Hardware-Efficient Approximate Arithmetic Circuits for Deep Learning and Other Computation-Intensive Applications

by

Mohammad Saeed Ansari

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Integrated Circuits and Systems

Department of Electrical and Computer Engineering

University of Alberta

© Mohammad Saeed Ansari, 2019

Abstract

Approximate computing (AC) is an emerging paradigm that leverages the inherent error tolerance of many applications—such as image recognition, multimedia processing, and machine learning (ML)—to allow some accuracy to be traded off to save energy consumption. AC techniques can be applied at both the circuit and/or architecture levels, possibly in coordination with software-level techniques.

Multiplication is one of the most resource- and power-hungry operations in many error-tolerant computing applications, such as image processing, neural networks (NN), and digital signal processing (DSP). In this research project, we focus on the design and implementation of hardware-efficient approximate computing circuits, aiming to simplify the multiplication operation and/or to reduce the number of required multiplications.

Two 4×4 approximate multiplier designs are proposed in which approximation is employed in the partial product reduction tree, the most expensive part of the design of a multiplier. The two proposed designs are then used to construct larger approximate multipliers.

Multiplication is the computational bottleneck in NNs. For the first time, we attempt to find the critical features in an approximate multiplier that make it superior to others for use in a NN. Inspired by the insight that adding small amounts of noise can improve the performance of NNs, we replaced the exact multipliers in two representative NNs with 600 approximate multipliers and then experimentally measured the effect on classification accuracy. In-

terestingly, some approximate multipliers improved the performance of NNs. Insight into which features of an approximate multiplier make it superior to others in the NN applications was gained by training a statistical predictor that anticipates how well a given approximate multiplier is likely to work in a NN application.

In the logarithmic number system (LNS) the multiplication operation is converted into simple shift and addition operations. We have proposed a novel exact leading-one detector (LOD) to speed up the calculation of the base-2 logarithm of the input operands to a logarithmic multiplier. In addition, since the logarithmic multipliers that use LODs always underestimate the actual multiplication product, a nearest-one detector (NOD) is proposed for a logarithmic multiplier that has a double-sided error distribution. Additionally, a logarithmic squaring circuit is proposed that uses a linear approximation for calculating the base-2 logarithm of the input operand.

Finally, we investigate the design of multiply-accumulate (MAC) units. An approximate logarithmic MAC (LMAC) unit is proposed for the first time. Furthermore, a soft-dropping low-power (SDLP) architecture is specifically designed for convolutional neural networks (CNNs) that, unlike the existing accelerators that simplify the multiplication/addition operations, reduces the number of required multiplications. The SDLP takes advantage of the spatial dependence between the input image pixels and skips some of the multiplications during the convolution operation and, thereby, reduces the energy consumption of the CNN inference calculation.

Preface

This dissertation presents the original work in the field of approximate computing (AC) by Mohammad Saeed Ansari.

In Chapter 3, we propose two 4×4 low-power approximate multipliers using encoded partial products and approximate compressors. These two multipliers are then used to build larger multipliers. This work has been published as M. S. Ansari, H. Jiang, B. F. Cockburn, and J. Han, "Low-Power Approximate Multipliers Using Encoded Partial Products and Approximate Compressors," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, Vol. 8, No. 3. pp. 404-416. I developed the circuit design and H. Jiang provided the hardware description language (HDL) code for some of the existing designs in the literature for comparison purposes. Dr. J. Han and Dr. B. F. Cockburn provided technical suggestions and revised the manuscript.

The use of approximate multipliers in neural networks (NNs) is investigated in Chapter 4, by replacing the exact multipliers in two NN benchmarks and evaluating the resulting NN's classification accuracy. A statistical analysis is then carried out to identify the critical features in an approximate multiplier that tend to improve its performance in NNs. This work has been accepted for publication as M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han, "Improving the Accuracy and Hardware Efficiency of Neural Networks Using Approximate Multipliers," *IEEE Transactions on Very Large Scale (VLSI) Systems*. I provided the HDL code for some approximate multipliers. V. Mrazek added my codes to his and evaluated the performance of all of the approximate multipliers in two NN benchmarks. Based on his results, I ran some statistical analysis to identify the critical features. Finally, I developed the classifiers that anticipate how well an approximate multiplier

would work in a NN. Drs. B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han provided technical suggestions and improved the flow of the manuscript by their comments.

Chapter 5 presents a new leading-one detector (LOD) design, which has been submitted *IET Computers and Digital Techniques* as M. S. Ansari, S. Gandhi, B. F. Cockburn, and J. Han,"Approximate Leading One Detector Design for a Hardware-Efficient Mitchell Multiplier". I developed the main idea and did the software level simulations and S. Gandhi helped with the HDL coding. Drs. J. Han and B. F. Cockburn provided constructive suggestions on improving the quality of the manuscript. An improved logarithmic multiplier (ILM) is also proposed in this chapter that uses a nearest-one detector instead of the conventional LOD. This work has been submitted to *IEEE Transactions on Computers* as M. S. Ansari, B. F. Cockburn, and J. Han,"An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing". I developed the main idea and performed the required simulations. Drs. J. Han and B. F. Cockburn contributed by providing technical suggestions that significantly improved the quality of the manuscript. Finally, a low-error squaring function (LESF) is proposed in this chapter. This work has been submitted to *IEEE Transactions on Emerging Topics in Computing* as M. S. Ansari, B. F. Cockburn, and J. Han, "Low-Power Approximate Logarithmic Squaring Circuit Design for DSP Applications". Drs. Han and Cockburn provided suggestions to the research and helped revising the manuscript.

A logarithmic multiply-accumulate (MAC) unit is proposed in Chapter 6. This work has been submitted to *IEEE Transactions on Very Large Scale (VLSI) Systems* as M. S. Ansari, B. F. Cockburn, and J. Han, "Design of a Fast and Energy-Efficient Approximate Logarithmic Multiply-Accumulate Unit". I developed the idea and ran all the required simulations. Drs. Han and Cockburn provided technical suggestions to improve the quality of the manuscript. A soft-dropping low-power (SDLP) accelerator is also presented for convolutional neural networks (CNNs). This work is drafted as M. S. Ansari, B. F. Cockburn, and J. Han, "Approximate Accelerators for CNN-based Image Classifiers that Rely on Pixel Spatial Dependence". I carried out all of the simu-

lations and Drs. Han and Cockburn provided valuable suggestions to improve the structure as well as the technical content of the manuscript.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Research Plan and Objectives 3
1.3 Contributions 4
1.4 Dissertation Outline 5

2 Computation-Intensive Applications and Approximate Arithmetic Circuits **7**
2.1 Computation-intensive applications 7
2.1.1 Image and digital signal processing 7
2.1.2 Multiple-input multiple-output (MIMO) systems 9
2.1.3 Neural networks 11
2.2 Approximate arithmetic for computation-intensive applications 15
2.2.1 Approximate adders 15
2.2.2 Approximate multipliers 16

3 Low-power approximate multipliers using encoded partial products and approximate compressors **21**
3.1 Proposed Multiplier Designs 22
3.1.1 Modified approximate 4:2 compressor 22
3.1.2 Two approximate 4×4 multipliers 25
3.1.3 Scaling up to larger multipliers 27
3.1.4 Extension to signed Booth multipliers 31
3.2 Performance Evaluation 31
3.2.1 Accuracy analysis 31
3.2.2 Hardware analysis 33
3.3 Example Applications 34
3.3.1 Image sharpening 34
3.3.2 JPEG image compression 36
3.3.3 Multiple-input multiple-output wireless systems 36
3.4 Summary 40

4 Improving the Accuracy and Hardware Efficiency of Neural Networks Using Approximate Multipliers **42**
4.1 Evaluation of Approximate Multipliers in Neural Networks . . 43
4.1.1 Application-independent metrics 43
4.1.2 Application-dependent metrics 45
4.1.3 Overfitting 46
4.2 Critical Features of Multipliers for NNs 49
4.2.1 Feature selection 50
4.2.2 Training the classifier 52
4.3 Error and Hardware Analysis of Approximate Multipliers . . . 57
4.3.1 Error analysis 57

4.3.2 Hardware analysis 57
4.4 Recommended Approximate Multipliers 60
4.5 Summary 63

5 Logarithmic Multiplier and Squaring Circuits 65
5.1 Fast and Low-Power Leading-One Detector for More Energy-Efficient Logarithmic Multipliers 65
5.1.1 The proposed LOD design 66
5.1.2 Hardware analysis of the proposed LOD 67
5.2 An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing 68
5.2.1 Proposed approximation approach 68
5.2.2 Improved logarithmic multiplier design 70
5.2.3 Performance evaluation of the ILM 75
5.2.4 Example application: neural networks 79
5.3 Low-Power Approximate Logarithmic Squaring Circuit Design for DSP Applications 83
5.3.1 Proposed squaring function 83
5.3.2 Performance evaluation 88
5.3.3 Example application: square law detector 91
5.4 Summary 92

6 Logarithmic Multiply-Accumulate Unit and Accelerators 94
6.1 Design of a Fast and Energy-Efficient Approximate Logarithmic MAC Unit 95
6.1.1 Proposed logarithmic MAC (LMAC) 95
6.1.2 Evaluation of the LMAC 100
6.1.3 Example application: image sharpening 102
6.2 Approximate Accelerators for CNN-based Image Classifiers that Rely on Pixel Spatial Dependence 103
6.2.1 Proposed approximate accelerators 106
6.2.2 Evaluation of the SDLP Accelerators 113
6.3 Summary 116

7 Conclusions and Future Work 118
7.1 Conclusions 118
7.2 Future work 120

References 121

Appendix A 131
A.1 Proof of (5.3) 131
A.2 Proof of (5.4) 131

List of Tables

3.1 Truth table of the proposed approximate compressor. 21
3.2 Truth table of the proposed approximate compressor. 23
3.3 Truth table for the Stage 2 compressor. 24
3.4 Truth table for the Stage 3 compressor. 25
3.5 Truth table for the Stage 4 compressor. 27
3.6 Using $M1$ and $M2$ to construct 8×8, 16×16, and 32×32 designs. 30
3.7 Accuracy comparison for 16×16 and 8×8 approximate multipliers. 32
3.8 Accuracy comparison for 8×8 Radix-4 Booth multipliers. . . . 33
3.9 Hardware comparison for 8×8 Radix-4 Booth multipliers. . . . 34
3.10 Decompressed image quality comparison, PSNR and SSIM metrics for JPEG compression. 37
3.11 Required increase in the number of iterations to get to a desired BER at a given SNR level. 40

4.1 Considered features of the output error function. 44
4.2 Ranking of the error function features. 52
4.3 Feature combinations that give the highest multiplier classification accuracy. 54
4.4 Classification accuracy of AlexNet on the ImageNet LSVRC-2010 dataset. 56
4.5 Hardware characteristics of the five best approximate multipliers. 62
4.6 Error characteristics of the five best approximate multipliers. . 62
4.7 Hardware characteristics of an artificial neuron implemented using recommended approximate multipliers. 63

5.1 Hardware metrics of five different logarithmic squaring functions. 67
5.2 The proposed vs. the conventional full adder. 74
5.3 Hardware comparison of the conventional and proposed full adders. 74
5.4 Error metrics of the LMs for general input distributions. . . . 77
5.5 Error metrics of the LMs for the two NN workloads. 78
5.6 Hardware metrics of the logarithmic multipliers. 78
5.7 Hardware characteristics of the artificial neuron when implemented using different types of LMs. 82
5.8 Error metrics of five different logarithmic squaring functions. . 90
5.9 Hardware metrics of five different logarithmic squaring functions. 90
5.10 Euclidean distance of five different logarithmic squaring functions. 92

6.1 Accuracy and hardware measures of the exact, logarithmic, and other approximate MAC units. 102
6.2 Average PSNR of reconstructed images by using approximate pixel values. 107
6.3 The AND-OR-Invert (AOI) logic implementation of the proposed compact adder. 113

6.4 Comparison of the hardware cost between the exact and the approximate variants of the SDLP accelerators. 116

List of Figures

2.1 Block diagram of an 8×8 MIMO system. 11
2.2 Model of an artificial neuron. 12
2.3 Structure of a feed-forward NN. 13
2.4 Feed-forward propagation in convolutional and activation layers. 15

