class c, i.e., $I'(c, q) = I$. During cluster learning, we repeat this operation over all N_{class} classes of the CNN, based on the learned data from $N_{image}(c)$ images for each class c. Due to the network training inaccuracies, or image pixel pattern variations, all the images with same class may not have the same activation pattern for a specific layer. Therefore, we deem a feature to be important for a particular class if it is important for a sufficiently large number of images in the class, i.e., if it is activated for a fraction T_2 of all training images. This provides a

criterion for determining the important features through the vector $\tilde{I}_c \in \mathbb{R}^{K^-}$, for class c of the current layer:

$$\tilde{I}_c = 1\{\sum_{q=1}^{N_{image}(c)} I'(c,q) \geq T_2 \times N_{image}(c)\} \tag{5.4}$$

Clustering

Classes with closely matched features are now grouped together as clusters that satisfy few properties. We first list the properties below and then illustrate the idea of clustering through an example.

Property: A cluster is a set of classes, with one or more clusters associated with each layer, satisfying these properties:

1. Clusters in the last layer are singleton sets, with each set containing one class.
2. No cluster is a subset of another cluster in the same layer.
3. If $\mathcal{C}$ and $\mathcal{C}^+$ are the sets of clusters in layer L_i and layer L_{i+1}, then each cluster $\mathcal{C}_x$ in layer L_i must satisfy

   $$\mathcal{C}_y^+ \subseteq \mathcal{C}_x \quad \text{for some } y.$$

   We refer to this as the **diverging clustering criterion**. We define a cluster graph in which each cluster forms a vertex, and an edge is drawn from x to y if the above criterion is satisfied. Moreover,

   $$\mathcal{C}_x = \bigcup_{x \to y} \mathcal{C}_y^+.$$

Fig. 5.2 shows a sample clustering of the network of Fig. 5.1. Clusters in the last layer are singletons, consistent with Property 1 above, and no cluster within the same layer is a subset of another, as stated in Property 2. Property 3 can be illustrated by cluster $\mathcal{C}_3^{L_1} = \{B, D, E, F\}$ of layer L_1, which diverges through its connections to clusters $\mathcal{C}_2^{L_2} = \{B, D\}$ and $\mathcal{C}_4^{L_2} = \{B, E, F\}$.

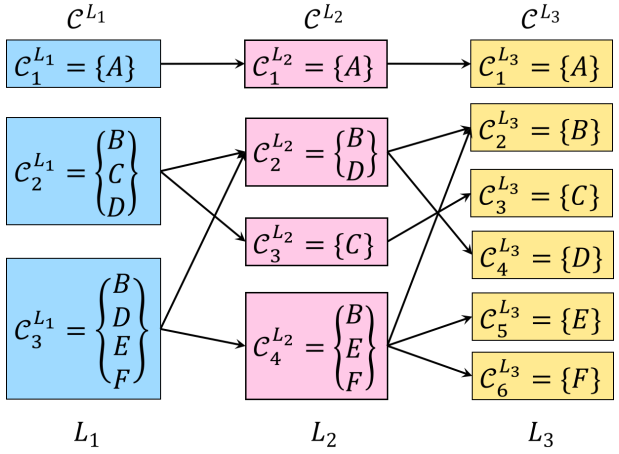


Figure 5.2: Example to illustrate the properties of clusters.

The diverging clustering criterion allows the network to reduce the number of predicted classes monotonically. For example, if an image of the letter, E, is provided to the network, then the probable classes for layer L_1, L_2, and L_3 are $\{B, D, E, F\}$, $\{B, E, F\}$ and $\{E\}$, respectively.

The class-based importance vector, $\tilde{I}_c$, obtained in (5.4), lists the set of important features in a layer. We use this information to create diverging clusters from layer to layer. We begin by creating singleton clusters in the last layer; the number of clusters equals the number of classes. Then, we topologically move backwards maintaining the diverging clustering criterion.

For each cluster $\mathcal{C}_k^+$ of layer L_{i+1}, we prepare a cluster-based important feature vector for layer L_i, J_k, that lists the features important to all cluster members, using the AND ($\bigwedge$) operator:

$$J_k = \bigwedge_{c \in \mathcal{C}_k^+} \tilde{I}_c \tag{5.5}$$

As an example, consider the cluster $\mathcal{C}_4^{L_2} = \{B, E, F\}$ in layer L_2 in Fig. 5.2. The class-based importance matrices of classes B, E and F for layer L_1 ($\tilde{I}_B^{L_1}$, $\tilde{I}_E^{L_1}$ and $\tilde{I}_F^{L_1}$, respectively) are shown in Fig. 5.3, which indicate features that are important to each member of the class. The binary AND operation provides the set of features that are important to all classes in the cluster.

We then combine clusters in layer L_{i+1} into larger clusters in layer L_i. We create

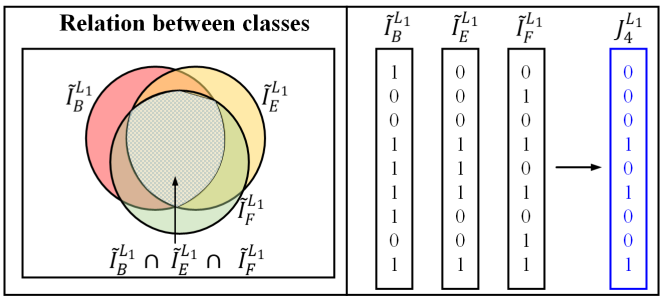


Figure 5.3: Example of preparing cluster-base important vector of layer L_1, $J_4^{L_1}$, for the cluster $\mathcal{C}_4^{L_2}$ of layer L_2.

a graph $G(V, E)$ whose vertex set $V = \mathcal{C}^+$, and with edge set E connecting clusters that have high affinity, but keeping clusters with low affinity unconnected. The affinity metric for clusters $\mathcal{C}_i^+$ and $\mathcal{C}_j^+$ is given by $A_{ij} = \langle J_i, J_j \rangle$, the inner product between J_i and J_j. Since these vectors are Boolean, the inner product measures the number of important features that these clusters have in common. We add an unweighted edge to E if the affinity exceeds a threshold, $T_3 \times A_{max}$, where A_{max} is the maximum of all cluster affinities:

$$(i, j) \in E \text{ iff. } \{A_{ij} > T_3 \times A_{max}\} \quad 1 \leq i, j \leq N_{\mathcal{C}^+} \tag{5.6}$$

The goal of clustering is to group the high-affinity clusters in layer L_{i+1} together. We map this problem to the *maximum independent set* (MIS) problem on graphs [61]. Two vertices are independent if they have no edge between them, i.e., they have low affinity. The MIS problem finds a maximal set of independent vertices, which act as roots of individual clusters. By definition, elements of the MIS have low affinity for each other and should be in different clusters. Each cluster is thus defined by an MIS element, and also includes its high affinity neighbors, i.e., classes.

Algorithm 5 finds the clusters in layer L_i by first identifying the vertices in G (i.e., cluster IDs) to be merged based on finding the MIS. Since the MIS problem is NP-hard, we employ a greedy heuristic [62]. After initialization (line 1), the algorithm sorts the vertices in non-ascending order of their degrees (line 2). In each iteration, the minimum degree node in V is added to the MIS (line 5). A new cluster is formed, combining the clusters associated with v and its neighbors with at least equal degree of v (line 7). The neighbors with lower degree than v are already included in previously formed clusters

that should not be combined with the current cluster, since other elements of the other cluster may not have much affinity with the current cluster. Hence, this additional check helps creating more disjoint clusters as well as increasing affinity between classes of same cluster. The iteration ends with v and its neighbors eliminated from V (line 8). Once all clusters have been found, the cluster IDs in $\mathcal{D}^+$ are used to construct the set of classes in the cluster set $\mathcal{C}$ (line 11). The complexity of the algorithm is $\mathcal{O}\left(N_{\mathcal{C}^+} \log\left(N_{\mathcal{C}^+}\right)\right)$, which is dominated by the sorting operation of the vertices (line 2).

Algorithm 5 $\{\mathcal{D}^+, \mathcal{C}\} \leftarrow \text{FindCluster}(G, \mathcal{C}^+)$

INPUT: Edge connectivity graph $G(V, E)$ based on $\mathcal{C}^+$
Cluster set of layer L_{i+1}: $\mathcal{C}^+$
OUTPUT: Diverging cluster set: $\mathcal{D}^+$ ▷ IDs of clusters to be merged
Cluster set of layer L_i: $\mathcal{C}$ ▷ Set of classes in the cluster
METHOD:

1: Initialize $I_s = \varnothing$, $\mathcal{D}^+ = \varnothing$
2: Sort the vertices in V in non-ascending order of degree
3: **while** $V \neq \varnothing$ **do**
4: Set v to be the minimum-degree vertex in V
5: $I_s = I_s \cup v$ ▷ Add v to the independent set
6: $n_v \leftarrow \{u\} \forall (u, v) \in E$ and $\text{degree}(u) \geq \text{degree}(v)$ ▷ Set of neighbors of v
7: $\mathcal{D}^+ \leftarrow \mathcal{D}^+ \cup \{\{v\} \cup n_v\}$ ▷ Add v and its neighbors to $\mathcal{D}^+$
8: $V \leftarrow V \setminus \{\{v\} \cup n_v\}$
9: **end while**
10: $\forall \mathcal{D}_k^+ \in \mathcal{D}^+$, $\mathcal{C}_k \leftarrow \cup_{v \in \mathcal{D}_k^+} \mathcal{C}_v^+$ ▷ Build clusters using cluster IDs
11: $\mathcal{C} \leftarrow \bigcup \mathcal{C}_k$ ▷ Set of all clusters at layer L_i
12: **return** $\{\mathcal{D}^+, \mathcal{C}\}$

Data Preparation for Testing Phase

We now encapsulate clustering information to enable its efficient use during the testing phase. At each layer L_i, the set $\mathcal{D}^+$ shows how the clusters in the current layer diverge to those in the next layer.

We prepare **stamps** for each cluster in layer L_i, corresponding to the features that can activate the cluster, i.e., the features that are important to all classes in the cluster. These stamps are used in the testing phase to determine the activated clusters, by comparing the list of important features in the input data with each cluster stamp. The

stamp $\mathcal{S}_k$ for cluster k is:

$$\mathcal{S}_k = \bigwedge_{c \in \mathcal{C}_k} \tilde{I}_c \tag{5.7}$$

For each cluster $\mathcal{C}_k$ in layer L_i, we now create a record of the features to be computed, $\mathcal{F}_k^+$, to identify potentially activated clusters in layer L_{i+1}. During the testing phase, for each activated cluster $\mathcal{C}_k$, only these features are inspected. For cluster $\mathcal{C}_k$, we compute $\mathcal{F}_k^+$ by combining, through a binary OR ($\bigvee$), the important features of layer L_{i+1} as:

$$\mathcal{F}_k^+ = \bigvee_{c \in \mathcal{C}_k} \tilde{I}_c^+ \tag{5.8}$$

For example, in Fig. 5.2, the stamps for all clusters in layer L_1 are used to check which classes are activated. In the testing phase, for an input image B, depending on which features are activated, the stamps for $\mathcal{C}_3^{L_1}$ could trigger the identification of this cluster.[1] Next, from $\mathcal{D}_3^+$, we know that the activated clusters in layer L_2 may be $\mathcal{C}_2^{L_2}$ and $\mathcal{C}_4^{L_2}$. Accordingly, we use the list of important features given by Eq. (5.8) to compute the important features for classes in clusters $\mathcal{C}_2^{L_2}$ and $\mathcal{C}_3^{L_2}$. If any other cluster is activated, then a similar approach is used to add to the list of important features to be computed.

Overall Algorithm

Algorithm 6 summarizes the cluster learning phase for layer L_i. Lines 1 and 2 prepare, respectively, the class-based and cluster-based importance feature vectors at level L_i. Based on the clusters and their affinities, the cluster graph G is formed. Next, Algorithm 5 is invoked to form the diverging clusters. Finally, in preparation for the testing phase, for each cluster, a cluster stamp and a record of important features for the next level are computed.

[1] Note that not all important features of a class are activated by each image: therefore, an image in class B may well activate only $\mathcal{C}_3^{L_1}$ and not $\mathcal{C}_2^{L_1}$.

Algorithm 6 The Cluster Learning Phase

INPUT: Layer L_i: ifmap data F^{if}; thresholds T_1, T_2, T_3
Layer L_{i+1}: importance vectors $\tilde{I}_c^+ \forall$ classes c; cluster set $\mathcal{C}^+$
OUTPUT: Layer L_i: importance vectors $\tilde{I}_c \forall$ classes c; cluster set $\mathcal{C}$; cluster stamps $\mathcal{S} \in \mathbb{R}^{N_\mathcal{C} \times K^-}$,
Layer L_{i+1}: divergent cluster set $\mathcal{D}^+$; features $\mathcal{F}^+ \in \mathbb{R}^{N_\mathcal{C} \times K}$
METHOD:

1: Create importance vectors for layer L_i, $\tilde{I}_c$ ▷ Use T_1, T_2, and (5.4)
2: Form cluster importance vectors in layer L_i, J ▷ Use $\mathcal{C}^+$ and (5.5)
3: Create $G(V, E)$ ▷ Use (5.6) and T_3
4: $\{\mathcal{D}^+, \mathcal{C}\} \leftarrow \text{FindCluster}(G, \mathcal{C}^+)$ ▷ Algorithm 5
5: **for** $k = 1 : N_\mathcal{C}$ **do**
6: Prepare stamp, $\mathcal{S}_k$; important features, $\mathcal{F}_k^+$ ▷ Use (5.7), (5.8) and $\tilde{I}_c^+$
7: **end for**
8: **return**

5.1.3 Testing Phase

The cluster testing phase for layer L_i is described in Algorithm 7. Based on the cluster activations, the class predictions are updated and the information is used to reduce computation for future layers. First, we count the number of stamp elements that are not matched by the input data (line 1). By performing these computations only on $\tilde{D}$ clusters, we significantly reduce energy in comparison to the basic (non-SeFAct) implementation. Next, we find the minimum mismatch to detect the activated clusters that are within some margin of this mismatch (lines 2 and 3). The activated clusters contain the updated predicted classes. The L1 norm used in these computations is the sum of absolute differences of each element of the vector.

Next, for the updated predicted classes, we identify the important features of ofmap, $\tilde{\mathcal{F}}$. The reduced ofmap data is determined in line 5. Finally, lines 6 and 7 determine, the important features and activated clusters in layer L_{i+1}, respectively. Note that these computations are identical to the cluster learning phase, except here we work with only one test image at a time.

Algorithm 7 The Cluster Testing Phase

INPUT: Layer L_i: ifmap F^{if}; filter F^{filter}; bias F^{bias}; importance features, I ; cluster set $\mathcal{C}$; cluster stamps $\mathcal{S}_k \forall$ cluster k; activated clusters $\tilde{\mathcal{D}}$; features $\mathcal{F}$; threshold T_4
Layer L_{i+1}: divergent cluster set $\mathcal{D}^+$; threshold T_1^+
OUTPUT: Layer L_i: ofmap data: F^{of}
Layer L_{i+1}: activated clusters $\tilde{\mathcal{D}}^+$; important features: I^+
METHOD:

1: Find cluster mismatch, $\mathcal{M}_k = \sum_{k \in \tilde{\mathcal{D}}} \|\mathcal{S}_k \wedge \neg I\|_1$
2: $m = \min(\mathcal{M})$ ▷ Minimum cluster mismatch
3: $\mathcal{A}_k = 1\{\mathcal{M}_k \leq m + T_4 \|\mathcal{S}_k\|_1\}, k \in \tilde{\mathcal{D}}$ ▷ Cluster activation
4: Obtain features to compute for ofmap, $\tilde{\mathcal{F}} = \vee_{\mathcal{A}_k=1} \mathcal{F}_k$
5: Compute ofmap using (2.6) and $\tilde{\mathcal{F}}$
6: Detect important features for layer L_{i+1}, I^+ ▷ Use (5.1) and T_1^+
7: Obtain clusters to check for layer, L_{i+1}, $\tilde{\mathcal{D}}^+ = \bigcup_{\mathcal{A}_k=1} \mathcal{D}_k^+$
8: **return**

5.1.4 SeFAct Implemention in Various Layers

The *cluster learning* and *testing* phase of SeFAct implementation in a layer, as described in Section 5.1.1, is a complex process which requires the two steps listed below:

- *Step* 1 Cluster preparation, class prediction updating and important feature identification
- *Step* 2 Computation reduction by detecting unimportant features based on the updated class prediction

The implementation of cluster preparation and prediction (*Step* 1) incurs energy overheads over the basic (non-SeFAct) implementation, whereas *Step* 2 is expected to substantially reduce energy, paying for the overhead of *Step* 1, by skipping the computation of unimportant features. Algorithms 6 and 7 are used together to implement these two major steps of SeFAct implementation. We implement SeFAct on various types of CNN layers (described in Sections 2.3 and 2.3.2) as follows:

1. *Conv and FC Layers* Both steps of SeFAct are implemented in the *Conv* and *FC* layers, i.e., both layers prepare clusters during the cluster learning phase and update the class prediction to reduce computation in the cluster testing phase.

2. *Pool Layer* The *Pool* layer reduces the spatial dimension of *Conv* layers and operates on each feature map independently, as mentioned in Section 2.3.1. Therefore, a *Pool* layer is mere representation of its immediate predecessor *Conv* layer and important features of both layers are the same. Hence, the *Pool* layer does not incur any computational overhead, as it receives SeFAct information from previous *Conv* layer: cluster prediction and importance of various feature planes. We skip the *pooling* operation for the unimportant feature planes.

3. *Norm Layer* The *Norm* layer is used to normalize the data of a *Conv* layer to zero mean and unit standard deviation. The important feature patterns for both the *Norm* layer as well as its predecessor *Conv* layer are similar since normalization produces just a scaled and shifted version of the data. Therefore, we adopt a similar strategy for *Norm* layer as for the *Pool* layer.

4. *NIN Module* The *network-in-network* (NIN) concept incorporates additional nonlinearity by introducing intermediate nonlinear layers as described in Section 2.3.2. There can be multiple intermediate layers in a NIN module. For example, there are three nonlinear intermediate layers in an *inception* module of GoogLeNet (Fig. 2.8). To limit the energy overhead, we do not implement *Step* 1 in the intermediate layers of an NIN modules, and only propagate the cluster prediction from the previous layer to reduce computation (*Step* 2). However, there is a difference between *Pool/Norm* layers and intermediate layers of NIN module with regard to important feature detection. The *Pool* and *Norm* layers are mere modifiers of their previous *Conv* layer and they have the same important feature pattern. On the other hand, the intermediate layers of an NIN module are generally implemented as *Conv* layers. Therefore, the important feature patterns of intermediate layers may not be the same as those in their predecessor layer. We accommodate this difference in *Step* 2 by implementing certain additional measures in Algorithm 6 (cluster learning algorithm) as well as Algorithm 7 (cluster testing algorithm). Specifically:

    - We compute class-based important features ($\tilde{I}_c$) as well as predictive data, cluster-activation-specific important features ($\mathcal{F}_k^+$) for a specific layer in Algorithm 6 (lines 1 and 6). We also determine these two metrics, $\tilde{I}_c$ and $\mathcal{F}_k^+$,

for the additional intermediate layers.

- During the testing phase (Algorithm 7), important features for the predictive classes ($\tilde{\mathcal{F}}$) are computed to save *ofmap* computations (lines 4 and 5). We additionally compute $\tilde{\mathcal{F}}$ for the intermediate layers to identify their important features and reduce computations.

5.2 Design Optimizations for Energy Reduction

We enable energy-efficient neural computation by combining selective feature activation, SeFAct, described in Section 5.1, with optimized reduced-precision approximation. Reduced precision schemes have been explored in recent research [26,31] as well as commercial platforms [13, 27]. Some approaches have used fixed bitwidths (for example, 8-bit [13]) for all the layers. Other approaches [63] have used layerwise bitwidth optimization for controlled error introduction and improved power savings. The reduced precision approximation and our SeFAct approach are two orthogonal processes that introduce controlled levels of error in network to achieve energy savings. We obtain optimized bitwidths for various layers with Monte-Carlo simulations.

5.2.1 Choice of Layers for Selective Activation Implementation

The SeFAct scheme is useful in network layers where a relatively few features are activated for each class. However, in early layers, individual neurons do not have enough information from the input to narrow down the set of possible classes, and many neurons may be activated, regardless of the class.

An alternative way to explain the usefulness of SeFAct implementation in later layers is through the concept of the *receptive field* [41]. The receptive field of a neuron is the region in the input image that affects the neuron. The dimensions of the square receptive field for different layers of LeNet, AlexNet, and GoogLeNet are given in Tables 5.1, 5.2, and 5.3, respectively. The size of input images for LeNet (AlexNet/GoogLeNet) is 28×28 (227×227). According to Fig. 2.8, there are multiple parallel paths for GoogLeNet with three different filters sizes (1×1, 3×3, 5×5). We use the median filter size, 3×3, to compute the receptive field.

Table 5.1: Receptive fields of various layers in LeNet.

Layer	$c1$	$p1$	$c2$	$p2$	$fc1$	$fc2$
Dimension	5	6	14	16	28	28

Table 5.2: Receptive fields of various layers in AlexNet.

Layer	$c1$	$p1$	$c2$	$p2$	$c3$	$c4$	$c5$	$p5$	$fc6$	$fc7$	$fc8$
Dimension	11	19	51	67	99	131	163	195	355	355	355

Table 5.3: Receptive fields of various layers in GoogLeNet.

Layer	$c1$	$p1$	$c2/reduce$	$c2$	$p2$	$inception3a$
Dimension	7	11	11	19	27	43
Layer	$inception3b$	$p3$	$inception4a$	$inception4b$	$inception4c$	$inception4d$
Dimension	59	75	107	137	171	203
Layer	$inception4e$	$p4$	$inception5a$	$inception5b$	$p5$	$fc1$
Dimension	235	267	331	395	459	459

To improve energy savings, SeFAct should be implemented at the earliest possible layer. However, the neurons in a specific layer can characterize the classes only if they see enough of the image to identify specific objects. For example, neurons in layer $c2$ of LeNet, AlexNet and GoogLeNet process information about $(14/28)^2 = 25\%, (51/227)^2 = 5\%$ and $(19/227)^2 = 0.7\%$ of the input image. Empirically, we choose to implement SeFAct from the layers whose receptive field covers about a quarter of the image, namely, from $c2$, $c5$ and $inception4b$ onwards in LeNet, AlexNet and GoogLeNet, respectively.

5.2.2 Choice of Data Bitwidth for Various Layers

SeFact cannot be implemented in the early layers of a CNN as they are unable to process enough data to correlate to specific classes due to limited receptive field. However, these early layers have high levels of resilience to inaccuracy. This provides an opportunity to

implement error sensitivity based circuit approximations for early layers. The reasons are as follows:

- The number of resilient neurons is significantly higher in initial layers of the network [32] in comparison to later layers. The reason is, neurons in the initial layers typically process features local to a certain region of the image, while the later layer neurons infer global features from the previously extracted local features.
- Errors in neurons near the inputs are more likely to be compensated/filtered out later in the network.

Therefore, we have achieved power savings through optimized reduced precision in early layers along with our selective activation approach for deeper layers.

5.3 Hardware Implementation

We implement our SeFAct scheme in combination with *optimized reduced precision bitwidths* in the testing phase. The *baseline implementation* of the testing phase is performed in three steps in each layer, L_i. First, the ifmap and the filter are loaded from the memory. Next, multiply-accumulate (MAC) operations are performed to compute the ofmap of layer based on (2.6), and data is written back to the memory. The ofmap computation leverages the ifmap data sparsity.

Memory hierarchies are used in neural network accelerators to reduce the cost of data movement [14, 27, 28]. Similar to [27], we assume that the hierarchy consists of a DRAM, then a 108 kB global SRAM buffer that services $12 \times 14 = 168$ neural processing elements (PE). Each PE has a total of 0.5 kB local register file (RF) storage. Each MAC computation requires four RF accesses: three read operations for the operands, and one write operation for the result.

The total computation energy is calculated as the product of the number of operations at each of the DRAM, SRAM, and RF levels, multiplied by the energy per unit operation (e_{DRAM}, e_{SRAM}, and e_{RF}) at each of these levels, incorporating the reduction in operation count from (2.6) due to sparsity.

Using $E_x^y, x \in \{r, w\}, y \in \{I, F, O\}$ to represent the energy for operation x and

computation y, the energy requirement, E, for layer L_i of the baseline testing phase is:

$$E = E_r^I + E_r^F + E_w^O + E_{RF} + E_{MAC} \tag{5.9}$$

where the first three memory access terms are a weighted sum of DRAM and SRAM energies. The weights correspond to the number of memory access to each level, which depends on the data movement and reuse pattern in the DRAM and SRAM. For both the baseline and our enhancement, all data communication with the DRAM (i.e., ifmap read or ofmap write) is performed in run-length compressed (RLC) format, incorporating data sparsity, which is decoded in the SRAM.