3.1 AOI logic implementation of the proposed compressors. 26
3.2 Partial product reduction in multipliers (a) $M1$ and (b) $M2$. . 28
3.3 Building $2n \times 2n$ multipliers using $n \times n$ multipliers. 29
3.4 MRED and PDP of the approximate multipliers. 35
3.5 PSNR and SSIM values for the image sharpening application. 36
3.6 BER vs. SNR. 39

4.1 Effects of multiplier size on classification accuracy. 45
4.2 MNIST classification accuracy, training and testing with additive Gaussian noise. 49
4.3 Effect of the number of selected features on approximate multiplier classifier accuracy. 55
4.4 Neural network accuracy using the same approximate multipliers for different datasets. 58
4.5 Classification of Class 0 and Class 1 multipliers based on the most important features. 59
4.6 Hardware comparison between Class 0 and Class 1 approximate multipliers. 61

5.1 Using a 16-bit LOD to find the position of the leading one in a 32-bit number. 67
5.2 Approximation of $log_2 N$. 69
5.3 The proposed improved logarithmic multiplier (ILM) design. . 72
5.4 Error visualization in LMs. 76
5.5 Probability distribution of the trained weights for the MLP, mapped into the range of [-127, 127]. 80
5.6 Probability distribution of the trained weights for Alexnet, mapped into the range of [-127, 127]. 80
5.7 Comparison of classification accuracy of the MNIST and CIFAR-10 datasets with logarithmic multipliers. 81
5.8 Signed relative error for the Mitchell and the LESF squaring circuits. 85
5.9 Architecture of the low-error squaring function LESF. 86
5.10 Accuracy of the LESF, with respect to the MRED, vs. the constant value in (5.13). 89
5.11 Comparison of the demodulated signal $m'(t)$ with exact and logarithmic squaring functions. 92

6.1 Distribution of $exp = |e_1 - e_2|$. 97

6.2 Comparison of the outputs of the exact and the LMAC in (6.9). 98
6.3 Block diagram of the LMAC unit. 101
6.4 Sharpened images using (a) exact MAC, (b) LMAC, (c) CFPU, and (4) Trun. 103
6.5 Inputs and outputs to a convolutional layer followed by an average pooling layer. 108
6.6 An example of the SDLP approximate accelerator for filter size of 5×5. 109
6.7 Architecture of the employed CNN. 114
6.8 Effects of the SDLP accelerator on the classification accuracy on CIFAR-10 dataset. 115

List of Abbreviations

AC approximate computing

ACA almost-correct adder

ACM approximate compressor-based multiplier

AE average error

AED absolute error difference

AF activation function

AM amplitude modulation

AOI AND-OR-Invert

BAM broken-array multiplier

BER bit error rate

CNN convolutional neural network

CPU central processing unit

DCT discrete cosine transform

DSP digital signal processing

ED error difference

ER error rate

ESA equal segmentation adder

ETM error-tolerant multiplier

FIR finite impulse response

FM feature map

FP floating point

GPU graphics processing unit

ICM inaccurate multiplier

ILM improved logarithmic multiplier

LDPC low-density parity-check

LESF low-error squaring function

LMAC logarithmic multiply-accumulate

LNS logarithmic number system

LOA lower-part-OR adder

LOD leading-one detector

LSB least significant bit

LUT look-up table

MAC multiply-accumulate

MI mutual information

ML machine learning

MMSE minimum mean squared error

MRED mean relative error distance

MSB most significant bit

MSE mean square error

NMED normalized mean error distance

NN neural network

NOD nearest-one detector

PDP power-delay product

PE priority encoder

PP partial product

PVT process, voltage and temperature

QF quality factor

RED relative error difference

RFE recursive feature elimination

RMS root mean square

SDLP soft-dropping low-power

SISO single-input single-output

SNR signal-to-noise ratio

SOA set-to-one adder

SSIM structural similarity index

TAM truncated approximate multiplier

UDM under-designed multiplier

Chapter 1

Introduction

1.1 Motivation

The demand for high performance and power efficiency as well as the error-resiliency feature of many applications—such as machine learning (ML) and digital signal processing (DSP)—has motivated the development of approximate computing [1]. The primary purpose of this research project is to propose high-performance computing platforms for error-resilient computation-intensive applications, where approximate computing (AC) is applicable.

The increasing energy consumption of computer systems remains a serious and growing challenge despite recent progress in energy-efficient design techniques [2]. Today's computing systems are increasingly used to process huge amounts of data and are expected to present computationally-demanding natural human interfaces [2]. Moreover, computation-intensive applications, such as pattern recognition and data mining, have emerged that account for a significant proportion of the computational resources. Hence, more energy-efficient computing platforms are required in order to keep up with the increasing amounts of data that need to be processed.

Fortunately, many of these applications are inherently error-resilient and either fully-accurate results are not required or there is a range of acceptable results rather than a unique result [3]. On the other hand, the continuing shrinkage in the minimum feature size of semiconductor structures makes recent integrated circuits vulnerable to process, voltage and temperature (PVT) variations as well as to soft errors [4]. Therefore, the challenge of ensuring

strictly deterministic computing is increasing [5]. Conventional fault-tolerant computing techniques require redundancy at different levels of the design hierarchy that require additional hardware and can cause significant energy overhead [2].

Motivated by the above challenges, a promising computing paradigm, i.e. AC, has emerged. By leveraging the inherent error tolerance of error-resilient applications, AC allows some accuracy to be traded off to reduce the power consumption and hardware implementation cost. This research project proposes hardware-efficient computing platforms that take advantage of AC. The motivations for this research are summarized as follows:

1. There are applications in which error correcting mechanisms are already employed, such as wireless communications, that use error correcting codes. These existing error correcting mechanisms can also be used to fix many of the errors caused by approximate computing blocks.

2. AC could be beneficial for many error-resilient computation-intensive applications, such as image processing, neural networks (NNs), DSP applications, etc. In particular, significant energy and area savings can be obtained at the cost of often negligible accuracy degradation.

3. NNs are recognized as some of the most effective solutions to many challenging ML tasks [6]. AC can be utilized at different levels of abstraction to manage a NN's increasing complexity and implementation costs by introducing more energy-efficient and smaller computing platforms.

4. Multiplication is a key arithmetic operation that is highly optimized in digital processors, including central processing units (CPUs) and graphics processing units (GPUs). Hence, hardware-efficient multiplier designs could be used to significantly enhance a processor's performance.

5. Multiplication has been shown to be the most power-hungry operation in NNs [7]–[10]. Many approximate designs have been devised for multipliers (circuit-level approximation techniques); however, it is not clear which multiplier designs are the most appropriate for use in a NN.

6. Multiply-accumulate (MAC) units are widely-used in the hardware implementation of ML and DSP applications. Designing low-error, more energy-efficient, and smaller approximate MAC units could significantly improve the performance of the entire application.

1.2 Research Plan and Objectives

Based on the above observations, the main objective of this research project is to investigate the design of hardware-efficient approximate circuits for ML and other computation-intensive applications by using AC techniques. Specifically, the following research topics are addressed:

1. Improving the hardware cost-accuracy trade-offs of approximate multipliers is of great importance. For the conventional multipliers, partial product accumulation—the most power-hungry stage of the design—can be approximated. For the logarithmic multipliers, on the other hand, finding the leading one is the main bottleneck. More accurate and hardware-efficient designs are required to find the position of the most significant one in logarithmic multipliers.

2. Several approximate multipliers have been proposed in the literature to improve the hardware-efficiency of NNs. One of our objectives in this research project is to identify the critical features in an approximate multiplier that enhance the multiplier's performance in NNs. To do so, a large set of approximate multipliers needs to be investigated. The critical features can then be found by comparing the performance of approximate multipliers in standard benchmark NN workloads.

3. The squaring function is frequently used in DSP applications and, therefore, a specific circuit for calculating the squaring function can be much more hardware-efficient than a general-purpose multiplier. Since the squaring function can be converted into simple addition operations in the logarithmic number system (LNS), even more savings on the hardware cost can be expected by designing a logarithmic squaring function.

4. Logarithmic MAC unit has not been exploited in the literature while, potentially, it can be more hardware-efficient than the conventional MAC units due to the conversion of the multiplication operation into simple addition operation in the LNS. Moreover, the dependency between input data can be exploited to reduce the number of required multiplications and additions in the design of the conventional MAC units.

1.3 Contributions

The main contributions of this research project are as follows:

1. Approximate multipliers using approximate compressors

An initial approximate 4:2 compressor is proposed that introduces a relatively large error to the output. However, the number of faulty rows in the compressor's truth table is reduced by encoding its inputs using generate and propagate signals. Based on this compressor, two 4×4 approximate multipliers are designed with different accuracy-cost trade-offs. Then they are used as building blocks for scaling up to 16×16 and 32×32 multipliers.

2. Improving the performance of NNs using approximate multipliers

There is a trade-off between the accuracy and hardware cost of approximate multipliers, and there is no one best design for all applications. Thus selecting the appropriate approximate multiplier for any specific application is a complex question that typically requires careful consideration of multiple alternative designs. The critical features in an approximate multipliers that tend to make one design outperform others with respect to NN accuracy are identified based on which a statistical predictor is built that anticipates how well an approximate multiplier would work in a NN.

3. Logarithmic multiplier and squaring circuits

Data representation has a significant impact on the complexity of arithmetic operations. For example, the multiplication operation is converted into simple shift and addition operations in the LNS. The LNS is exploited to design more hardware-efficient approximate multipliers. To multiply in the LNS, leading-one detector (LOD)s are used to find the base-2 logarithm of the input

operands. Then the base-2 logarithms of the two inputs are summed up and, finally, the antilogarithm of the result is calculated, which yields the multiplication product. An exact LOD is proposed to speed up and improve the hardware efficiency of approximate logarithmic multipliers. However, LODs always underestimate the actual base-2 logarithm of the inputs. Thus a nearest-one detector (NOD) circuit is proposed that rounds both inputs to their nearest powers. Then an improved logarithmic multiplier (ILM) is built by using the proposed NOD circuit. Finally, due to the importance of the squaring function in DSP applications, a low-error squaring function (LESF) is proposed. The LESF operates in the LNS and approximates a base-2 logarithmic function with a piece-wise linear polynomial.

4. Logarithmic MAC unit and accelerators for convolutional NNs

Approximate MAC units are widely used as accelerators in many tasks, including DSP and image processing, and NN applications. The first fully-logarithmic multiply-accumulate (LMAC) unit is proposed that calculates the sum of products $AB + CD$ for the four inputs A, B, C and D which are floating point (FP) numbers. The partial products $P_1 = AB$ and $P_2 = CD$ are obtained by using simple additions in the logarithmic domain. Then a linear approximation of logarithmic addition is used to accumulate them. Additionally, a soft-dropping low-power (SDLP) architecture is proposed that accelerates the convolutional layers of CNNs . SDLP is specifically designed for CNNs and takes advantage of the spatial dependence between the input image pixels and skips some of the multiplications during the convolution operation and, thereby, reduces the energy consumption of the CNN's inference evaluation. Unlike the existing designs that simplify the arithmetic operations by using approximate computing blocks, SDLP reduces the number of required arithmetic operations.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows: The background on computation-intensive applications and the recent development of approximate arith-

metic circuits in these applications are reviewed in Chapter 2. In Chapter 3, an approximate multiplier is proposed and evaluated in image sharpening, the JPEG compression algorithm, and a MIMO system. The use of approximate multipliers in NNs is discussed in Chapter 4, where the critical features of an approximate multiplier that tend to indicate its better performance in NNs are identified. Chapter 5 presents the design of the proposed LOD, NOD, approximate multiplier ILM, and the squaring circuit LESF. In Chapter 6, we discuss the LMAC unit and the CNN accelerators based on the proposed SDLP approximate architecture. Finally, conclusions and a discussion of possible future work are provided in Chapter 7.

Chapter 2

Computation-Intensive Applications and Approximate Arithmetic Circuits

2.1 Computation-intensive applications

This section provides a brief background on several error-resilient computation-intensive applications and discusses the most common approximation technique used in these applications, i.e., the use of approximate multipliers.