Our energy savings appear due to two factors, outlined in Section 5.2. The first contribution is due to the use of reduced bitwidths, which is a static optimization performed during the training phase, primarily in early layers of the network. The other contribution, obtained from later layers of the network, involves the addition of hardware that supports dynamic adaptation of computations during the testing phase, as described in Algorithm 7. We now summarize these changes.

Line 1 performs inexpensive mismatch computations which involves summations of one-bit numbers. The *min* computation in line 2 is a sequence of compare (i.e., subtract) operations over all clusters. The implementation cost of both lines 2 and 3 are linear in the number of clusters. Line 4 associates flags in $\tilde{\mathcal{F}}$ for all activated clusters.

Line 5 performs reduced ofmap computation using the flags in $\tilde{\mathcal{F}}$. Here, we only load the ifmap and filter data based on the important features, I. As described in Section 5.1.4, additional inexpensive $\tilde{\mathcal{F}}$ flags are prepared for intermediate layers of corresponding NIN modules. All the flags, such as I, $\tilde{\mathcal{F}}$, are stored in single-bit register files which are used in inexpensive decision circuitries to reduce memory access operations. Compared to the baseline, energy savings are achieved from (i) bitwidth reduction, (ii) fewer memory fetches, and (iii) a reduction in the number of MAC operations as only $\tilde{\mathcal{F}}$ features are computed. The change in bitwidth affects memory access energy linearly and computation energy quadratically, since the dominant component of MAC operations is multiplication, whose complexity is quadratic in the number of bits.

Line 6 detects the important features of ofmap in layer L_i, and only these features of the ofmap are written into the memory (note that due to the large volume of data at each level, the data within a level cannot be completely stored within the SRAM, and

DRAM writes are essential). This reduces the memory overhead and also effectively further increases the inherent sparsity (due to ReLU) for level L_{i+1}, which uses this ofmap as its ifmap. Line 7 prepares cluster activation flags to limit computations at layer L_{i+1} which is similar as line 4.

From (5.1), the energy overhead for detecting important features arises from (i) computation of the threshold and (ii) a comparison operation. We limit the summation of all ofmap data, F^{of} only to important features, $\tilde{\mathcal{F}}$, for the currently predicted classes. For these computations, we model energy for an n-bit adder as $n^{0.922} \times e_{add}$ [64]. The multiplication of this summation by T_1^+/K corresponds to a few add-and-shift operations: empirically, this number varies from 3–5 for standard CNN topologies, and the computed threshold can be represented by at most 6 bits. For the *Conv* layer, the check is simplified to (5.2), where the sparsity summation involves the addition of one-bit zero flags, an inexpensive operation.

The energy savings, ΔE for layer L_i can be formulated as:

$$\Delta E = \Delta E_r^I + \Delta E_r^F + \Delta E_w^O + \Delta E_{RF} + \Delta E_{MAC} - E_{ov} \tag{5.10}$$

where E_{ov} includes the energy associated with lines 1 through 4 and lines 6 through 7 in Algorithm 7.

5.4 Threshold Formulation

The SeFAct implementation process in *each layer* of a neural network depends on four thresholds, T_1, T_2, T_3 and T_4 as described in equations (5.1), (5.4), (5.6) and line 3 in Algorithm 7, respectively. We update these threshold notations as $T_{1,D}, T_{2,D}, T_{3,D}$ and $T_{4,D}$ where *depth*, D, is a parameter that identifies the location of a layer from the input layer in a network. For example, $D = 0$ and $D = N_L - 1$ are the input and output layers for a network, respectively, where N_L is the total number of the layers in the network. The SeFAct implementation prepares *predictive data* to save computation in the *subsequent layers*. Hence, we do not implement SeFAct in the last layer, $N_L - 1$, and start from the penultimate layer of the network, $D = N_L - 2$. We observe in Section 5.2.1, the earliest layer where SeFAct is beneficial, D_{min}, depends on the receptive field of the network. Hence, the total number of layers where SeFAct is implemented is $N_{SeFAct} = N_L - D_{min} - 1$.

The SeFAct implementation leverages a trade-off relation between accuracy and energy savings. We need many observations under various operating conditions of SeFAct to estimate the trend of this trade-off. However, to observe only one trade-off point, $4N_{SeFAct}$ parameters are required to be tuned. Additionally, there exists a complicated relationship between accuracy, energy savings, and thresholds of various layers of the network. This complexity makes the entire design space exploration very difficult.

We have developed a threshold tuning knob, τ, to modulate thresholds $T_{1,D} - T_{4,D}$ of N_{SeFAct} layers through a set of analytical equations with the help of eight additional parameters, $\alpha_0 - \alpha_7$. We systematically explore N_τ trade-off observations by varying the parameter τ for a specific ensemble of $\alpha_0 - \alpha_7$.

For each value of τ, we explore N_α ensembles of α parameters to attune the relationship among τ and thresholds $T_{1,D}, T_{2,D}, T_{3,D}$ and $T_{4,D}$. The exploration requires minimum and maximum limits for τ and the α parameters and is based on Latin hypercube sampling [65]. Hence, the number of tunable parameters becomes 18. However, based on the interdependencies of the analytical equations, we will show that the number of tunable parameters can be reduced from 18 to 5. Finally, we choose optimal operating conditions based on two additional parameters, β_{N_τ} and β_{APC_α}. Therefore, the designer specifies a total of 7 parameters that enable the exploration of the trade-off space. Note that the total number of tuning parameters does not change even if we seek a larger number of trade-off points: these points can be obtained by increasing the number of parameter samples.

Table 5.4: Required number of parameters for SeFAct implementation in N_{SeFAct} layers with and without analytical equations.

# Ensembles	# Data points	# Tunable parameters	
		Without analytical equations	With analytical equations
1	1	$4N_{SeFAct}$	7
1	N_τ	$4N_{SeFAct}N_\tau$	7
N_α	N_τ	$4N_{SeFAct}N_\tau N_\alpha$	7

A naïve alternative to our approach would be to explore the design space by tuning all $4N_{SeFAct}$ parameters. To find N_τ trade-off points with N_α sampled ensembles of

parameters, exploring the same volume of the design space would require the empirical tuning of $4N_{SeFAct}N_{\tau}N_{\alpha}$ parameters. The prohibitively large number of tunable parameters can make it hard to explore the design space to find optimal solution. Table 5.4 summarizes the required number of tunable parameters for N_{α} ensembles of α parameters and N_{τ} tuning parameters with and without the analytical equations.

Next, we describe the formulation of analytical equations modeling the relationship between τ and the thresholds of various layers and the algorithm that systematically analyze the trade-off between accuracy and energy savings. During this discussion, we will use the following notation:

- $T_{x,D}^{max/min}$ denotes the maximum/minimum value of threshold $T_{x,D}$ over all layer depths D, for all $\tau \in \{\tau_{min}, \tau_{max}\}$
- $T_{x,y}^{max/min}$ denotes the maximum/minimum value of threshold $T_{x,y}$ for a specific layer at depth y, for all $\tau \in \{\tau_{min}, \tau_{max}\}$.

5.4.1 Formulation of Threshold, $T_{1,D}$

The first step of the cluster learning phase of the SeFAct implementation is *important feature identification*, which depends on the threshold $T_{1,D}$ as described in Eq. (5.1). The earlier layers have smaller receptive fields, and they detect simpler and basic features. This threshold must be kept low to avoid discarding any essential features in the earlier layers. On the other hand, the later layers have larger receptive fields, and increasing $T_{1,D}$ can help reduce the number of predicted classes. Hence, for two layers with depth D_1 and D_2, where $D_2 > D_1$, we prefer $T_{1,D_2} > T_{1,D_1}$. We model the threshold $T_{1,D}$ in layers $D = \{D_{min}, \cdots, N_L - 2\}$ based on a geometric progression of the threshold tuning knob, τ. The energy savings can be increased by tuning τ to increase $T_{1,D}$. The empirical equation for $T_{1,D}$ is as follows:

$$T_{1,D} = \tau \times \left[\frac{D}{N_L - 2}\right]^{\alpha_0}, \quad D_{min} \leq D \leq N_L - 2, \text{ and } \alpha_0 > 0 \tag{5.11}$$

where, α_0 is a fitting parameter.

5.4.2 Formulation of Threshold, $T_{2,D}$

We use threshold $T_{1,D}$ to determine the important features for a particular input image of a class. Due to the network training inaccuracies, or image pixel pattern variations, all the images of the same class may not have the same activation pattern for a specific layer. The threshold $T_{2,D}$ is used to determine whether a certain feature is important for all images of an input class in layer D. As discussed in Eq. (5.4), we deem a feature to be important for a particular class if it is important for a sufficiently large number of images in the class. We model $T_{2,D}$ using a network-depth-independent parameter α_1.

$$T_{2,D} = \alpha_1, \qquad 0 \leq \alpha_1 \leq 1 \tag{5.12}$$

5.4.3 Formulation of Threshold, $T_{3,D}$

The threshold $T_{3,D}$ of layer D quantifies the affinity between two clusters and converts this affinity into a graph edge using Eq. (5.6). According to Eq. (5.5), the measure of cluster affinity is obtained based on the cluster-based important features, J_k, which evidently relies on the number of class-based important features, $\tilde{I}_c$. We compute $\tilde{I}_c$ using threshold $T_{1,D}$ as shown in equations (5.1)-(5.4). Hence, there is an interdependency between thresholds $T_{3,D}$ and $T_{1,D}$. This dependency is twofold:

1. *In the same layer* A higher value of $T_{1,D}$ in layer D corresponds to more aggressive approximation, which also requires more rigid clustering criteria, i.e., higher $T_{3,D}$. Hence, there is a positive nonlinear correlation between $T_{1,D}$ and $T_{3,D}$. We empirically model this relation using an exponential equation.

2. *Across layers* For any two layers at depth D_1 and D_2, where $D_2 > D_1$, the input classes will produce more similar feature patterns in the shallower layer at depth D_1 due to its smaller receptive field. Hence, for the same threshold $T_{1,D_1} = T_{1,D_2}$, the clusters in D_1 will have higher affinity than the clusters in layer D_2. Hence, we prefer to set the cluster threshold $T_{3,D_1} > T_{3,D_2}$ as a shallower layer would need a higher threshold to distinguish between classes. We model this observation using a product term which includes the deeper layer thresholds, $T_{1,D}$.

The analytical equation for $T_{3,D}$ that addreses both of these effects is as follows:

$$T_{3,D} = \alpha_2 \times \exp\{\alpha_3 \left(T_{1,D} - T^{max}_{1,N_L-2}\right)\} \prod_{d=D+1}^{N_L-2} T_{1,d}^{-\alpha_4} \tag{5.13}$$

where $\alpha_2 \in [0, 1]$ and $\alpha_3, \alpha_4 > 0$ are three fitting parameters. The exponential term lies within the range $(0, 1]$ as according to Eq. (5.11), $T^{max}_{1,D} = T^{max}_{1,N_L-2}$. The parameter α_2 scales $T_{3,D}$, the exponential term addresses the same-layer positive correlation between $T_{1,D}$ and $T_{3,D}$, and the product term incorporates the across-layer effect of network depth on clustering thresholds.

5.4.4 Formulation of threshold, $T_{4,D}$

The cluster activation operation for the *cluster testing phase* is regulated by the threshold $T_{4,D}$ as shown in line 3 in Algorithm 7. According to the algorithm, this threshold provides a margin for the activation of a cluster. For aggressive approximation to achieve higher energy savings, a smaller margin for the activation, i.e., lower $T_{4,D}$, is required. Similar to the threshold formulation of $T_{3,D}$, the threshold $T_{4,D}$ also has dependencies on $T_{1,D}$:

1. *In the same layer* A higher $T_{1,D}$ in layer D corresponds to an aggressive approximation which requires a lower $T_{4,D}$. We model the negative nonlinear correlation between $T_{1,D}$ and $T_{4,D}$ using a power-law equation.

2. *Across layers* For two layers with depth D_1 and D_2, where $D_2 > D_1$ and $T_{1,D_1} = T_{1,D_2}$, the shallower layer should activate a larger number of clusters to avoid faulty class prediction. This can be achieved by requiring $T_{4,D_1} > T_{4,D_2}$. As in the case of $T_{3,D}$, we model the across-layer effect of $T_{4,D}$ using a term related to the thresholds of later layers.

The overall equation is as follows:

$$T_{4,D} = \alpha_5 \times \left[\frac{T_{1,D}}{T^{min}_{1,N_L-2}}\right]^{-\alpha_6} \times \prod_{d=D+1}^{N_L-2} T_{1,d}^{-\alpha_7} \tag{5.14}$$

Here, $\alpha_5 \in [0, 1]$ is a scaling parameter that modulates $T_{4,D}$, $\alpha_6 > 0$ attunes the nonlinear relation between $T_{1,D}$ and $T_{4,D}$, and $\alpha_7 > 0$ modulates the across-layer relations

among the $T_{4,D}$ values of subsequent layers. The power-law term lies within the range $(0, 1]$.

5.4.5 Parameter Choices

Based on the description so far, the SeFAct implementation process requires nine parameters, τ and $\alpha_0 - \alpha_7$, as described in the analytical threshold formulations (5.11)-(5.14). However, the parameter τ alone can be used to modulate $T_{1,D}$, $T_{3,D}$ and $T_{4,D}$, and to tune the energy savings. A smaller value of τ reduces the important feature detection threshold, $T_{1,D}$, which results in the detection of a larger number of important features. This smaller value of $T_{1,D}$ then influences the clustering threshold $T_{3,D}$ (cluster activation threshold $T_{4,D}$) to be small (large), which improves the class prediction and classification accuracy but results in smaller energy savings. Similarly, a larger value of τ will lead to the opposite trends in the thresholds and energy savings. Thus, tuning τ helps explore the energy-accuracy trade-off space.

The initial constraints on the ranges of the parameters, based on the discussion of Sections 5.4.1-5.4.4, are listed in Table 5.5. The blank entries in the table indicate that no upper bound can be set from the initial understanding of the parameters. Next, we will update the ranges of these parameters based on network-dependent criteria and some heuristics. Once these are determined, we determine the trade-off between accuracy and energy by sampling the parameters from the updated range.

Table 5.5: Range of various parameters used in SeFAct implementation.

Parameter, x	τ	α_0	α_1	α_2	α_3	α_4	α_5	α_6	α_7
Minimum, x_{min}	0	0	0	0	0	0	0	0	0
Maximum, x_{max}	–	–	1	1	–	–	1	–	–

We will now discuss the criteria for choosing suitable ranges for these parameters to efficiently explore the design space.

1. **Choice of τ:** The tuning parameter τ directly influences $T_{1,D}$, as described in equation (5.11). By tuning τ in the interval $\{\tau_{min}, \tau_{max}\}$, $T_{1,D}$ can be modulated to change the number of features deemed important, and hence alter energy savings. We begin with a small value of $\tau_{min} \approx 0$ that provides the baseline accuracy, as

defined in the beginning of Section 5.3, and it is kept constant for all networks. On the other hand, the value of τ_{max} is strongly dependent on the properties of the network and the input data. When relatively larger networks are tasked with working with simpler input data sets, there are redundancies in feature detection and the important feature detection threshold can be higher. For example, in comparison with other networks running other applications, the task of classifying 10 handwritten digits is not very difficult for the LeNet network and very high accuracies (98.75% top-1 accuracy) are achievable; in contrast, the best achievable accuracy for AlexNet to work with 1000 ImageNet classes is lower (77.95% top-5 accuracy). For cases where the input data set is "easy" for the network, we have observed that we can use a large threshold margin, $T_{1,D}^{max} = T_{1,N_L-2}^{max} = \tau_{max} > 1$. On the other hand, for relatively smaller networks detecting complex features, the threshold should be smaller, and a somewhat conservative value, $0 < \tau_{max} < 1$, is required to be chosen.

2. **Choice of α_0:** The convexity/concavity trend of $T_{1,D}$ of Eq. (5.11) depends on the parameter α_0, as shown in Fig. 5.4. By definition, $T_{1,D}$ is convex (concave) if $\frac{\partial^2 T_{1,D}}{\partial {\alpha_0}^2} > 0$ $\left(\frac{\partial^2 T_{1,D}}{\partial {\alpha_0}^2} < 0\right)$. From Eq. (5.11), $\frac{\partial^2 T_{1,D}}{\partial {\alpha_0}^2} = \text{Constant} \times \alpha_0 \times (\alpha_0 - 1)$, where Constant > 0, hence $T_{1,D}$ is convex (concave) when $\alpha_0 > 1$ $(0 < \alpha_0 < 1)$.

   A choice of α_0 that results in a convex trend will assign lower thresholds to the shallower layers and more aggressive thresholds in the later layers. This will save less computations in early layers which will result in higher accuracy and lower energy savings. A concave trend, on the other hand, will reverse all these: higher thresholds will be assigned to shallower layers and less aggressive thresholds to later layers, resulting in lower accuracy and higher energy savings. Therefore, depending on the target accuracy, the value of α_0 that reflects the convex or concave trend can be chosen. We sample various α_0 and choose the optimal one based on the accuracy and energy savings trade-offs.

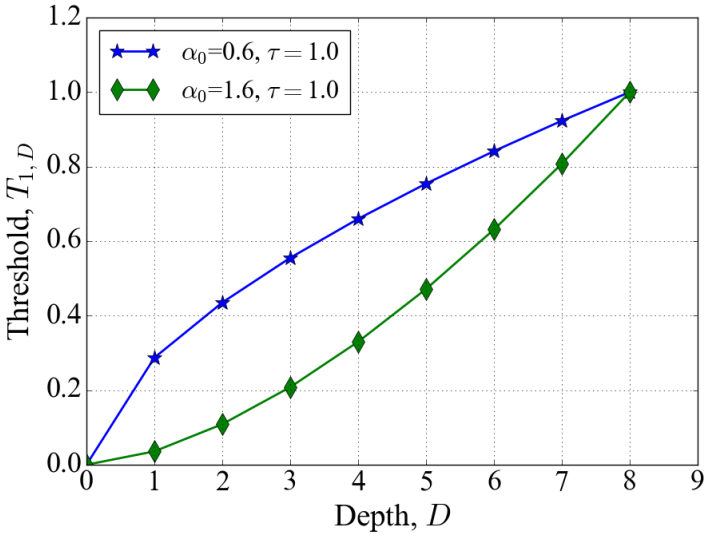


Figure 5.4: The effect of α_0 on $T_{1,D}$ using (5.11) for various network depth, D.

3. **Choice of α_1:** According to Section 5.4.2, α_1 is a network-independent parameter. If α_1 is too low then the number of important features, $\tilde{I}_c$ in (5.4) will be high and SeFAct will be not effective in providing energy savings. On the other hand, if α_1 is very large, the number of important features will reduce dramatically and affect the accuracy. Therefore, we prefer α_1 to be assigned some moderate value within the interval $[0, 1]$ to balance the energy savings and accuracy.

4. **Choice of α_2:** The threshold $T_{3,D}$ is used to prepare clusters by quantifying the affinity between classes. According to (5.13), threshold $T_{3,D}$ depends on a network-dependent scaling parameter, α_2. Networks with complex identification tasks should have lower $T_{3,D}$ to ensure cluster formulation. As described in Section!5.4.3, $T_{1,D}$ and $T_{3,D}$ have a positive correlation. Hence, if τ_{max}, the parameter controlling $T_{1,D}$, is set high value then α_2 must also be increased.

5. **Choice of α_3:** The exponential relation between $T_{3,D}$ and $T_{1,D}$ in Eq. (5.13) is regulated via the parameter, α_3. For efficient design space exploration, we want to keep T^{min}_{3,N_L-2} is within some factor, k_1, of T^{max}_{3,N_L-2}. The equation that ensures

the condition is as follows:

$$T^{min}_{3,N_L-2} = k_1 \times T^{max}_{3,N_L-2}, \qquad 0.2 \leq k_1 \leq 0.6$$
$$\alpha_3 = -\frac{\log_e k_1}{T^{max}_{1,N_L-2} - T^{min}_{1,N_L-2}} \tag{5.15}$$

where the range of k_1 is empirically chosen.

6. **Choice of α_4** The parameter α_4 in Eq. (5.13) is used to modulate the inter-layer relationship of cluster preparation threshold, $T_{3,D}$. We observe that the product term in (5.13) increases faster as we move to shallower layers. We choose α_4 in a way that ensures $T_{3,D} \leq 1, \forall D$. A detailed discussion on the choice of α_4 is provided in Appendix C, and leads to the following simplified equation for α_4:

$$\alpha_4 = \frac{k_2}{N_{SeFAct} - 1} \tag{5.16}$$

Here, N_{SeFAct} is the number of layers where we implement SeFAct and $0.05 \leq k_2 \leq 0.4$ is chosen based on the Eq. C.5 and empirical observations.

7. **Choice of α_5:** The threshold $T_{4,D}$ in Algorithm 7 is used as a cluster activation margin for layer D in the *cluster testing phase*. Networks with complex identification tasks should have higher $T_{4,D}$ for higher cluster activation to avoid significant accuracy degradation. According to (5.14), $T_{1,D}$ and $T_{4,D}$ have a negative correlation. The thresholds $T_{1,D}$ and $T_{4,D}$ are controlled by τ and α_5, respectively. Hence, the choice of α_5 should follow an opposite trend of τ, i.e., τ_{max}.

8. **Choice of α_6:** The criteria to choose α_6 is similar to α_3. The parameter α_6 is used to modulate the nonlinear relation between $T_{4,D}$ and $T_{1,D}$ as described in (5.14). We ensure that the minimum of T_{4,N_L-2} stays within a factor, k_3, from the maximum value, T^{max}_{4,N_L-2}. For simplicity, we choose $k_3 = k_1$ as they have similar ranges. The equation that ensures the condition is as follows:

$$T^{min}_{4,N_L-2} = k_1 \times T^{max}_{4,N_L-2}, \qquad 0.2 \leq k_1 \leq 0.6$$
$$\alpha_6 = -\frac{\log k_1}{\log T^{max}_{1,N_L-2} - \log T^{min}_{1,N_L-2}} \tag{5.17}$$

9. **Choice of α_7:** The parameter α_7 is used to modulate the inter-layer relationship between thresholds $T_{1,D}$ and $T_{4,D}$ as described in (5.14). The conditions to choose

α_7 is similar to that of α_4. Hence, for simplicity we use same equation for both of these parameters. The equation for α_7 is as follows:

$$\alpha_7 = \alpha_4 = \frac{k_2}{N_{SeFAct} - 1} \tag{5.18}$$

Table 5.6: Source of various parameters used in SeFAct implementation.

Parameter, x	τ	α_0	α_1	α_2	α_3
Minimum, x_{min}	Preset	Preset	Preset	Network dependent	Obtained from (5.15)
Maximum, x_{max}	Network dependent	Preset	Preset	Network dependent	Obtained from (5.15)
Parameter, x	α_4	α_5	α_6	α_7	
Minimum, x_{min}	Obtained from (5.16)	Network dependent	Obtained from (5.17)	Obtained from (5.18)	
Maximum, x_{max}	Obtained from (5.16)	Network dependent	Obtained from (5.17)	Obtained from (5.18)	

The analytical threshold formulations require nine parameters, τ and $\alpha_0 - \alpha_7$, for the SeFAct implementation process. We sample these parameters within an acceptable minimum and maximum values for an efficient design space exploration. The minimum of τ and α parameters are τ_{min}, and $\alpha_{min} = \{\alpha_{0,min}, \cdots, \alpha_{7,min}\}$, respectively. On the other hand, the maximum of τ and α parameters are τ_{max} and $\alpha_{max} = \{\alpha_{0,max}, \cdots, \alpha_{7,max}\}$, respectively. However, according to discussion on parameter choices, it is observed that there are only five network-dependent parameters, $\tau_{max}, \alpha_{2,min}, \alpha_{2,max}$, $\alpha_{5,min}$, and $\alpha_{5,max}$. The parameters τ_{min}, $\alpha_{0,min}, \alpha_{0,max}, \alpha_{1,min}$ and $\alpha_{1,max}$ are network-independent preset values and can be set a priori. The ranges for remaining parameters, $\alpha_3, \alpha_4, \alpha_6$ and α_7 are obtained using the network-dependent parameters and analytical equations (5.15)-(5.18). Table 5.6 summarizes the sources of all SeFAct parameters.

5.4.6 Algorithm for Obtaining the Optimal Parameter Set

We have developed an algorithm to explore N_α combinations of α vectors and obtain the optimal parameters in terms of accuracy and energy savings trade-off. A set of criteria have been proposed to discard suboptimal α choices from the design space. The criteria are as follows:

- *Number of Acceptable Data Points* We implement our SeFAct scheme for N_τ accuracy and energy savings trade-off points for each α vector. We discard the trade-off data points for which the network accuracies are smaller than minimum acceptable accuracy, Acc_{min}. Hence, the number of remaining observations are $N'_\tau \leq N_\tau$ and if N'_τ is very small, the particular α vector is suboptimal. We use a tunable parameter, β_{N_τ}, to discard the α set if $N'_\tau \leq \beta_{N_\tau} \times N_\tau$.