2.1.1 Image and digital signal processing

Image sharpening

Image sharpening is a technique for increasing the sharpness of an image. This algorithm is used to overcome blurring that might be introduced by camera equipment, to draw attention to certain areas and to increase legibility.

The image sharpening algorithm computes $R(x, y) = 2I(x, y) - S(x, y)$ [11], where I is the input image, R is the sharpened image, and S is specified by:

$$S(x, y) = \frac{1}{4368} \sum_{m=-2}^{2} \sum_{n=-2}^{2} G(m + 3, n + 3) I(x - m, y - n), \tag{2.1}$$

where the spatial filter G is given by:

$$G = \begin{bmatrix} 16 & 64 & 112 & 64 & 16 \\ 64 & 256 & 416 & 256 & 64 \\ 112 & 416 & 656 & 416 & 112 \\ 64 & 256 & 416 & 256 & 64 \\ 16 & 64 & 112 & 64 & 16 \end{bmatrix} \tag{2.2}$$

The large number of multiplications in the image sharpening algorithm makes it a computation-intensive DSP application which has led to its use for evaluating approximate multipliers [12], [13]

JPEG image compression

The JPEG compression standard is widely used for saving storage space or transmission bandwidth for digital images [14]. This compression algorithm is lossy and causes image quality degradation depending on the compression quality factor (QF). QF is a scaling factor ranging from 1 (high recovered image quality) to 100 (high compression ratio at the cost of poorer image quality).

The basic idea of JPEG image compression is to reduce the data correlation by transforming it from the image plane domain into the spatial frequency domain. The human visional system is less sensitive to higher frequencies, therefore images can be compressed by suppressing their high frequency components. The image-to-spatial frequency domain transformation is done by applying the discrete cosine transform (DCT) [15].

In standard JPEG compression, the input image is divided into 8×8 pixel blocks. Then the 8×8 DCT of each 8×8 image pixel block is computed and unimportant DCT elements (corresponding to high frequencies) are discarded by multiplying the DCT coefficient matrix with a quantization matrix. The resulting matrix is then dequantized and its inverse DCT is computed to reconstruct an 8×8 image block. Finally, all of the image blocks are reassembled to form an image of the same size as the original one [12].

Several matrix multiplications during the JPEG compression algorithm make its hardware implementation a challenging task. Moreover, reasonable quality loss is acceptable during the image compression. Therefore, approxi-

mate multipliers can be used to improve the hardware efficiency of the JPEG compression algorithm.

Square law detector

A radio signal must be modulated to be able to carry information efficiently over a band-limited channel, such as audio information for broadcasting [16]. For example, amplitude modulation (AM) is a common modulation technique that is widely used in telecommunications [16].

Let $m(t)$ be any arbitrary message signal and $c(t) = A_c \cos(2\pi f_c t)$ be the carrier signal, where A_c and f_c denote the amplitude and frequency of the carrier signal, respectively. The AM signal $s(t)$ can then be calculated by:

$$s(t) = \big(1 + k_a m(t)\big) c(t). \tag{2.3}$$

where the amplitude sensitivity k_a is a constant such that $k_a m(t) << 1$ [16]. We will set $k_a m(t) = 0.01$.

Once the modulated signal $s(t)$ is calculated, it is transmitted over an ideal communication channel. At the receiver, the amplitude-modulated signal $s(t)$ can be demodulated using a square law detector [16]. Thus $s^2(t)$ can be expressed as:

$$s^2(t) = \big(1 + k_a m(t)\big)^2 \left(\frac{1 + \cos(4\pi f_c t)}{2}\right). \tag{2.4}$$

After dropping the DC terms, $s^2(t)$ is passed through a low-pass filter which removes the high-frequency term $\cos(4\pi f_c t)$ and outputs a replica of the message $m(t)$. Approximate multipliers (or squaring functions) can be used to calculate (2.4) in order to reduce the hardware costs.

2.1.2 Multiple-input multiple-output (MIMO) systems

Today, MIMO technology is being employed in wireless communications instead of the conventional single-input single-output (SISO) technology due to

its higher data bandwidth and power efficiency of MIMO over multipath fading channels [17].

In digital communication, a transmitted '1/0' could be changed to a '0/1' due to various factors, such as noise and fading. The ratio of erroneous bits to the total number of transmitted bits over a channel is called the bit error rate (BER). Channel coding is a technique where functionally dependent bits are inserted so that most of the errors that occur in data transmission over noisy communication channels can be detected and corrected.

Given the error tolerance provided by error correcting codes, computation errors in an approximate design are mixed with the errors caused by noise so that a system can recover from some of the approximation errors using error detection and correction coding. We use four coding schemes to evaluate the performance produced by the proposed approximate multipliers. The evaluation is done by using BER vs. SNR (short for signal-to-noise ratio) curves in the standard way that is used in communication engineering to illustrate the error correcting performance of codes.

We modeled an 8×8 MIMO system, see Fig. 2.1. The baseband system model for Fig. 2.1 is specified algebraically by:

$$y = Hx + N, \tag{2.5}$$

in which x is the coded user bit stream (i.e., a vector of baseband symbols), H is the channel matrix that models the interference in the channel, N models the additive white Gaussian channel noise, and y is the received vector of corrupted baseband symbols. In an 8×8 MIMO system, y, x and N are 8×1 complex matrices while H is a complex 8×8 matrix.

In the receiver block, the minimum mean squared error (MMSE) interference nulling matrix w is multiplied by the incoming signal vector y. The MMSE approach aims to find the matrix w that minimizes the criterion, E. Nulling matrix w is specified by:

$$w = [H^*H + N_0 I]^{-1} H^*, \tag{2.6}$$

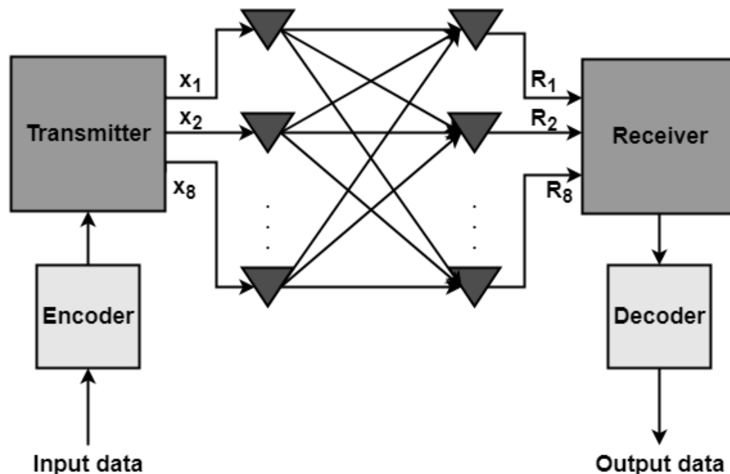


Figure 2.1: Block diagram of an 8×8 MIMO system.

where N_0 is 2× the variance of the noise at the receiver antennas and $(.)^*$ is the conjugate transpose operator [18]. The final results at each receiver can be obtained by left-multiplying (using the approximate multipliers) the incoming signal vector y by the obtained nulling matrix w.

2.1.3 Neural networks

NNs process information in an entirely different way than a conventional (von Neumann) computer [19]. Weights are adjusted in the neurons of a NN to allow the NN to perform certain computations (e.g., pattern recognition and classification on vectors or arrays of input values) [20]. Note that the neurons in a NN are arranged in several layers including an input layer, a variable number of hidden layer(s) (of the same or different types) followed by an output layer. The neurons within the same layer process inputs from the earlier layer in parallel. The outputs from the output layer are often used to signal the likelihood of membership in two or more disjoint classes.

As experience is being gained in machine learning tasks, diverse types of hidden NN layers have been proposed. The authors in [21] employed convolutional layers that function as local filters to data from the previous layers. Other common types of hidden layers are the average and max pooling layer that are used for weighted sub-sampling [22]. More recently, several application-specific layers have been proposed for image classification [23], seg-

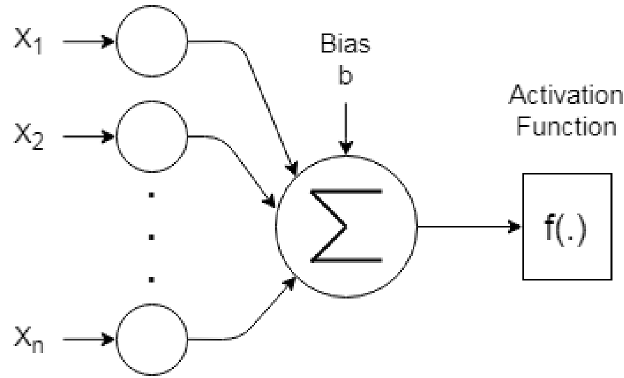


Figure 2.2: Model of an artificial neuron.

mentation [24] and speech processing [25].

Artificial neuron

Neurons are the main processing units of NNs that compute a weighted sum of their inputs and then send the result through an activation function (AF). The AF introduces non-linearity into a NN's behavior and maps the resulting output values either into either the interval (-1, 1) or (0, 1) [8]. The AF can be either a hard-limiting function (e.g., a step function) or a soft-limiting function (e.g., a sigmoid function) [26].

Fig. 2.2 shows the structure of an artificial neuron. A neuron has $n \geq 2$ inputs (depending on the network structure) and one output. Each input x_i is multiplied by its corresponding synaptic weight $w_i, i = 0, 1, ..., n$. An adder tree is then used to sum up the products. The resulting sum is then input to the AF. An external bias b is often included to increase or lower the sum that is the input to the AF [20].

Feed-forward neural networks

The two major operating modes for NNs are training and inference. The training process is usually performed infrequently and off-line and, therefore, its energy consumption is less of a concern [26]. The inference process, on the other hand, is done frequently. Although it is less computation-intensive than the training process, inference still requires significant computation for large networks. Note that a trained network can be retrained and used to perform

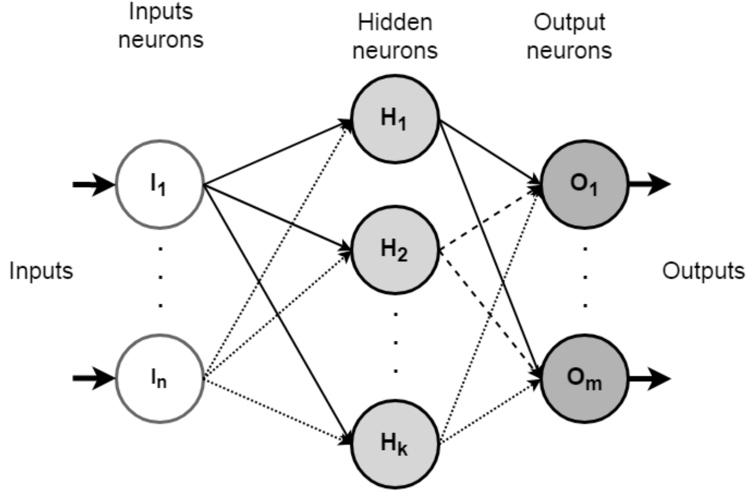


Figure 2.3: Structure of a feed-forward NN.

a different task on a different dataset. Usually only a few steps of retraining are required to fine-tune the pre-trained network for another problem. Fig. 2.3 shows a feed-forward NN with n, k, and m neurons in the input, hidden, and output layers, respectively.

Convolutional neural networks

CNNs are a class of deep neural networks, which are mainly used to analyze visual imagery [6], [27], [28]. CNNs are feed-forward neural networks consisting of a pipeline of layers. Each layer inputs a set of data, known as a feature map (FM), and produces a new set of FMs with higher-level semantics [6]. The four main computations involved in the major types of a typical CNN layers are:

1. Convolutional layer: A convolutional layer applies a set of trained convolution filters Θ to a set of input volumes X^{conv} (i.e., a color image in the case of the first convolutional layer or an output generated by previous layers in the network) and outputs a set of FMs, Y^{conv}. The computations involved in a convolutional layer are thus:

$$Y^{conv} = conv(X^{conv}, \Theta) + \beta[n] \tag{2.7}$$

where β denotes the trained bias term. Note that during convolution, the kernel Θ slides across the whole range of X^{conv}.