- *Number of Average Probable Classes* During the testing phase, an input image activates the clusters that have similar feature patterns to reduce computations. We define *average probable classes*, $APC_{\alpha\tau D}$, as the average number of predicted classes for all the input test images in layer D for a specific combination of parameters, $\{\alpha, \tau\}$. We compute average probable class, APC_α, over all $D \leftarrow \{D_{min}, \cdots, N_L - 2\}$ and τ for each α vector. The APC_α acts as a metric for computation reduction for the α vector. A smaller value of APC_α indicates narrower class prediction for the given α which contributes to higher energy savings. The α vectors with very large APC_α are suboptimal for SeFAct implementation. The maximum number of probable classes in a layer can be the total number of classes in a network, N_{class}. Hence, the maximum average probable class over all N_{SeFAct} layers is: $APC_\alpha^{max} = N_{class} N_{SeFAct}$. We discard the suboptimal α vectors that produce $APC_\alpha > \beta_{APC_\alpha} \times APC_\alpha^{max}$, where β_{APC_α} is a tunable parameter.

Algorithm 8 summarizes the process to obtain optimal operating parameters for the SeFAct implementation process. The inputs to the algorithm are as follows:

- *Network properties* are specific to a given network, such as total number of layers, N_L, and the original accuracy without the SeFAct process, Acc_{orig}.

- *Network-dependent SeFAct parameters* are dependent on user preferences. For a target application, the user will decide the minimum acceptable accuracy of the network, Acc_{min}. The other network-dependent properties such as first layer to implement SeFAct, D_{min}, and the SeFAct parameters, $\tau_{max}, \alpha_{2,min}, \alpha_{2,max}$ and $\alpha_{5,min}, \alpha_{5,max}$ will be chosen according to Section 5.4.5.

- *Network-independent SeFAct parameters* can be set a priori for any types of network. The algorithm creates total N_α sets of acceptable α parameters to find

optimal network parameters, where N_α is an input to the algorithm. The other network-independent inputs are minimum relative threshold, τ_{min}, and the number of tunable thresholds, N_τ. Two additional parameters β_{N_τ} and β_{APC_α}, as described in early part of Section 5.4.6, are used to discard suboptimal data points.

Algorithm 8 starts with creating a set, S_D, to store the identities of all the SeFAct layers (line 1). Line 2 prepares the acceptable minimum and maximum range of each α parameter based on the network-dependent input parameters. Next, N_α samples of the α parameters are generated in line 3 using Latin hypercube sampling algorithm (LHS) [65]. For each α sample vector, an empty set, S_α, is initialized to store the accuracy and energy savings (line 4 5). Simultaneously, N_τ tunable thresholds are generated in the interval $\{\tau_{min}, \tau_{max}\}$ (line 6). For each τ, all the necessary thresholds are obtained for the SeFAct implemented layers based on the analytical equations (line 8). The algorithm uses these thresholds to learn important features and clusters in N_{SeFAct} layers using Algorithm 6. Line 10 tests the SeFAct implementation in real time. It checkes the cluster activation and reduces the computations accordingly based on Algorithm 7. The network accuracy, $Acc_{\alpha\tau}$ is later computed in line 11. The $Acc_{\alpha\tau}$ and percentage energy savings, $PES_{\alpha\tau}$, are stored in the set S_α if $Acc_{\alpha\tau} \geq Acc_{min}$ (line 15). The algorithm also computes the average probable class of each layer, $APC_{\alpha\tau D}$. Simultaneously, a moving average of $APC_{\alpha RD}$ over all SeFAct layers of all the selected data points, APC_α is also computed. As described in the earlier part of section 5.4.6, APC_α indicates the extent of probable class reduction for a given α.

After performing necessary computations for all N_τ points, the algorithm decides whether the current $\alpha[i]$ sample vector is suboptimal. The check of suboptimal α vector in line 19 depends on two parameters, β_{APC_α} and β_{N_τ}, based on the discussion of Section 5.4.6. The remaining α vectors are the candidates for optimal combination. Later, the algorithm models the trend of the accuracy and PES for the candidate combinations to predict PES for an acceptable accuracy interval. We use a linear fitting model for simplicity. The slope and intercept of the linear fit are stored in a map format with each candidate α vector as the key (line 20 and 21). The final step of the algorithm is to choose optimal α combinations based on the maximum achievable energy savings for N_τ points in the accuracy interval $\{Acc_{min}, Acc_{orig}\}$.

Algorithm 8 Algorithm to obtain optimal α parameters

INPUT: Network Properties: Number of layers in the network: N_L; Original accuracy: Acc_{orig};

Network Dependent SeFAct Properties: Minimum acceptable accuracy: Acc_{min}; Minimum depth of the SeFAct layer: D_{min}; SeFAct parameters: $\tau_{max}, \{\alpha_{2,min}, \alpha_{2,max}\}$ and $\{\alpha_{5,min}, \alpha_{5,max}\}$

Network Independent SeFAct Properties: Number of α samples: N_α; minimum threshold tuning parameter: τ_{min}; Number of tunable threshold points: N_τ; Data point threshold: β_{N_τ}; Average probable class threshold: β_{APC_α}

OUTPUT: Optimal parameters: $\alpha_{opt} \in \mathbb{R}^{N_\tau \times 8}$

METHOD:

$S_D = \{D_{min}, \cdots, N_L - 2\}$

▷ Prepare α parameters using (5.15)-(5.18), τ_{min}, τ_{max} and D_{min}

Obtain $\alpha_{min} = \{\alpha_{0,min}, \cdots, \alpha_{7,min}\}$ and $\alpha_{max} = \{\alpha_{0,max}, \cdots, \alpha_{7,max}\}$

Generate Latin hypercube samples, $\alpha = \text{LHS}(\alpha_{min}, \alpha_{max}, N_\alpha) \in \mathbb{R}^{N_\alpha \times 8}$

for each $\alpha[i] \in \mathbb{R}^8 \quad i \leftarrow \{0, 1, \cdots, N_\alpha - 1\}$ **do**

  Initialize $S_\alpha \leftarrow \varnothing$

  Generate data points $\tau = \text{linspace}(\tau_{min}, \tau_{max}, N_\tau)$

  for each $\tau[j] \quad j \leftarrow \{0, 1, \cdots, N_\tau - 1\}$ **do**

    ▷ Compute thresholds using (5.11)-(5.14) and $\alpha[i], \tau[j]$

    Compute $T_{1,D}, T_{2,D}, T_{3,D}, T_{4,D}, \quad \forall D \in S_D$

    Learn clusters for layer D using Algorithm 6, $\quad \forall D \in S_D$

    Test SeFAct implementation for layer D using Algorithm 7, $\quad \forall D \in S_D$

    Compute network accuracy, $Acc_{\alpha\tau}$

    if $Acc_{\alpha\tau} > Acc_{min}$ **then**

      Compute percentage energy savings , $PES_{\alpha\tau}$ using (5.9) and (5.10)

      Obtain average probable classes, $APC_{\alpha\tau D}, \quad \forall D \in S_D$

      $S_\alpha \leftarrow S_\alpha \cup \{\{Acc_{\alpha\tau}, PES_{\alpha\tau}\}\}$

      $APC_\alpha += \left(\sum_D APC_{\alpha\tau D}/N_{SeFAct} - APC_\alpha\right)/|S_\alpha|$

    end if

  end for

  if $N'_\tau = |S_\alpha| > \beta_{N_\tau} N_\tau$ and $APC_\alpha < \beta_{APC_\alpha} APC_\alpha^{max}$ **then**

    Compute $\text{Slope}_\alpha, \text{Intercept}_\alpha = \text{curvefit}(S_\alpha)$

    $\text{Slope} = \text{map}\,\langle \text{key} = \alpha[i], \text{value} = \text{Slope}_\alpha \rangle$

    $\text{Intercept} = \text{map}\,\langle \text{key} = \alpha[i], \text{value} = \text{Intercept}_\alpha \rangle$

  end if

end for

Generate $Acc = \text{linspace}(Acc_{min}, Acc_{orig}, N_\tau)$

for $x \leftarrow \{0, 1, \cdots, N_\tau - 1\}$ **do**

  ▷ Obtain optimal parameters, $\alpha_{opt}[x]$ for each $Acc[x]$ points

  Assign $\alpha_{opt}[x] = \text{key}$ for $\max_{\text{key}}\{\text{Slope[key]} \times Acc[x] + \text{Intercept[key]}\}$

end for

5.5 Results

The section is organized as follows: first, we report the simulation parameters and models. Next, we discuss the optimization of two orthogonal approximation processes implemented in this work: reduced bitwidth and selective feature activation Finally, we present the effect of both approximation approaches on accuracy and energy savings for optimal operating points.

5.5.1 Simulation Parameters/Models

We demonstrate our energy-efficient CNN framework on three well-studied networks, LeNet [41] applied to the MNIST [66] handwritten digit recognition dataset, AlexNet [39] and GoogLeNet [44] both applied to the ImageNet [67] dataset using Caffe platform [68].

We assume the baseline network, as defined at the beginning of Section 5.3, uses 8-bit words for ifmaps, filters, and ofmaps [13,26]. We use 5,000 (10,000) images for the cluster learning phase and 2,000 images for testing phase for LeNet (GoogLeNet/AlexNet). The top-1 (top-5) accuracy of LeNet (GoogLeNet/AlexNet) for the baseline (8-bit implementation) is 99.06% (89.00%/77.95%). We use CNNergy [69], an open-sourced simulator [70], to determine the number of DRAM, SRAM, and RF memory accesses and computations for GoogLeNet, AlexNet and LeNet. The per unit energy for DRAM, SRAM, and RF memory accesses are obtained from [14].

5.5.2 Reduced Bitwidths

We implement reduced precision bitwidths from the *input* layer to the $c1$, $c4$ and *inception4a* layer in LeNet, AlexNet and GoogLeNet, respectively. For later layers, the bitwidths are the same as the baseline.

The reduced precision bitwidths in various layers of the CNN trade-off classification accuracy and energy savings. We used Latin hypercube sampling [65] based Monte-Carlo simulations for 500 prospective bitwidth combinations in early layers and checked the classification accuracy on 5,000 test images. The optimal bitwidth is chosen based on the criteria of maximum energy savings for the minimum accuracy drop. The modified bitwidths in LeNet, AlexNet and GoogLeNet are listed in Tables 5.7, 5.8 and 5.9, respectively. The *Pool* and *Norm* layers use the same bitwidths as their immediately

preceding *Conv* layers.

Table 5.7: Modified bitwidths of early layers in LeNet.

Layer	*input*	$c1$	$c2$	$fc1$	$fc2$
ifmap/ofmap	4	3	8	8	8
filter	3	4	8	8	

Table 5.8: Modified bitwidths of early layers in AlexNet.

Layer	*input*	$c1$	$c2$	$c3$	$c4$	$c5$	$fc6$	$fc7$	$fc8$
ifmap/ofmap	6	7	5	6	5	8	8	8	8
filter	8	7	7	6	7	8	8	8	

Table 5.9: Modified bitwidths of early layers in GoogLeNet.

Layer	*input*	$c1$	$c2/reduce$	$c2$	*inception3a*	*inception3b*	*inception4a*
ifmap/ofmap	7	7	6	7	6	7	7
filter	7	7	8	7	8	7	7
Layer	*inception4b*	*inception4c*	*inception4d*	*inception4e*	*inception5a*	*inception5b*	$fc1$
ifmap/ofmap	8	8	8	8	8	8	8
filter	8	8	8	8	8	8	

5.5.3 Selective Feature Activation

We use our clustering based method for selective feature activation, as explained in Section 5.1.2, for all layers from $c2$, $c5$ and *inception4b* for LeNet, AlexNet and GoogLeNet, respectively. During the testing phase, the input image activates the clusters similar to its feature pattern to reduce class prediction. For example, we find that the images of various non-shedding dogs (e.g., shih-tzus, spaniels, and terriers) activate the same cluster in layer $fc7$ of AlexNet.