2. Activation layer: A convolutional layer is usually followed by an activation layer that applies a non-linear function to all of the FM's values. The most common activation function, which is also used in this work, is the rectified linear unit (ReLU) that implements $Y^{act} = \max(0, X^{act})$ [23], where X^{act} denotes the input to this layer.

3. Pooling layer: A pooling layer sub-samples the output of the convolution layer and reduces the spatial dimension by discarding irrelevant detail [29]. The intuitive reasoning behind this layer is that the exact location of a specific feature (which is extracted in the convolution layer) is not as important as its location relative to the other features [30]. The typical pooling layers are the maximum and average pooling layers, which produce almost identical results [29]. The average pooling layer, which slides over the input to this layer and outputs the average of every subregion that the filter convolves around, is used in this research study.

4. Fully-connected layer: The fully-connected layers are usually form the last few layers of a CNN. A fully-connected layer takes the output of the previous layer (i.e., the activation maps of high-level features) and determines which features most strongly correlate to a particular class.

Fig. 2.4 shows an illustrative example of the feed-forward propagation in the convolution and activation layers. The bias is omitted in Fig. 2.4 for simplicity. Parameter N in this figure indicates the number of filters.

The computational workload of a CNN inference is the result of an intensive use of the multiply-accumulate (MAC) operations. Most of these MACs occur in the convolutional layers and, therefore, convolutional layers are responsible of more than 90% of execution time during the inference [31], [32]. Thus we focus on the convolution operation in the convolutional layers.

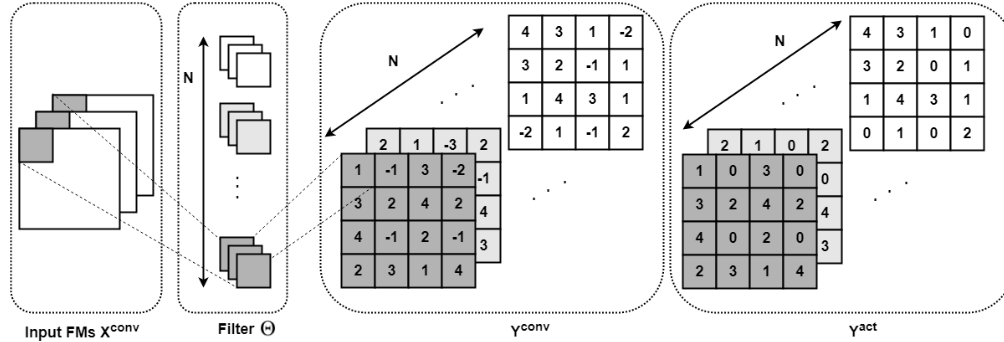


Figure 2.4: Feed-forward propagation in convolutional and activation layers.

2.2 Approximate arithmetic for computation-intensive applications

The most common arithmetic circuits are the multipliers, adders, and dividers. Dividers are not used as frequently as multipliers and adders [13], as shown in the example applications in Section 2.1. Hence, only approximate adders and multipliers are reviewed in this section.

2.2.1 Approximate adders

Several approximation techniques have been proposed to reduce the hardware cost and energy consumption of an exact adder. The three main schemes for approximating an exact adder are:

1. Predicting S_i (i.e. the ith LSB of the final result) by its previous k, where $k < i$, LSBs [33], [34]. Approximate adders that are built by using this techniques are usually referred to as speculative adders.

2. Dividing an adder into several smaller adders that operate in parallel and, thereby, accelerating the addition operation [35], [36]. Approximate adders that fall into this category are referred to as segmented adders.

3. Approximating one-bit full adders and then using them to implement a few LSBs in an accurate adder [37], [38]. More approximate LSBs usually results in more hardware savings at the cost of a lower accuracy.

The most common and widely-used approximate adders in each category are briefly summarized below.

The **almost-correct adder (ACA)** [34] is based on the speculative adder design of [33]. In an n-bit ACA, k LSBs are used to predict the carry for each sum bit. Note that the authors in [34] reduce the hardware cost in [33] by sharing some components among the sub-carry generators.

The **equal segmentation adder (ESA)** [39] divides an n-bit adder into a number of smaller k-bit sub-adders. All the sub-adders operate in parallel with fixed carry inputs. In other words, so no carry is propagated among the sub-adders. ESA is the fastest segmented adder [13].

The **lower-part-OR adder (LOA)** [38] approximates an exact adder by simply using logical OR gates for a few LSBs. The LOA also uses one logical AND gate for carry propagation of the LSBs and exact one-bit adders for the MSBs.

The **truncated adder** [13] truncates a few LSBs. This adder is most likely the simplest approximate adder that always underestimates the actual summation results. Hence, it is not the best choice in applications with successive and iterative additions.

2.2.2 Approximate multipliers

Multiplication is more resource- and power-hungry than adders and, therefore, this research project mainly focuses on the design of approximate multipliers. Note that approximate MAC units are often designed by using approximate multipliers and, thus we do not discuss them separately here in this section.

Approximate multipliers in this section are divided into two main groups: (1) conventional approximate multipliers and (2) logarithmic approximate multipliers.

Conventional multipliers

We divide the conventional approximate multiplier into two main categories: (1) deliberately-designed approximate multipliers and (2) CGP-based approximate multipliers. These two categories are discussed below.

Deliberately-designed approximate multipliers:

Deliberately-designed approximate multipliers are obtained by making carefully chosen simplifying changes in the truth table of the exact multiplier. In general, there are three ways of generating approximate multipliers [12], [13]: (1) approximation in generating the partial products, such as the under-designed multiplier (UDM) [40]; (2) approximation in the partial product tree, such as the broken-array multiplier (BAM) [38] and the error-tolerant multiplier (ETM) [41]; and (3) approximation in the accumulation of the partial products, such as the inaccurate multiplier (ICM) [42], the approximate compressor-based multiplier (ACM) [43], the approximate multiplier (AM) [44], and the truncated approximate multiplier (TAM) [45].

Here we briefly review the design of the deliberately-designed approximate multipliers.

The **under-designed multiplier (UDM)** [40] is designed based on an approximate 2×2 multiplier. This approximate 2×2 multiplier produces "111_2" instead of "1001_2" to save one output bit when both of the inputs are "11_2".

The **broken-array multiplier (BAM)** [38] omits the carry-save adders for the least significant bit (LSB) in an array multiplier in both the horizontal and vertical directions. In other words, it truncates the LSBs of the inputs to permit a smaller multiplier to be used for the remaining bits.

The **error tolerant multiplier (ETM)** [41] divides the inputs into separate LSB and most significant bit (MSB) parts that do not necessarily have equal width. Every bit position in the LSB part is checked from left to right and if at least one of the two operands is '1', checking is stopped and all of the remaining bits from that position onward are set to '1'. On the other hand, normal multiplication is performed for the MSB part.

The **imprecise compressor multiplier (ICM)** [42] uses an approximate (4:2) counter to build approximate multipliers. The approximate 4-bit multiplier is then used to construct larger multipliers.

The **approximate compressor-based multiplier (ACM)** [43] is designed by using approximate 4:2 compressors. The two proposed approximate 4:2 compressors (AC1 and AC2) are used in a Dadda multiplier with four

different schemes.

The **approximate multiplier (AM)** [44] uses a novel approximate adder that generates a sum bit and an error bit. The error of the multiplier is then alleviated by using the error bits. The truncated version of the AM multiplier is called the TAM [45].

Based on these main designs, variants were obtained by changing the configurable parameter in each design, forming a set of 100 deliberately-designed approximate multipliers. For example, removing different carry-save adders from the BAM multiplier results in different designs; also the width of the MSB and LSB parts in the ETM multiplier can be varied to yield different multipliers.

CGP-based approximate multipliers:

Unlike the deliberately-designed approximate multipliers, the CGP-based designs are generated automatically using Cartesian Genetic Programming [8]. Although several heuristic approaches have been proposed in the literature for approximating a digital circuit, we used CGP since it is intrinsically multi-objective and has been successfully used to generate other high-quality approximate circuits [46].

A candidate circuit in CGP is modeled as a two-dimensional array of programmable nodes. The nodes in this problem are the 2-input Boolean functions, i.e. AND, OR, XOR, and others. The initial population P of CGP circuits includes several designs of exact multipliers and a few circuits that are generated by performing mutations on accurate designs. Single mutations (by randomly modifying the gate function, gate input connection, and/or primary output connections) are used to generate more candidate solutions. More details are provided in [8] and [46].

Logarithmic multipliers

Let $Z = Z_n Z_{n-1} ... Z_1 Z_0$ be the n-bit binary representation of a positive integer N. Without loss of generality, let Z_k, where $k \leq n$, be the most significant '1' in Z. Hence, N can be represented as:

$$N = 2^k(1 + x), \tag{2.8}$$

where $0 \leq x < 1$.

Let A and B be the multiplicand and the multiplier, respectively. Following (2.8), once the base-2 logarithms of input operands A and B are calculated as:

$$\log_2 A = k_1 + \log_2(1 + x_1), \tag{2.9}$$

$$\log_2 B = k_2 + \log_2(1 + x_2), \tag{2.10}$$

their product can be obtained by:

$$A \times B = 2^{k_1+k_2}(1 + x_1)(1 + x_2). \tag{2.11}$$

Depending on the computation process, different values for $log_2 A$ and $log_2 B$ and, consequently, different approximate products can be obtained. For example the Mitchell algorithm uses the following approximation [47]:

$$A \times B \approx \begin{cases} 2^{k_1+k_2}(1 + x_1 + x_2), & x_1 + x_2 < 1, \\ 2^{k_1+k_2+1}(x_1 + x_2), & x_1 + x_2 \geq 1. \end{cases} \tag{2.12}$$

It was found in [48] that the average error for given k_1, k_2, $x_1 \in [0, 1)$, and $x_2 \in [0, 1)$ for the Mitchell algorithm can be expressed as:

$$E_A = -0.08333 \times 2^{k_1+k_2}. \tag{2.13}$$

Hence, an error correction term c can be added to the Mitchell algorithm to reduce the average error [48]:

$$A \times B \approx \begin{cases} 2^{k_1+k_2}(1 + x_1 + x_2 + c), & x_1 + x_2 < 1, \\ 2^{k_1+k_2+1}(x_1 + x_2 + \frac{c}{2}), & x_1 + x_2 \geq 1. \end{cases} \tag{2.14}$$

However, this modified technique increases the area and power consumption compared to the Mitchell algorithm [48].

The approximate LM in [49] uses a so-called set-to-one adder (SOA) (ALM-SOA). The set-one-adder (SOA) with k approximation bits (SOA-k) puts '1'

on the k LSBs and, therefore, the actual product is overestimated. Given the fact that the Mitchell multiplier always underestimates the actual product, using a SOA can compensate for the accuracy loss in the multiplier. This technique is used in [49] to improve the accuracy of the Mitchell multiplier with less hardware cost.

A low-power implementation of the Mitchell multiplier is proposed in [50]. As extended work, a parameter w is introduced in [51] for a customizable LM in which only the most significant w bits of the operands are taken into account. Subsequently, truncation is performed after the approximate logarithms of the operands are calculated (using the Mitchell algorithm). This differs from truncating the input operands before computing their logarithm. Due to the truncation, this multiplier is more hardware-efficient than the Mitchell multiplier. However, it is less accurate than it in terms of both mean and worst-case errors.

Unfortunately, Mitchell's method can have relatively large approximation errors [52]. Several Mitchell-based multipliers have been proposed to improve the accuracy. They usually divide the power-of-two intervals into more than one region and then apply piece-wise linear approximation within each region. Different designs differ in the number of regions and in the piece-wise linear approximation functions used in each region [53].

The approximation error in the reported Mitchell-based multipliers is always negative (i.e., the magnitude of the approximate product is smaller than the exact product) [49]. This systematic error causes problems in repetitive or iterative operations, such as matrix multiplications, since the errors do not cancel and are accumulated.

Chapter 3

Low-power approximate multipliers using encoded partial products and approximate compressors

Consider two 4-bit unsigned operands $\alpha = \sum_{i=0}^{3} \alpha_i 2^i$ and $\beta = \sum_{i=0}^{3} \beta_i 2^i$. The partial product (PP) array pp is a 4×4-bit array of the partial product bits $pp_{(i,j)} = \alpha_i.\beta_j$, where $i, j \in \{0, 1, 2, 3\}$. Table 3.1 gives all the PPs for a 4-bit multiplication and their corresponding product bits, with columns corresponding to increasing powers of two going from 2^0 at the right to 2^7 at the left.

The product is denoted by $\gamma = \sum_{k=0}^{7} \gamma_k 2^k$. The bits of γ are produced in stages going from the LSB to the MSB. According to Table 3.1, $\gamma_0 = pp_{(0,0)}$ and there is no further operation in Stage 0. In Stage 1, to generate γ_1, we can simply use a half adder that produces a sum bit γ_1 and a carry bit (c_1) for the next stage. Since the half adder circuit is already a simple design, there is no benefit to approximating it.

Table 3.1: Truth table of the proposed approximate compressor.

Stage 7	Stage 6	Stage 5	Stage 4	Stage 3	Stage 2	Stage 1	Stage 0
	$pp_{(3,3)}$	$pp_{(3,2)}$	$pp_{(3,1)}$	$pp_{(3,0)}$	$pp_{(2,0)}$	$pp_{(1,0)}$	$pp_{(0,0)}$
		$pp_{(2,3)}$	$pp_{(2,2)}$	$pp_{(2,1)}$	$pp_{(1,1)}$	$pp_{(0,1)}$	
			$pp_{(1,3)}$	$pp_{(1,2)}$	$pp_{(0,2)}$		
				$pp_{(0,3)}$			
γ_7	γ_6	γ_5	γ_4	γ_3	γ_2	γ_1	γ_0

In Stage 2, there are three *pp* terms and the carry in from the previous stage (c_1) that must be added together. Thus a 4:2 compressor is required to generate γ_2 and a carry out to the next stage. The PPs can be accumulated using a compressor circuit. An exact 4:2 compressor calculates:

$$\begin{aligned} Sum &= x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus c_{in}, \\ C_{out} &= (x_1 \oplus x_2)x_3 + (\overline{x_1 \oplus x_2})x_1, \\ Carry &= (x_1 \oplus x_2 \oplus x_3 \oplus x_4)c_{in} + (\overline{x_1 \oplus x_2 \oplus x_3 \oplus x_4})x_4. \end{aligned} \tag{3.1}$$

Note that a 4:2 compressor has four data inputs (x_1, x_2, x_3, and x_4), one carry input (c_{in}) and three outputs (Sum, C_{out} and $Carry$). The Sum output has the same weight as the four input signals while the C_{out} is used as the carry in for the next higher-order inputs and the output $Carry$ weighted like a (*pp*) bit in a one-bit-higher position. Note that outputs C_{out} and $Carry$ have the same weight.

3.1 Proposed Multiplier Designs

3.1.1 Modified approximate 4:2 compressor

Several 4:2 compressors are required to implement one 4× 4 multiplier. However, the function of an exact 4:2 compressor can be approximated to reduce the hardware cost. Ignoring C_{out} (due to its small impact on the compressor's accuracy [54]) as well as our goal to use as few gates as possible led to the approximate compressor truth table given in Table 3.2.

As shown in Table 3.2, there are five/seven incorrect values for the approximate $Carry/Sum$ outputs which correspond to an output error. To reduce this source of inaccuracy, we encode the inputs to the compressor using conventional *propagate* and *generate* signals given by:

$$\begin{aligned} P_{(i,j)} &= pp_{(i,j)} + pp_{(j,i)}, \\ G_{(i,j)} &= pp_{(i,j)} . pp_{(j,i)}, \end{aligned} \tag{3.2}$$

This encoding ensures that, although the approximate circuit may have a fairly large number of faulty output entries in the truth table, it in fact rarely produces those outputs. To see how this approach affects the compressor's accuracy, consider Stage 2 in which the following terms are added: $pp_{2,0}$, $pp_{1,1}$,

Table 3.2: Truth table of the proposed approximate compressor.

x_1	x_2	x_3	x_4	**Carry** Exact / Approximate	**Sum** Exact / Approximate
0	0	0	0	0/0 ✓	0/0 ✓
0	0	0	1	0/0 ✓	1/1 ✓
0	0	1	0	0/0 ✓	1/1 ✓
0	0	1	1	1/1 ✓	0/1 ✗
0	1	0	0	0/0 ✓	1/1 ✓
0	1	0	1	1/0 ✗	0/1 ✗
0	1	1	0	1/0 ✗	0/1 ✗
0	1	1	1	1/1 ✓	1/1 ✓
1	0	0	0	0/0 ✓	1/1 ✓
1	0	0	1	1/0 ✗	0/1 ✗
1	0	1	0	1/0 ✗	0/1 ✗
1	0	1	1	1/1 ✓	1/1 ✓
1	1	0	0	1/1 ✓	0/1 ✗
1	1	0	1	1/1 ✓	1/1 ✓
1	1	1	0	1/1 ✓	1/1 ✓
1	1	1	1	0/1 ✗	0/1 ✗
Approximate Sum $= (x_1 + x_2) + (x_3 + x_4)$					
Approximate Carry $= (x_1.x_2) + (x_3.x_4)$					

$pp_{0,2}$, and c_1. Table 3.3, where NA stands for *Not Applicable*, shows how encoding the PPs using (3.2) helps to improve the design accuracy compared to the situation in Table 3.2. Note that all possible input combinations for the 4×4 multiplier were considered ($2^4 \times 2^4 = 256$) to obtain the probability of each input combination shown in Table 3.3.

Using the proposed technique, the number of faulty $Carry/Sum$ values is reduced from 5/7 to 2/4. Note that the two approximated cases for the $Carry$ signal occur only with a small probability of 0.078 (0.0624+0.0156), see Table 3.3. It is also worth mentioning that the following combinations in Table 3.3 cannot occur, so they do not contribute to the output errors for the approximate compressor:

- (0,1) for $(pp_{(1,1)}, c_1)$: since $c_1 = pp_{(0,1)}.pp_{(1,0)} = (\alpha_0.\beta_1).(\alpha_1.\beta_0)$, c_1 = '1' means that α_0, β_1, α_1, and β_0 are '1'. Consequently, $pp_{(1,1)} = \alpha_1.\beta_1 = 1$. Hence, it is impossible to have the (0,1) combination for $(pp_{(1,1)}, c_1)$.

- (0,1,1) for (c_1,$pp_{(1,1)}$,$G_{(2,0)}$): having $c_1 = pp_{(0,1)}.pp_{(1,0)} = (\alpha_0.\beta_1).(\alpha_1.\beta_0)$=

Table 3.3: Truth table for the Stage 2 compressor.

$P_{(2,0)}$	$G_{(2,0)}$	$pp_{(1,1)}$	c_1	Carry	Sum	Probability
0	0	0	0	✓	✓	0.4218
0	0	0	1	NA	NA	0.0000
0	0	1	0	✓	✓	0.1251
0	0	1	1	✓	✗	0.0156
0	1	0	0	NA	NA	0.0000
0	1	0	1	NA	NA	0.0000
0	1	1	0	NA	NA	0.0000
0	1	1	1	NA	NA	0.0000
1	0	0	0	✓	✓	0.2814
1	0	0	1	NA	NA	0.0000
1	0	1	0	✗	✗	0.0624
1	0	1	1	✓	✓	0.0312
1	1	0	0	✓	✗	0.0468
1	1	0	1	NA	NA	0.0000
1	1	1	0	NA	NA	0.0000
1	1	1	1	✗	✗	0.0156
Approximate Sum $= \boldsymbol{x_1 + x_3}$						
Approximate Carry $= \boldsymbol{x_2 + x_4}$						

‘0’ and $pp_{(1,1)} = \alpha_1.\beta_1=$ ‘1’ means at least one of a_0 or b_0 is ‘0’, which leads to $G_{(2,0)} = pp_{(2,0)}.pp_{(0,2)} = (\alpha_2.\beta_0).(\alpha_0.\beta_2)=$ ‘0’. Thus, the (0,1,1) combination for (c_1,$pp_{(1,1)}$,$G_{(2,0)}$) is not possible.

- (0,1) for ($P_{(2,0)}$,$G_{(2,0)}$): $G_{(2,0)} = pp_{(2,0)}.pp_{(0,2)}=$ ‘1’ means that both $pp_{(2,0)}$ and $pp_{(0,2)}$ are ‘1’. Therefore, $P_{(2,0)} = pp_{(2,0)} + pp_{(0,2)}=$ ‘1’ and so we cannot have the (0,1) combination for $(P_{(2,0)}, G_{(2,0)})$.

To compute bit γ_3 of the product in Stage 3, the four $pp_{(i,j)}$ terms and the carry c_2 from Stage 2 should be added and, therefore, a 5:2 compressor is required. Since the proposed compressor is a 4:2 design, we can merge two of these five signals to reduce them to four, as specified by:

$$\begin{gathered} x_1 = c_2, \\ x_2 = G_{(3,0)} + G_{(3,0)}, \\ x_3 = P_{(2,1)}, \\ x_4 = P_{(3,0)}. \end{gathered} \tag{3.3}$$

where x_1, x_2, x_3 and x_4 are the inputs to the compressor that generates γ_3 and a carry out (c_3) for the next stage. Table 3.4 shows how altering the partial

Table 3.4: Truth table for the Stage 3 compressor.

c_2	$G_{(3,0)} + G_{(3,0)}$	$P_{(2,1)}$	$P_{(3,0)}$	Carry	Sum	Probability
0	0	0	0	✓	✓	0.3087
0	0	0	1	✓	✓	0.1953
0	0	1	0	✓	✓	0.1952
0	0	1	1	✓	✗	0.1092
0	1	0	0	NA	NA	0.0000
0	1	0	1	✗	✗	0.0273
0	1	1	0	✗	✗	0.0315
0	1	1	1	✓	✓	0.0233
1	0	0	0	✓	✓	0.0079
1	0	0	1	✗	✗	0.0156
1	0	1	0	✗	✗	0.0158
1	0	1	1	✓	✓	0.0314
1	1	0	0	NA	NA	0.0000
1	1	0	1	✓	✓	0.0079
1	1	1	0	✓	✓	0.0038
1	1	1	1	✗	✗	0.0273
$\boldsymbol{Approximate\ Sum = x_1 + x_3 + x_4}$						
$\boldsymbol{Approximate\ Carry = (x_1.x_2) + (x_3.x_4)}$						

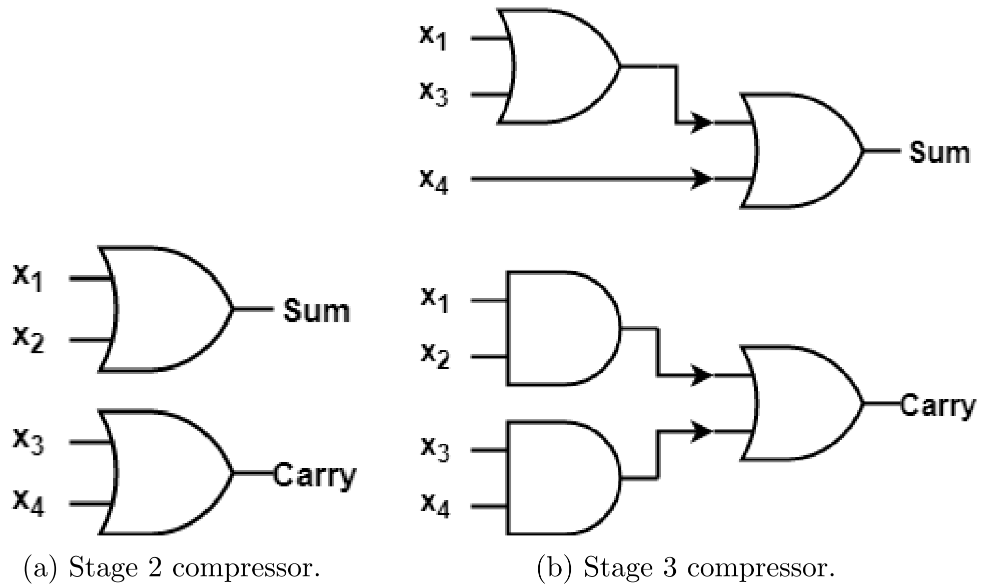
products affects the compressor's truth table in Stage 3. As for Stage 2, the design can be simplified by using Boolean algebra, as given in Table 3.4.