Early prediction of a reduced number of classes reduces the number of computations and the energy, as compared to the baseline. The amount of reduction in the number of classes based on the SeFAct prediction depends on thresholds $T_{1,D}, T_{2,D}, T_{3,D}$ and $T_{4,D}$, which rely on the α parameters and threshold tuning knob, τ, as detailed in Section 5.4. For each α parameter combination, the tuning knob τ can be varied to achieve various trade-off data points. Next, we will show the effect of parameters α and τ on accuracy

and energy savings, as well as simulation setups to find optimal operating parameters.

Effect of Parameters α and τ The trade-off between the percentage energy savings, PES, and the accuracy (all normalized to the baseline) for various combinations of α (represented with various colors) and τ parameters for LeNet, GoogLeNet, and AlexNet are shown in Fig. 5.5(a), (b), (c), respectively. The red vertical lines show the accuracy of the baseline. The accuracy gap between the red vertical line and the rightmost points in the scatter plots can be attributed to the loss in accuracy due to reduced bitwidth approximation in the early layers. The reduced bitwidth contributes towards a fixed energy savings for all operating conditions as shown in green horizontal lines in each plot. All energy savings above this line are due to our selective activation approach. The figure shows that, depending on the choice of the α parameters, different trade-offs between the energy savings and accuracy can be obtained for the same τ.

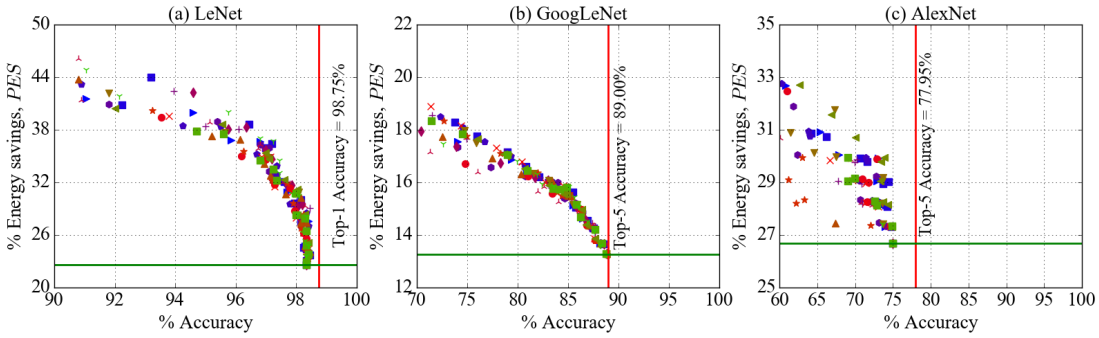


Figure 5.5: PES vs. accuracy for various combinations of the α parameters for (a) LeNet (b) GoogLeNet (c) AlexNet.

Finding Optimal Operating Parameters Algorithm 8 automates the search for optimal α combination. It first discards the suboptimal α combinations and outputs the optimal α set based on the detection of envelope for the linearly modelled trade-off trends of various α parameter sets. A linear fit of the energy savings and accuracy trade-off for one α combination is shown in Fig. 5.6. It can be seen that the linear model provides a good estimation of the trend.

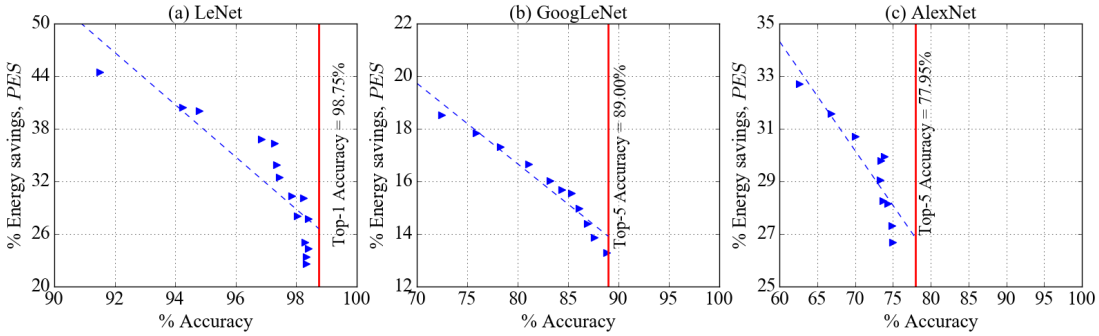


Figure 5.6: Example of linear modeling for PES vs. accuracy for (a) LeNet (b) GoogLeNet (c) AlexNet.

Fig. 5.7 shows the linearly fitted trade-off curves of various α parameters using dashed lines. The envelope of all trade-off curves is shown using the black solid line. It can be seen that the envelope is piecewise linear i.e. for different accuracy interval, the α set is different. The optimal α parameters for various networks obtained from Algorithm 8 are listed in Table D.1 of Appendix D.

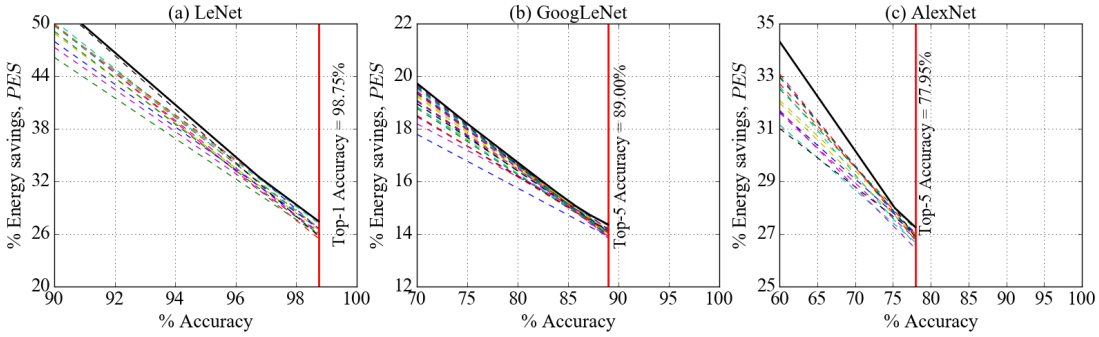


Figure 5.7: Optimal PES vs. accuracy envelope using Algorithm 8 for (a) LeNet (b) GoogLeNet (c) AlexNet.

From these plots, we observe that significant energy savings are achievable using our systematic analysis: for small (5–10%) degradations in accuracy, 15–25% energy savings are possible. The inexpensive overhead circuitries, described in Section 5.3, consume less than 1% of the total energy. For scenarios where low accuracy is acceptable (e.g., in mobile embedded systems or edge devices, where battery limitations are the paramount

consideration, and a best-effort accuracy is good enough), improvements of almost 30-40% are visible.

Simulation Setups to Find Optimal Operating Parameters The required inputs for Algorithm 8 for various networks are provided in Table 5.10. We chose the number of α samples, $N_\alpha = 50$, and the number of threshold tuning parameters, $N_\tau = 16$.

Table 5.10: Parameters used for SeFAct implementation for various networks.

Network		LeNet	GoogLeNet	AlexNet
Network parameters	N_{class}	10	1000	1000
	N_L	5	14	9
	Acc_{orig}	99.06% (Top-1)	89.00% (Top-5)	77.95% (Top-5)
User input	Acc_{min}	90.00% (Top-1)	70.00% (Top-5)	60.00% (Top-5)
Designer choice	N_{SeFAct}	2	6	3
	τ_{max}	1.50	1.20	0.80
	$\{\alpha_{2,min}, \alpha_{2,max}\}$	{0.30,0.80}	{0.25,0.75}	{0.10,0.40}
	$\{\alpha_{5,min}, \alpha_{5,max}\}$	{0.10,0.50}	{0.15,0.60}	{0.20,0.70}
	β_{N_τ}	0.50		
	β_{APC_α}	0.95		
Network independent empirical parameters	τ_{min}	0.10		
	$\{\alpha_{0,min}, \alpha_{0,max}\}$	{0.60,1.60}		
	$\{\alpha_{1,min}, \alpha_{1,max}\}$	{0.55,0.65}		

The input parameters for the algorithm are of four types:

1. *User input* The user will choose a neural network and the minimum acceptable accuracy, Acc_{orig}, for the network.

2. *Network parameters* The basic network properties for the user-provided network, such as total number of layers in the network, N_L, the total number of classes, N_{class} and the baseline accuracy, Acc_{orig} are included in the network parameters.

3. *Designer choice* Based on the user-provided network information, the designer will tune the following network-dependent operating parameters:

   (a) N_{SeFAct}, the total number of layers where SeFAct will be implemented, is obtained based on the receptive field criteria described in Section 5.2.1.

(b) τ_{max} is chosen based on the network complexity and classification task. According to the discussion of Section 5.4.5, we have chosen larger τ_{max} for LeNet compared to AlexNet. As GoogLeNet is better trained for complex classification task of ImageNet data than that of AlexNet, τ_{max} is larger.

(c) The range of α_2 (α_5) is selected while maintaining positive (negative) correlation with τ_{max}, as mentioned in criterion 4 (criterion 7) of Section 5.4.5.

(d) The parameters, β_{N_τ} and β_{APC_α} are used to discard suboptimal α vectors in line 19 of Algorithm 8. The designer may choose to impose less/more aggressive check for suboptimal points.

4. *Empirical tuning* During the development of SeFAct process, we have determined some network-independent parameters based on empirical observations. For example, as mentioned in criterion 1 of Section 5.4.5, we prefer relatively small τ_{min} to attain the baseline accuracy. Additionally, the parameter α_0 determines whether the trend of $T_{1,D}$ of Eq. (5.11) becomes convex or concave. We set the range of α_0 to cover both trends to observe how the trend affects the accuracy and energy savings trade-off. On the other hand, based on the explanation in Section 5.4.5, we have varied α_1 within a moderate interval.

5.5.4 Accuracy and Energy Savings Trade-off for Optimal Parameters

In this section, we show the combined effect of reduced bitwdith and SeFAct approximations for the optimal parameters. The average number of probable classes, $APC_{\alpha\tau D}$ of each layer, D, and the layer-wise energy for the baseline and our enhancement, are shown for LeNet, GoogLeNet, and AlexNet in Figs. 5.8, 5.9 and 5.10, respectively. Here, all results are shown using the optimal α combinations obtained from Algorithm 8 for various tunable thresholds, τ.

Figures 5.8(a)–5.10(a) show the number of average probable classes, $APC_{\alpha\tau D}$ for each layer on which SeFAct is applied, whereas Figures 5.8(b)–5.10(b) report the energy requirement for individual layers of the network. It can be seen that the number of average probably classes is reduced monotonically, resulting in a significant reduction in energy requirement with respect to the baseline, at the cost of a loss in accuracy. For example, the total number of classes in LeNet is 10 and the number of average

probable classes is reduced to 6 in the $fc1$ layer for a classification accuracy of 96.8%. For GoogLeNet and AlexNet, the total number of classes is 1000, and the number of average probable classes is reduced to 6 (445/763) in the *inception5b* and $fc7$ layers, respectively. The corresponding classification accuracy is 78.2% for GoogLeNet and 66.8% for AlexNet.

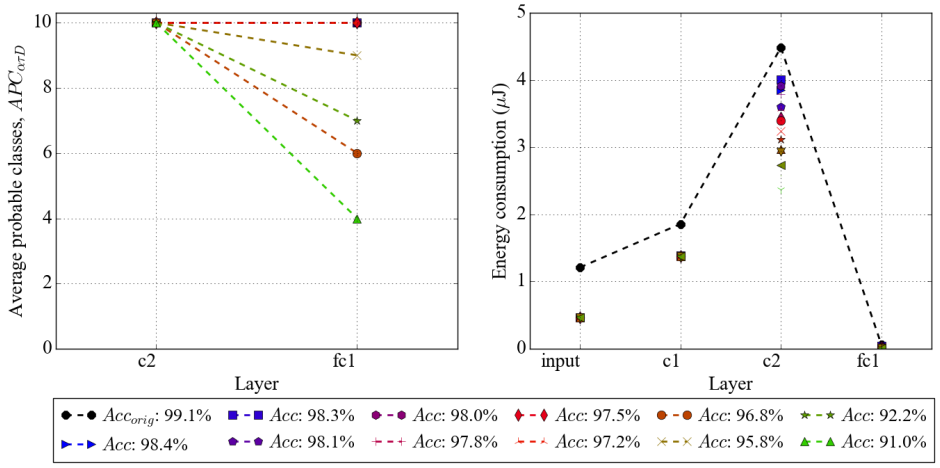


Figure 5.8: (a) Average probable classes in layers with SeFAct implementation and (b) layer-wise energy consumption in LeNet.