Similar calculations are done for Stage 4 and the results are provided in Table 3.5. Using the same argument as in Table 3.3, when $G_{(3,1)}$ = '1', $P_{(3,1)}$ must be '1'. Hence, the entries that do not follow this are NA entries.

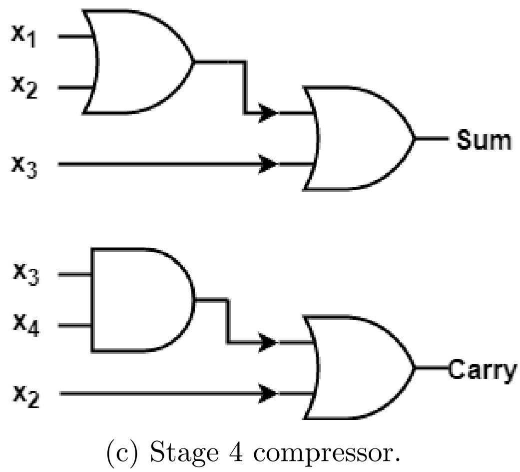
The AND-OR-Invert (AOI) logic implementations of the three proposed compressors are provided in Fig. 3.1

3.1.2 Two approximate 4×4 multipliers

The two proposed 4×4 approximate multipliers are referred to as: (1) $M1$, which considers the carry from the previous stage (c_4) and uses an exact full-adder to add pp terms $pp_{(3,2)}$, $pp_{(2,3)}$, and c_4; and (2) $M2$, which ignores c_4 and uses an exact half adder to add $pp_{(3,2)}$ and $pp_{(2,3)}$. Note that $M1$ is more accurate than $M2$. By ignoring c_4, multiplier $M2$ breaks the longest path (that is, the carry propagation path), which is a common technique to reduce the circuit's latency [55]. Fig. 3.2 summarizes the two designs by showing the



(a) Stage 2 compressor. (b) Stage 3 compressor.



(c) Stage 4 compressor.

Figure 3.1: AOI logic implementation of the proposed compressors.

Table 3.5: Truth table for the Stage 4 compressor.

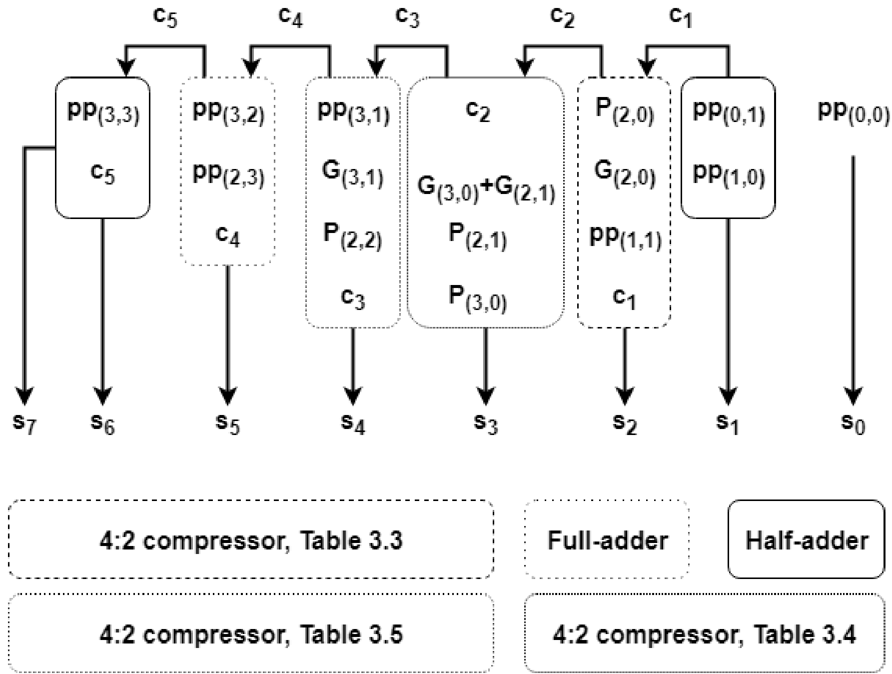
c_2	$G_{(3,0)}+G_{(3,0)}$	$P_{(2,1)}$	$P_{(3,0)}$	**Carry**	**Sum**	**Probability**
0	0	0	0	✓	✓	0.3087
0	0	0	1	✓	✓	0.1953
0	0	1	0	✓	✓	0.1952
0	0	1	1	✓	✗	0.1092
0	1	0	0	NA	NA	0.0000
0	1	0	1	✗	✗	0.0273
0	1	1	0	✗	✗	0.0315
0	1	1	1	✓	✓	0.0233
1	0	0	0	✓	✓	0.0079
1	0	0	1	✗	✗	0.0156
1	0	1	0	✗	✗	0.0158
1	0	1	1	✓	✓	0.0314
1	1	0	0	NA	NA	0.0000
1	1	0	1	✓	✓	0.0079
1	1	1	0	✓	✓	0.0038
1	1	1	1	✗	✗	0.0273
$\boldsymbol{Approximate\ Sum = x_1 + x_2 + x_3}$						
$\boldsymbol{Approximate\ Carry = x_2 + (x_3.x_4)}$						

employed blocks for reducing the partial products.

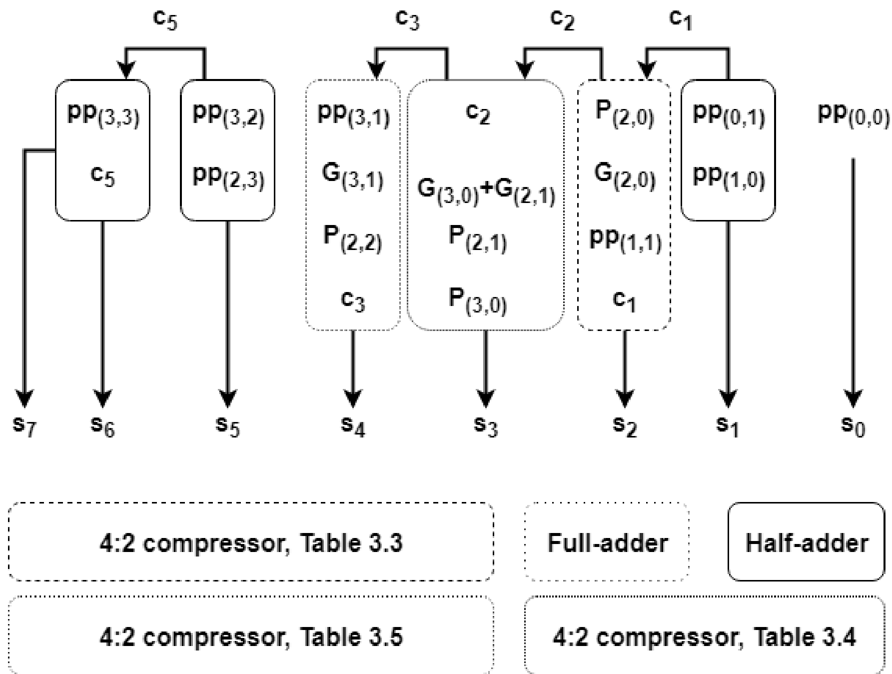
3.1.3 Scaling up to larger multipliers

To construct larger approximate multipliers, the two proposed 4×4 multipliers can be combined in an array structure. For instance, to construct an 8×8 multiplier using a 4×4 design, the two 8-bit operands A and B are partitioned into two 4-bit nibbles, namely α_H and α_L for A and β_H and β_L for B, where α_H and β_H are the 4 MSBs and α_L and β_L indicate the 4 LSBs of A and B, respectively. Each two of these four nibbles are then multiplied using 4×4 multipliers and the partial products are then shifted (based on the nibble's relative weight) and added together (using a fast Wallace tree architecture) to produce the final multiplication product. Building $2n \times 2n$ multipliers using $n \times n$ multipliers is specified in Fig. 3.3 and is described by:

$$\begin{aligned}\gamma = \alpha \times \beta &= (2^n \times \alpha_H + \alpha_L) \times (2^n \times \beta_H + \beta_L) \\ &= 2^{2n} \times (\alpha_H \times \beta_H) + 2^n \times (\alpha_H \times \beta_L + \alpha_L \times \beta_H) + \alpha_L \times \beta_L \\ &= 2^{2n} \times P_1 + 2^n \times (P_2 + P_3) + P_4.\end{aligned} \tag{3.4}$$



(a) Multiplier $M1$



(b) Multiplier $M2$

Figure 3.2: Partial product reduction in multipliers (a) $M1$ and (b) $M2$.

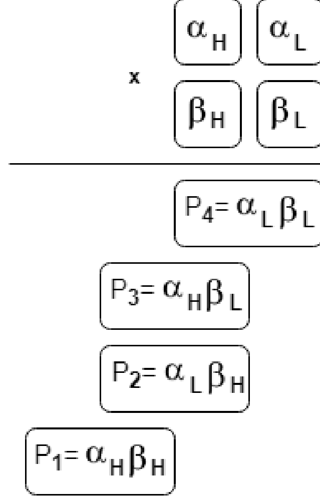


Figure 3.3: Building $2n \times 2n$ multipliers using $n \times n$ multipliers.

Note that each partial product P_i, where $i \in \{1, 2, 3, 4\}$, in (3.4) is generated using an $n \times n$ multiplier and multiplications by 2^{2n} and 2^n are simply done by $2n$-bit and n-bit left shifts, respectively.

Given that P_4 is the least and P_1 is the most significant partial products, whereas P_2 and P_3 are equivalently significant, multipliers with different accuracies can be designed with different configurations. We propose six 8×8 approximate multipliers, three of which, i.e. M8-1, M8-3, and M8-5, use $M1$ and the other three use $M2$ as their main building block. Table 3.6 shows how each of these six 8×8 multipliers is constructed.

According to Table 3.6, M8-1 and M8-2 use 4×4 approximate multipliers $M1$ and $M2$, respectively, to generate all four partial products from P_1 to P_4. M8-3 and M8-4 are more accurate designs in which the most significant partial product, P_1, is generated using an exact 4×4 multiplier, and $M1$ and $M2$ are used respectively to generate P_2, P_3, and P_4. M8-5 and M8-6 are the most accurate designs in which only the least significant partial product, P_4, uses approximate multipliers $M1$ and $M2$, respectively, and the other three partial products are generated using exact multipliers.

Note that 16×16 and 32×32 approximate multipliers can be constructed by considering (3.4). We scaled up the six 8×8 designs in Table 3.6 to form six 16×16 and 32×32 multipliers. Using the six 8×8 multipliers in Table 3.6

Table 3.6: Using $M1$ and $M2$ to construct 8×8, 16×16, and 32×32 designs.

Size	**Design**	$\boldsymbol{P_1}$	$\boldsymbol{P_2}$	$\boldsymbol{P_3}$	$\boldsymbol{P_4}$
8×8	M8-1	M1	M1	M1	M1
	M8-2	M2	M2	M2	M2
	M8-3	Exact	M1	M1	M1
	M8-4	Exact	M2	M2	M2
	M8-5	Exact	Exact	Exact	M1
	M8-6	Exact	Exact	Exact	M2
16×16	M16-1	M8-1	M8-1	M8-1	M8-1
	M16-2	M8-2	M8-2	M8-2	M8-2
	M16-3	M8-3	M8-3	M8-3	M8-3
	M16-4	M8-4	M8-4	M8-4	M8-4
	M16-5	M8-5	M8-5	M8-5	M8-5
	M16-6	M8-6	M8-6	M8-6	M8-6
32×32	M32-5	M16-5	M16-5	M16-5	M16-5
	M32-6	M16-6	M16-6	M16-6	M16-6

to construct 16×16 ones, as specified in (3.4), we obtain 64 possible 16×16 multiplier designs. Since this is an impractically large number of possible designs, we only consider six designs using the simple scheme shown in Table 3.6. These designs are (1) the most accurate scaled-up variants using $M1$ and $M2$, referred to as M16-5 and M16-6, respectively; (2) the most hardware efficient scaled-up variants using $M1$ and $M2$, referred to as M16-1 and M16-2, respectively; and (3) two designs (one using $M1$, i.e. M16-3 and the other one using $M2$, i.e. M16-4) that appear to have a good trade-off between accuracy and hardware. Only one type of 8×8 multiplier is used to construct all of the 16×16 designs. The most accurate variants of the 16×16 multipliers, i.e. M16-5 and M16-6, are selected to construct 32×32 multipliers M32-5 and M32-6, respectively.

The same design approach can be applied to any $n \times n$ multiplier where n is a power of 2. Since we have six 8×8 multipliers and four $n \times n$ multipliers are required to build a $2n \times 2n$ multiplier, the number of possible designs is given by:

$$(6^4)^{log_2(\frac{n}{8})} = (6^4)^{log_2 n-3} = 6^{(\frac{n^2}{64})}. \quad (3.5)$$

According to (3.5), the number of possible designs increases exponentially with n^2. These designs have a wide range of accuracy-hardware trade-offs and could be utilized in different applications, based on application requirements.

3.1.4 Extension to signed Booth multipliers

The proposed approximate compressor can also be utilized in signed Booth multipliers. In a Booth multiplier, the PPs are generated using a Booth encoder, and the major difference between the unsigned and signed Booth multiplication is in the generation of the partial products. Therefore, the partial products in Booth multipliers can be accumulated using approximate compressors, but not the sign extension bits [56]. Following [55], an 8×8 Booth multiplier was designed and implemented using the proposed approximate compressors for the 8 LSBs while the 8 MSBs use exact compressors. Note that the sign extension of the partial product array is usually simplified by using the Baugh-Wooley algorithm [55].

3.2 Performance Evaluation

3.2.1 Accuracy analysis

An important metric for an approximate design is the output accuracy with respect to the exact result. We used the mean relative error distance (MRED) [13] as the metric to quantify the accuracy of the approximate designs. Table 3.7 shows the MRED, the error rate (ER), and the normalized mean error distance (NMED) (the mean error distance normalized by the maximum output of the accurate design [13]) for several 16×16 unsigned multipliers recently reported in the literature. Note that the ER is the percentage of the multiplications for which the approximate design produces a different result than the exact one. Better designs will tend to have a lower ER in addition to a small MRED. Since an exhaustive simulation of all possible input combinations is very time-consuming and unnecessary to gain the key insights, we simulated the accuracy of the approximate multipliers using MATLAB with 10 million uniformly distributed randomly generated input combinations. Error metrics

Table 3.7: Accuracy comparison for 16×16 and 8×8 approximate multipliers.

Multiplier size	Multiplier type	MRED	ER (%)	NMED
16×16	M16-1	0.0644	96.71	5.7×10^{-2}
	M16-2	0.0839	96.67	7.2×10^{-2}
	M16-3	0.0168	94.74	1.2×10^{-3}
	M16-4	0.0224	94.65	1.9×10^{-3}
	M16-5	0.0013	72.49	5.1×10^{-6}
	M16-6	0.0017	72.33	5.7×10^{-6}
	UDM [40] (2011)	0.0333	80.99	1.4×10^{-2}
	AM2-16 [44] (2014)	0.0013	97.96	5.3×10^{-6}
	ETM-7 [41] (2010)	0.0156	99.99	2.2×10^{-3}
	ACM4 [43] (2015)	0.0026	99.97	6.4×10^{-6}
	MUL2 [56] (2017)	0.0020	84.67	7.1×10^{-6}
	BAM-16 [38] (2010)	0.0021	99.97	3.5×10^{-5}
	TAM2-16 [45] (2016)	0.0020	99.98	3.1×10^{-5}
	AWTM-4 [57] (2014)	0.0033	99.94	8.3×10^{-6}
8×8	M16-1	0.0649	73.17	1.9×10^{-2}
	M16-2	0.0846	73.17	2.8×10^{-2}
	M16-3	0.0170	66.36	2.1×10^{-3}
	M16-4	0.0227	66.43	3.2×10^{-3}
	M16-5	0.0013	36.22	6.8×10^{-5}
	M16-6	0.0018	36.22	9.6×10^{-5}
	UDM	0.0328	47.09	1.4×10^{-2}
	AM2-16	0.0014	95.23	5.3×10^{-4}
	ETM-3	0.0846	93.10	1.3×10^{-2}
	ACM4	0.0028	99.03	1.2×10^{-4}
	MUL2	0.0022	79.23	3.1×10^{-4}
	BAM-16	0.0176	99.23	1.8×10^{-2}
	TAM2-16	0.0024	99.11	7.2×10^{-4}
	AWTM-4	0.1532	99.92	5.4×10^{-3}

MRED, ER, and NMED were calculated by simulating the 8×8 multipliers over their entire input space (65536 cases) and the results are also provided in Table 3.7.

The results in Table 3.7 show that the most accurate of the proposed 16×16 designs, M16-5 and M16-6, are more accurate than their competitors except AM2-16, which has the same MRED as M16-5. However, with respect to the ER and NMED, M16-5 is clearly more accurate than AM2-16. Note that according to Table 3.7, the trend that is seen in the 16×16 multipliers can also be seen in the 8×8 multipliers.

We also measured the MRED, ER, and NMED for the radix-4 Booth multiplier. This proposed design will be referred to as the compressor-based approximate Booth multiplier (CABM). CABM was compared to two state-of-the-art

Table 3.8: Accuracy comparison for 8×8 Radix-4 Booth multipliers.

Multiplier type	**MRED**	**ER (%)**	**NMED**
AWBM1 [14]	0.051	98.26	0.30
AWBM2 [14]	0.029	91.49	0.18
CABM	0.014	84.72	0.18

approximate Radix-4 Booth multipliers and the results are given in Table 3.8. Table 3.8 shows that CABM has the same NED as the AWBM2 while it has a smaller error rate. NED refers to the normalized error distance, which is the average error distance normalized by the maximum possible error. Moreover, CABM is the most accurate design with respect to the MRED.

3.2.2 Hardware analysis

All of the designs were implemented in VHDL and then synthesized by using the Synopsys Design Compiler (DC) for ST's CMOS 28-nm process. The supply voltage and the temperature in all simulations were set to $1\,V$ and $25°C$, respectively. Note that we used an exact 16-bit Wallace tree multiplier (Wallace-16) as the baseline exact multiplier for the comparison.

According to our simulations, for the 16×16 unsigned designs, the fastest and the smallest design is ETM-7, which is 3.08% faster and 40.74% smaller and than our fastest and smallest design, M16-2; however, ETM-7 consumes 15.22% more power than M16-2. With respect to power consumption, the proposed designs M16-2 and M16-1 are the most power-efficient multipliers. Even our most accurate designs, M16-5 and M16-6, are among the most power-efficient and energy-efficient ones with relatively small power-delay product (PDP) values. The only design that consumes less energy than M16-5 and M16-6, is ETM-7; however, ETM-7 is almost 10× less accurate than M16-5 and M16-6. Also, the proposed M16-2 has the lowest PDP value among all the designs.

The same trends that appear in 16×16 designs were also observed in 32×32 multipliers, i.e. M32-5 and M32-6 are the most hardware-efficient designs with at least 21.51% (for M32-5) and 21.61% (for M32-6) smaller PDP compared to ACM4, which has the smallest PDP among the other designs.

Table 3.9: Hardware comparison for 8×8 Radix-4 Booth multipliers.

Multiplier type	**Delay (nS)**	**Power (μW)**	**Area (μm^2)**	**PDP (fJ)**	**PDP×MRED**
AWBM1	1.80	99.375	393.12	178.875	9.122
AWBM2	1.66	68.750	285.62	114.125	3.309
CABM	1.63	69.678	284.32	113.575	1.788
Exact Booth	2.01	125.42	436.87	252.094	-

We further considered both the MRED and PDP metrics to evaluate different designs, as in [13]. Fig. 3.4(a) compares the products of MRED and PDP values and Fig. 3.4(b) shows the -log10 (MRED) vs. PDP for the considered unsigned 16×16 multipliers. Since the MRED values are so close, they are plotted on a logarithmic scale for easier comparison. Note that designs at the top-left corner are the best designs, which have small PDPs with high accuracies. As the results in Fig. 3.4(a) show, M16-5 and M16-6 have the smallest PDP-MRED products.

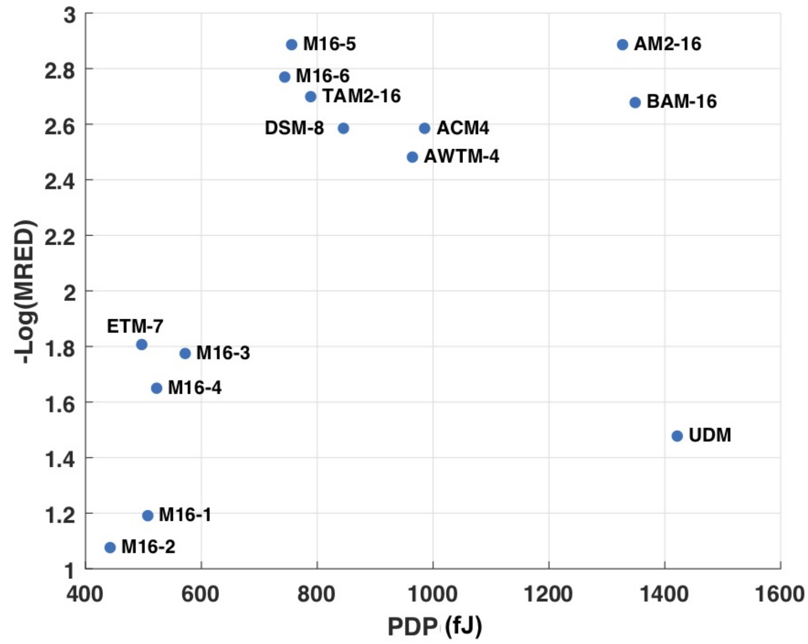
A similar comparison was done for the radix-4 Booth multipliers. As the results in Table 3.9 show, AWBM2 is slightly more efficient than the proposed CABM in terms of delay, power, and area; however, according to Table 3.9, AWBM2 is more than 2× less accurate than the CABM. The MRED-PDP products are also obtained for Radix-4 Booth multipliers and the results are given in Table 3.9. It is shown that the proposed CABM has the lowest MRED-PDP product.

3.3 Example Applications

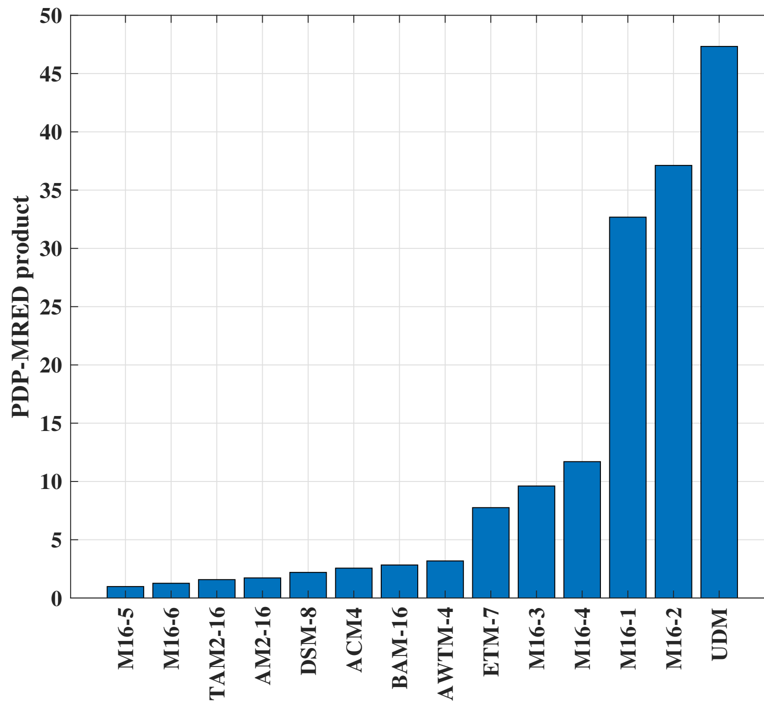
The effectiveness of the proposed designs was evaluated using image sharpening, JPEG applications, and for the first time, an interference nulling calculation for the receiver in a MIMO wireless communication system.