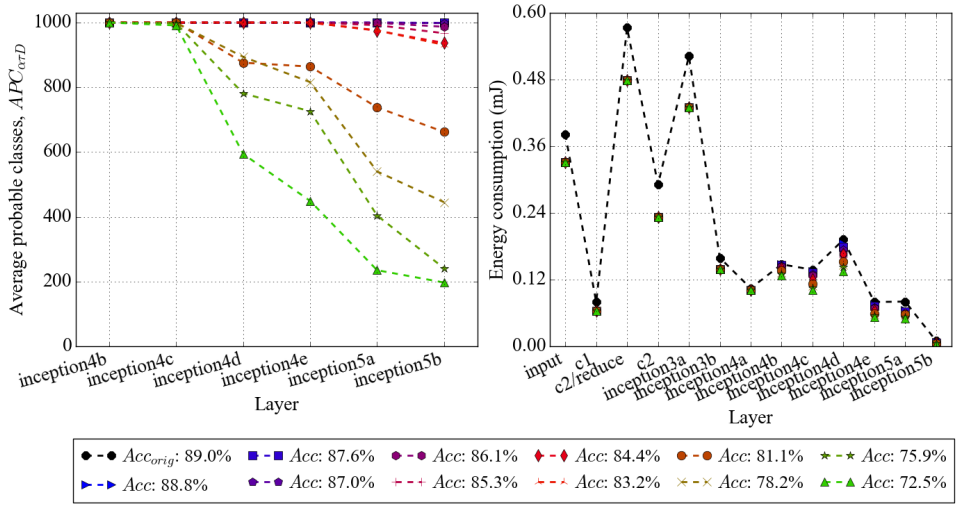


Figure 5.9: (a) Average probable classes in layers with SeFAct implementation and (b) layer-wise energy consumption in GoogLeNet.

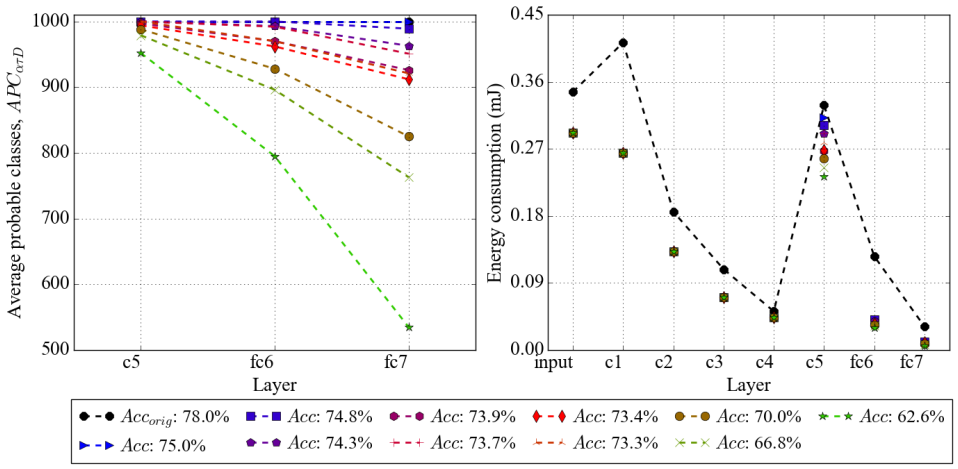


Figure 5.10: (a) Average probable classes in layers with SeFAct implementation and (b) layer-wise energy consumption in AlexNet.

For these CNNs, the energy savings in the early layers are attributable to the optimized bitwidth, while those in later layers are attributed to the selective feature activation. For the same energy savings, the relative percentage contributions between reduced bitwidth approximation and SeFAct are about 60% and 40%, respectively.

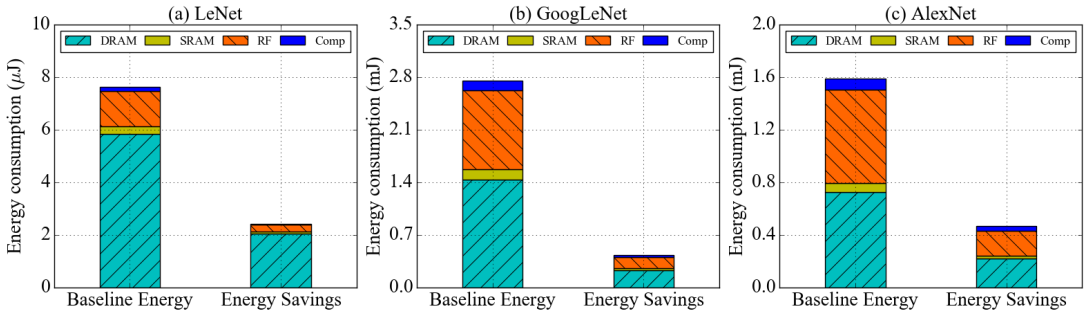


Figure 5.11: Contribution of energy consumption of various operations for baseline energy and energy savings for (a) LeNet (b) GoogLeNet (c) AlexNet.

The total energy and the contributions from DRAM, SRAM, and RF memory accesses, and MAC computations are shown in Fig. 5.11. The results for both the baseline implementation and the energy savings for our approach in LeNet, GoogLeNet and AlexNet are shown, and it is easily seen that the memory access operations are the

dominant components of all three cases.

5.6 Conclusion

This chapter has proposed an effective and automated way to dynamically reduce the energy of a CNN accelerator, using bitwidth reduction in early layers, and selective feature activation in later layers. Large energy savings are seen, even for small accuracy losses for three well-known networks. The concepts and framework of this chapter can be generalized for other types of CNNs beyond those evaluated in our study.

Chapter 6

Thesis Conclusion

The recent slow growth of traditional CMOS technology and the ever-expanding functionalities of mobile devices have driven the need for energy efficient system design. For the error-tolerant applications, such as multimedia, artificial intelligence etc, the deliberate introduction of errors to reduce energy requirement has become a viable solution to design low power system. Approximate computation is a new paradigm that explores the trade-off relation between energy and accuracy in various such applications. The design of approximate hardware requires some careful adjustment based on the application requirement, system architecture as well as input characteristics. No one design will improve the energy efficiency of all types of applications. The fundamental mechanism of approximation may be the same, but the parameters and the systems must be tuned separately based on the application characteristics. This thesis has developed novel techniques for the design, simulation, and optimization of widely-used error-tolerant applications for mobile devices: JPEG architecture and neural networks.

We have first explored the arena of system-dependent approximation by exploiting the error sensitivity of JPEG architecture. The architecture-sensitive approximate design showed significant energy improvement over previous approximate designs despite adopting the conservatism in error introduction to meet the user-defined error budget for any input. Next, we expanded the approximation scheme design based on input-dependent information. The exploitation of input patterns helps to increase the approximation level in real-time. We find that the input-dependent approximation scheme is superior to the input-independent approach as the additional overhead is paid off by

the achieved large energy gains for reducing the conservatism of the first approach.

Afterwards, we have introduced the input-dependent approximation idea in widely used computationally intensive neural network architectures. We have developed a selective feature activation framework, SeFAct, that dynamically prune out inessential computations to save energy with the cost of a small accuracy drop. We have also prepared an algorithm and necessary analytical equations to automate the process that helps to efficiently obtain optimal results.

The input-independent and input-dependent approximation frameworks have been validated on standard benchmarks for the corresponding applications.

References

[1] Y. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, and M. Smelyanskiy. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE*, 96(5):790–807, May 2008.

[2] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[3] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the IEEE European Test Symposium*, pages 1–6, 2013.

[4] D. Sengupta, J. Hu, and S. S. Sapatnekar. *Error analysis and optimization in approximate arithmetic circuits*, pages 225–246. Springer, New York, NY, 2019.

[5] V. Bhaskaran and K. Konstantinides. *Image and video compression standards: algorithms and architectures*. Kluwer Academic Publishers, 1997.

[6] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, 2013.

[7] J. Park, J. H. Choi, and K. Roy. Dynamic bit-width adaptation in DCT: An approach to trade off image quality and computation energy. *IEEE Transactions on VLSI Systems*, 18(5):787–793, 2010.

[8] F. S. Snigdha, D. Sengupta, J. Hu, and S. S. Sapatnekar. Optimal design of JPEG hardware under the approximate computing paradigm. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*, pages 1–6, 2016.

[9] F. S. Snigdha, D. Sengupta, J. Hu, and S. S. Sapatnekar. Dynamic approximation of JPEG hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):295–308, 2019.

[10] D. Sengupta, F. S. Snigdha, J. Hu, and S. S. Sapatnekar. SABER: Selection of approximate bits for the design of error tolerant circuits. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*, 2017.

[11] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. 521:436 EP –, 2015.

[12] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM International Symposium on Computer Architecture*, pages 1–12, 2017.

[14] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.

[15] F. S. Snigdha, I. Ahmed, S. D. Manasi, M. G. Mankalale, J. Hu, and S. S. Sapatnekar. SeFAct: Selective feature activation and early classification for CNNs. In

Proceedings of the Asia-South Pacific Design Automation Conference, pages 487–492, 2019.

[16] A. Alaghi, W. J. Chan, J. P. Hayes, A. B. Kahng, and J. Li. Optimizing stochastic circuits for accuracy-energy tradeoffs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 178–185, Nov 2015.

[17] G. Varatkar and N. Shanbhag. Energy-efficient motion estimation using error-tolerance. In *Proceedings of the ACM International Symposium on Low Power Electronics and Design*, pages 113–118, 2006.

[18] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(4):850–862, 2010.

[19] W. Xu, S. S. Sapatnekar, and J. Hu. A simple yet efficient accuracy-configurable adder design. *IEEE Transactions on VLSI Systems*, 26(6):1112–1125, June 2018.

[20] L. B. Soares, S. Bampi, and E. Costa. Approximate adder synthesis for area- and energy-efficient FIR filters in CMOS VLSI. In *Proceedings of the IEEE International New Circuits and Systems Conference*, pages 1–4, 2015.

[21] B. Shao and P. Li. A model for array-based approximate arithmetic computing with application to multiplier and squarer design. In *Proc. ISLPED*, pages 9–14, 2014.

[22] L. N. Chakrapani, K. K. Muntimadugu, A. Lingamneni, J. George, and K. V. Palem. Highly energy and performance efficient embedded computing through approximately correct arithmetic: A mathematical foundation and preliminary experimental validation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 187–196, 2008.

[23] J. Miao, K. He, A. Gerstlauer, and M. Orshansky. Modeling and synthesis of quality-energy optimal approximate adders. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 728–735, 2012.

[24] A. Madanayake, R. J. Cintra, V. Dimitrov, F. Bayer, K. A. Wahid, S. Kulasekera, A. Edirisuriya, U. Potluri, S. Madishetty, and N. Rajapaksha. Low-power VLSI architectures for DCT/DWT: Precision vs approximation for HD video, biomedical, and smart antenna applications. *IEEE Circuits and Systems Magazine*, 15(1):25–47, Firstquarter 2015.

[25] A. Zlateski, K. Lee, and H. S. Seung. ZNN – A fast and scalable algorithm for training 3D convolutional networks on multi-core and many-core shared memory machines. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 801–811, 2016.

[26] P. Gysel, M. Motamedi, and S. Ghiasi. Hardware-oriented approximation of convolutional neural networks, 2016. arXiv preprint arXiv:1604.03168.

[27] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[28] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. DaDianNao: A machine-learning supercomputer. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[29] Z. Du, K. Palem, A. Lingamneni, O. Temam, Y. Chen, and C. Wu. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *aspdacFull*, pages 201–206, Jan 2014.

[30] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding, 2015. arXiv preprint arXiv:1510.00149.

[31] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning*, pages 1737–1746, 2015.

[32] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. AxNN: Energy-efficient neuromorphic systems using approximate computing. In *Proceedings of*

the ACM International Symposium on Low Power Electronics and Design, pages 27–32, 2014.

[33] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 1–7, 2016.

[34] G. K. Wallace. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, Feb 1992.

[35] J. F. Blinn. What's that deal with the DCT? *IEEE Computer Graphics and Applications*, 13(4):78–83, July 1993.

[36] W. Chen, C. Smith, and S. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*, 25(9):1004–1009, 1977.

[37] B. Lee. A new algorithm to compute the discrete cosine transform. *IEEE Transactions on Acoustics Speech and Signal Processing*, 32(6):1243–12457, 1984.

[38] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 988–991, 1989.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 1097–1105, 2012.

[40] V. Nair and G. E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the International Conference on Machine Learning*, pages 807–814, 2010.

[41] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[42] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. arXiv preprint arXiv:1502.03167.

[43] M. Lin, Q. Chen, and S. Yan. Network in network, 2013. arXiv preprint arXiv:1312.4400.

[44] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition*, June 2015.

[45] A. Mammeri, A. Khoumsi, D. Ziou, and B. Hadjou. Modeling and adapting JPEG to the energy requirements of VSN. In *Proceedings of the International Conference on Computer Communications and Networks*, pages 1–6, 2008.

[46] Y. Emre and C. Chakrabarti. Energy and quality-aware multimedia signal processing. *IEEE Transactions on Multimedia*, 15(7):1579–1593, 2013.

[47] K. Nepal, Y. Li, R. I. Bahar, and S. Reda. ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. In *Proceedings of the Design, Automation & Test in Europe*, pages 361:1–361:6, 2014.

[48] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu. Joint precision optimization and high level synthesis for approximate computing. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*, pages 1–6, 2015.

[49] NanGate 45nm Open Cell Library. `http://www.si2.org/openeda.si2.org/projects/nangatelib`, Accessed May 30, 2019.

[50] Synopsys, Inc. Design Compiler. `http://beethoven.ee.ncku.edu.tw/testlab/course/VLSIdesign_course/course_96/Tool/Design_Compiler%20_User_Guide.pdf`, Accessed May 30, 2019.

[51] Standard image database. `http://sipi.usc.edu/database/`. Accessed May 30, 2019.

[52] N. Asuni and A. Giachetti. TESTIMAGES: A large data archive for display and algorithm testing. *Journal of Graphics Tools*, 17(4):113–125, 2013.

[53] KNITRO user manual. `https://www.artelys.com/tools/knitro_doc/`. Accessed May 30, 2019.

[54] Z.-L. He, C.-Y. Tsui, K.-K. Chan, and M. L. Liou. Low-power VLSI design for motion estimation using adaptive pixel truncation. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):669–678, 2000.

[55] A. Raha, H. Jayakumar, and V. Raghunathan. Input-based dynamic reconfiguration of approximate arithmetic units for video encoding. *IEEE Transactions on VLSI Systems*, 24(3):846–857, 2016.

[56] P. Panda, A. Sengupta, S. S. Sarwar, G. Srinivasan, S. Venkataramani, A. Raghunathan, and K. Roy. Cross-layer approximations for neuromorphic computing: From devices to circuits and systems. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*, pages 1–6, 2016.

[57] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proc. BMVC*, 2014.

[58] W. Yu, K. Yang, Y. Bai, T. Xiao, H. Yao, and R. Yong. Visualizing and comparing AlexNet and VGG using deconvolutional layers. In *Proceedings of the International Conference on Machine Learning*, 2016.

[59] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision*, Oct 2017.

[60] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the ACM International Symposium on Computer Architecture*, pages 267–278, June 2016.

[61] T. H. Corman, C. E. Leiserson, R. A. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Boston, MA, 3 edition, 2009.

[62] M. M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica*, 18(1):145–163, May 1997.

[63] D. D. Lin, S. S. Talathi, and V. S. Annapureddy. Fixed point quantization of deep convolutional networks. In *Proceedings of the International Conference on Machine Learning*, pages 2849–2858, 2016.

[64] M. Horowitz. Computing's energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 10–14, Feb 2014.

[65] M. Stein. Large sample properties of simulations using latin hypercube sampling. *Technometrics*, 29(2):143–151, 1987.

[66] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. Accessed May 30, 2019.

[67] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[68] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678, 2014.

[69] S. D. Manasi, F. S. Snigdha, and S. S. Sapatnekar. NeuPart: Using analytical models to drive energy-efficient partitioning of CNN computations on cloud-connected mobile clients, 2019. arXiv preprint arXiv:1905.05011.

[70] CNNergy: An analytical CNN energy model. `https://github.com/manasiumn37/CNNergy/`. Accessed May 30, 2019.

Appendix A

Formulation of New Error Budget

A new error budget, $\sigma^2_{m_i,NB}$, for nodes $m_i \in \mathcal{M}_{j,d}$, obtained from (4.20) is combined with (4.18) to get:

$$\sigma^2_{m_i,NB} = \sigma^2_m(\alpha_{m_i}) + \sum_{m_k \in \mathcal{Y}_{m_i}} \chi_{m_i}[m_k] \cdot (1 - \eta_{m_k}) \cdot \sigma^2_m(\alpha_{m_k}) \tag{A.1}$$

The value of $\sigma^2_{m_i,NB}$ can be different for each output x_j. Since the error variance constraint at each output must be satisfied, $\sigma^2_{m_i,NB}$ must be chosen in such a way that $\sigma^2_{x_j,dyn} \leq \sigma^2_{x_j,stat}, \forall x_j$. In other words, the value of $\chi_{m_i}[m_k]$ must be chosen so that the redistributed slack on a path to any output x_j does not exceed $\sigma^2_{x_j,stat}$. This is achieved by setting

$$\chi_{m_i}[m_k] = \min_{\text{all outputs } j} \left[S^2_{k,j}/S^2_{i,j}\right] \tag{A.2}$$

Additionally, the computation of (A.1) can be further simplified. In practice, for all $m_k \in \mathcal{Y}_{m_i}$, η_{m_k} have very similar values. Assuming each such term as η_{m_i}, we get:

$$\begin{aligned} \sigma^2_{m_i,NB} &\approx \sigma^2_m(\alpha_{m_i}) + (1 - \eta_{m_i}) \times \sum_{m_k \in \mathcal{Y}_{m_i}} \chi_{m_i}[m_k] \cdot \sigma^2_m(\alpha_{m_k}) \\ &= \tilde{\mathcal{C}}_{m_i} - \tilde{\mathcal{D}}_{m_i} \times \sigma^2_{m,true}(\alpha_{m_i}) \qquad \text{(A.3)} \\ \text{where} \quad \tilde{\mathcal{C}}_{m_i} &= \sigma^2_m(\alpha_{m_i}) + \sum_{m_k \in \mathcal{Y}_{m_i}} \chi_{m_i}[m_k] \cdot \sigma^2_m(\alpha_{m_k}), \\ \tilde{\mathcal{D}}_{m_i} &= \sum_{m_k \in \mathcal{Y}_{m_i}} \chi_{m_i}[m_k] \cdot \sigma^2_m(\alpha_{m_k})/\sigma^2_m(\alpha_{m_i}) \end{aligned}$$

The evaluation of (A.3) requires the computation of $\sigma^2_{m,true}(\alpha_{m_i})$, $\tilde{\mathcal{C}}_{m_i}$, and $\tilde{\mathcal{D}}_{m_i}$. The true error variance, $\sigma^2_{m,true}(\alpha_{m_i})$, is image-dependent and is obtained using a sampling procedure using (4.4). On the other hand, both $\tilde{C}_{m_i}$ and $\tilde{\mathcal{D}}_{m_i}$ are constant for node m_i for a specific error budget, and are determined using $\sigma^2_m(\alpha_{m_i})$ (obtained from (3.2) using α_{m_i} from the solution of the static optimization problem, (3.12)), $\mathcal{Y}_{m_i}$, and χ_{m_i}.

Appendix B

A Goodness Metric for Dynamic Node Selection

As a stepping stone to defining ω_{PB}, we introduce a total error variance metric, $\sigma^2_{m_i,T}$, for multiplier m_i. If a dynamic node m_i uses z_i more approximate bits beyond static approximation, the error variance at output x_j is:

$$\sigma^2_{x_j,m_i} = S^2_{ij} \times \sigma^2_m(\alpha_{m_i} + z_i) \tag{B.1}$$

The total error variance over all eight outputs due to m_i is:

$$\sigma^2_{m_i,T} = \sum_{j=0}^{7} \sigma^2_{x_j,m_i} = \sigma^2_m(\alpha_{m_i} + z_i) \sum_{j=0}^{7} S^2_{ij} \tag{B.2}$$

The dynamic error variance sensitivity, ω_i, is the sensitivity of $\sigma^2_{m_i,T}$ to z_i for a unit increase in approximation as:

$$\omega_i = \left.\frac{d\sigma^2_{m_i,T}}{dz_i}\right|_{z_i=1} = \ln(4) \times \sigma^2_m(\alpha_i + 1) \times \sum_{j=0}^{7} S^2_{ij} \tag{B.3}$$

where the last expression follows from (3.2) and (B.2). To determine whether a node is a good candidate for dynamic reconfiguration, its ω_i should be sufficiently small.

We define per-bit error, ω_{PB}, as the average value of ω_i over the elements of Λ. This serves as a goodness metric for selecting dynamic nodes for inclusion in Λ.

Appendix C

Detailed Explanation for the Choice of Parameter α_4

The inter-layer relationship of cluster preparation threshold $T_{3,D}$ is modulated by the parameter α_4. It is observed that the product term of $T_{3,D}$ in Eq. (5.13) increases faster as we move to more shallower layers and the growth rate of the term depends on the parameter α_4. The shallowest layer where we implement SeFAct is D_{min}. Hence, we impose the following condition to normalize $T_{3,D_{min}}$ in the interval (0,1]:

$$T_{3,D_{min}} = \alpha_2 \times \exp\{\alpha_3 \left(T_{1,D_{min}} - T^{max}_{1,N_L-2}\right)\} \prod_{d=D_{min}+1}^{N_L-2} T_{1,d}^{-\alpha_4} \leq 1 \tag{C.1}$$

As described to Section 5.4.3, the exponential term $\alpha_2 \times \exp\left\{\alpha_3 \left(T_{1,D_{min}} - T^{max}_{1,N_L-2}\right)\right\}$ is normalized to lie in the interval $(0, 1]$. Therefore, we can assume:

$$\prod_{d=D_{min}+1}^{N_L-2} \tau^{-\alpha_4} \times \left[\frac{d}{N_L-2}\right]^{-\alpha_0\alpha_4} = C, \quad \text{from (5.11)} \tag{C.2}$$

$$-\alpha_4 \left[(N_{SeFAct}-1)\log\tau + \sum_{d=D_{min}+1}^{N_L-2} \log\left[\frac{d}{N_L-2}\right]^{\alpha_0}\right] = \log C \tag{C.3}$$

$$\text{where } C = \left[\alpha_2 \times \exp\left\{\alpha_3 \left(T_{1,D_{min}} - T^{max}_{1,N_L-2}\right)\right\}\right]^{-1} \geq 1 \tag{C.4}$$

where N_{SeFAct} is the number of layers over which we implement SeFAct.

According to Table 5.5, $\alpha_4 \geq 0$. We can see from Eq. (C.2), $\tau^{-\alpha_4}$ dominates when τ is small. Hence, to satisfy (C.2), we must consider the worst-case scenario where $\tau = \tau_{min}$. The product term in Eq. (C.3) is also dependent on the number of SeFAct layers, $N_{SeFAct} = N_L - D_{min} - 1$. As D_{min} is generally implemented beyond the mid-depth of the network, $(N_{SeFAct} - 1)\log\tau_{min} \gg \sum_{d=D_{min}+1}^{N_L-2} \log\left[\frac{d}{N_L-2}\right]^{\alpha_0}$. The simplified equation becomes:

$$\alpha_4 = \frac{k_2}{(N_{SeFAct} - 1)}, \qquad k_2 = -\frac{\log C}{\log \tau_{min}} \tag{C.5}$$

Empirically, we set $1.25 \leq C \leq 2.5$ which results in $0.05 \leq k_2 \leq 0.40$ for $\tau_{min} = 0.10$.

Appendix D

Optimal α Parameters for Various Networks

Table D.1: Optimal parameters α_{opt} for SeFAct implementation in various accuracy interval.

Network	Accuracy interval	$\alpha_{opt} = \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7\}$
LeNet	{90.00,96.83}	{1.47 0.61 0.60 0.81 0.24 0.38 0.44 0.16}
	{96.83,98.35}	{1.51 0.64 0.39 0.76 0.32 0.15 0.56 0.19}
GoogLeNet	{70.00,75.18}	{0.77 0.57 0.51 1.18 0.03 0.32 0.21 0.02}
	{75.18,83.82}	{1.31 0.65 0.32 1.12 0.04 0.19 0.33 0.05}
	{83.82,85.93}	{0.75 0.63 0.40 1.16 0.06 0.49 0.37 0.04}
	{85.93,87.27}	{0.65 0.65 0.59 1.10 0.03 0.52 0.44 0.01}
	{87.27,89.00}	{0.95 0.56 0.67 0.98 0.01 0.48 0.60 0.02}
AlexNet	{60.00,75.41}	{0.81 0.63 0.13 2.04 0.16 0.66 0.77 0.12}
	{75.41,77.95}	{0.69 0.62 0.17 1.97 0.13 0.35 0.49 0.08}