3.3.1 Image sharpening

The PSNR and structural similarity structural similarity index (SSIM) are used as objective quality measures. The PSNR and SSIM values for several approximate multipliers are depicted in Fig. 3.5, which confirms that M16-5 is more accurate than the other designs. Note that with respect to the



(a) MRED vs. PDP.



(b) PDP-MRED product.

Figure 3.4: MRED and PDP of the approximate multipliers.

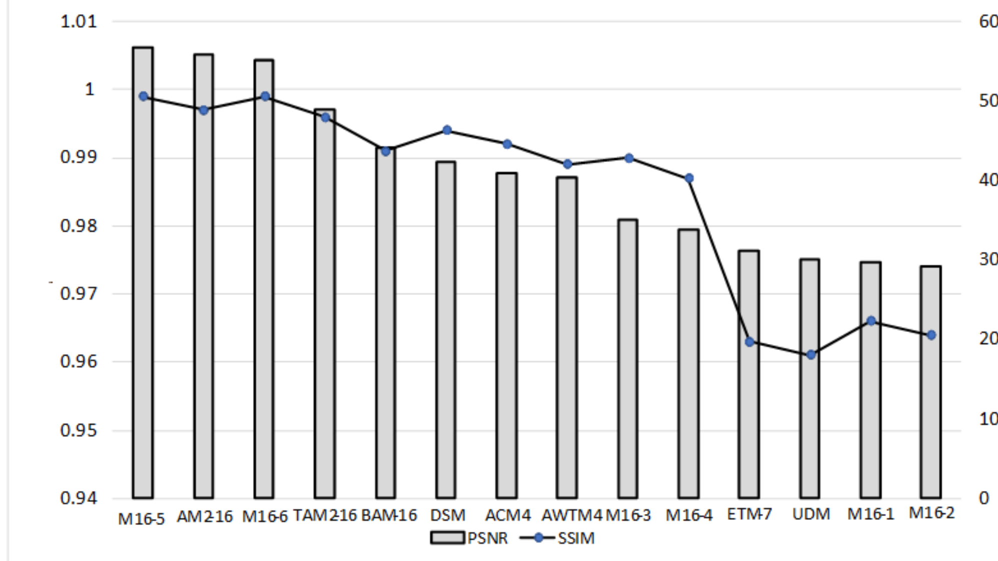


Figure 3.5: PSNR and SSIM values for the image sharpening application.

SSIM, M16-5 and M16-6 are the best designs with the highest SSIM values. In general, the rankings of the weaker designs are slightly different in some cases, but the trend is roughly the same as the PSNR values.

3.3.2 JPEG image compression

Table 3.10 reports the PSNR and SSIM values for several approximate multipliers for four increasing QFs. As the results in Table 3.10 show, M16-5 has the highest PSNR and SSIM values for the four considered QFs, followed by M16-6. Note that the reference image for computing the SSIM and PSNR values in image sharpening application is the image reconstructed using exact multipliers. However, it might be more reasonable to use the original image, i.e. the image before compression, as the reference image in the JPEG compression application. The results in Table 3.10 show that the exact multipliers in a JPEG compressor can be replaced by approximate multipliers for power and area saving purposes at the cost of negligible image quality degradation.

3.3.3 Multiple-input multiple-output wireless systems

We modeled an 8×8 multiple-input multiple-output (MIMO) system, see Fig. 2.1, in which all multiplications in the receiver block use the proposed approxi-

Table 3.10: Decompressed image quality comparison, PSNR and SSIM metrics for JPEG compression.

Metric	Multiplier type	$QF = 60$	$QF = 70$	$QF = 80$	$QF = 90$
PSNR	Exact	27.81	27.34	27.01	26.91
	M16-1	23.54	17.21	22.22	18.26
	M16-2	22.93	13.27	19.11	13.96
	M16-3	25.21	25.72	23.17	18.65
	M16-4	24.51	23.80	19.63	14.11
	M16-5	26.43	26.65	25.72	25.54
	M16-6	26.41	26.63	25.62	25.28
	AM2-16	26.17	26.07	25.51	24.48
	ACM4	26.02	25.95	25.13	24.21
	MUL2	26.21	26.44	25.64	25.37
	AWTM4	25.88	25.67	24.82	24.03
SSIM	Exact	0.98	0.98	0.97	0.97
	M16-1	0.83	0.79	0.79	0.79
	M16-2	0.83	0.79	0.79	0.73
	M16-3	0.95	0.93	0.90	0.87
	M16-4	0.95	0.93	0.90	0.83
	M16-5	0.97	0.97	0.96	0.95
	M16-6	0.97	0.96	0.96	0.95
	AM2-16	0.96	0.93	0.92	0.90
	ACM4	0.95	0.92	0.92	0.89
	MUL2	0.96	0.96	0.95	0.95
	AWTM4	0.93	0.92	0.91	0.88

mate multipliers. In addition, three different codes were considered: Hamming (7, 4), extended Golay (24, 12), and two variants of low-density parity-check (LDPC) codes: LDPC (1024, 512), and LDPC (2048, 1024) [18].

The bit error rate (BER) vs. signal-to-noise ratio (SNR) characteristic was computed for seven different cases: one for the exact multiplier and six for the six variants of the proposed design. The results are shown in Fig. 3.6 for the Hamming (7, 4), extended Golay (24, 12), LDPC (1024, 512), and LDPC (2048, 1024) codes.

Since the six proposed 16×16 approximate multipliers cover a wide range of accuracy, and given that M16-2 and M16-5 are the least and the most accurate designs, we only consider these six designs in this sub-section. First, we aim to show the practicality of approximate multipliers in MIMO receiver applications in general. Second, we hypothesize that the performance of the other designs in the described MIMO system would be similar to one of the six proposed multipliers with the closest accuracy.

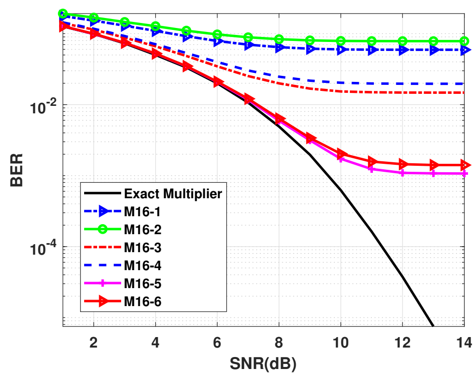
Fig. 3.6 shows that for the lowest SNRs and, consequently, relatively high

BERs, the exact and approximate designs are equally (and massively) affected by noise. This implies that the computation errors caused by the use of approximate multipliers are insignificant compared to the already large number of errors caused by noise. Although the least accurate approximate multiplier designs should be quite acceptable at low SNR operation, there are essentially no applications that will operate in that regime. When operating at higher, more typical SNR levels, Fig. 3.6 shows that the most accurate variants of the proposed design, namely M16-5 and M16-6, can match the BER vs. SNR performance of a design that uses exact multipliers down to lower and hence more practical BERs.

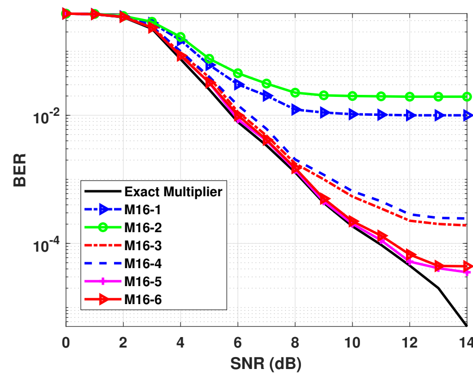
As the SNR increases, the computation errors caused by the use of approximate multipliers will eventually dominate the random errors and produce a leveling off of the BER curve, a so-called error floor. This is the operating region where the error correcting code cannot compensate for the multiplier's inaccuracies. This error floor can be easily seen, especially in Figs. 3.6(a) and 3.6(b), where the weakest codes, i.e. the (7, 4) Hamming code and the extended (24, 12) Golay code, respectively, are employed. Note that depending on the accuracy of the approximate design and the strength of the correcting code, the error floor is encountered at different SNR levels.

Figs. 3.6(c) and 3.6(d) show that approximate multipliers, especially M16-5 and M16-6, can safely replace exact multipliers in a MIMO system with LDPC codes to reduce the power, delay, and area (M16-5 and M16-6 have 59.25% and 59.89% smaller PDP compared to the exact Wallace-tree multiplier, respectively) at a relatively low cost in performance degradation. The advantages of the approximate multiplier implementation could be even more significant in larger MIMO systems, such as massive MIMO systems with many dozens of antennas, and also if many parallel multipliers are required to meet the required data throughput.

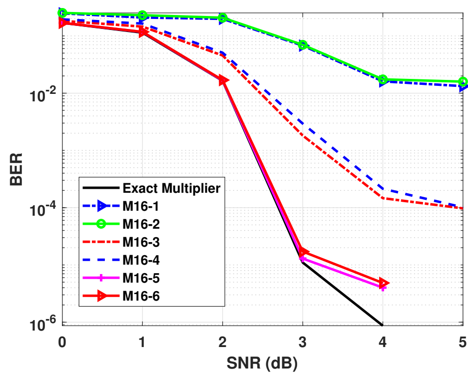
We performed some simulations and realized that using approximate multipliers increases the number of required iterations to get to a desired BER at a given SNR; the results are given in Table 3.11. According to Table 3.11 the number of required iterations increases at higher SNR levels, where the



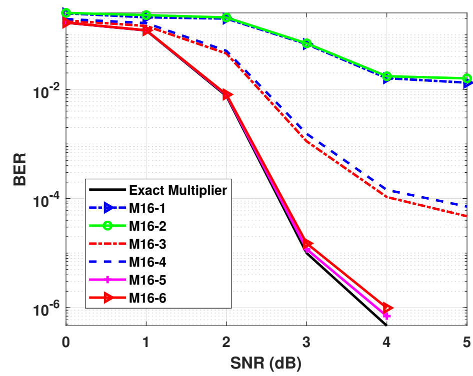
(a) (7, 4) Hamming code.



(b) Extended (24, 12) Golay code.



(c) (1024, 512) LDPC code.



(d) (2048, 1024) LDPC code.

Figure 3.6: BER vs. SNR.

Table 3.11: Required increase in the number of iterations to get to a desired BER at a given SNR level.

Approximate multiplier	**(BER, SNR)**	
	(10^{-5}, 3 dB)	(5×10^{-6}, 4 dB)
M16-5	3.5%	9.3%
M16-3	6.1%	15.6%

errors caused by approximate multipliers dominate channel noise. Table 3.11 also shows that less accurate multipliers require more iterations to get to the desired BER at a given SNR level. This trade-off could be exploited in some design problems to save on hardware but pay more in decoding time and energy.

Analyzing the results at a reasonable operating point, e.g. an SNR of 4 dB for the (2048, 1024) LDPC code using the M16-5 approximate multiplier, shows a 9.3% increase in the number of required iterations. More iterations mean more execution time and, consequently, more energy consumption. In fact, the energy consumption increases by 9.3%. However, as previously mentioned, M16-5 consumes 59% less energy than the exact multiplier and saves 20% on the area. Hence, it would still be practical to use approximate multipliers in this application.

3.4 Summary

We proposed an approximate 4:2 compressor that is employed to construct two 4×4 multipliers with different accuracies. The 4×4 designs are then scaled up to 16×16 and 32×32 multipliers that provide a wide range of accuracy-performance trade-offs. The least accurate of the proposed designs, M16-2, has the smallest PDP among other approximate designs while the most accurate of the proposed designs, M16-5, has a 44% smaller PDP compared to AM2-16, which has a similar accuracy in MRED. Moreover, M16-5 is more accurate than the other approximate designs in the literature. The proposed compressor is also employed in a radix-4 Booth multiplier, resulting in a low-power signed multiplier (CABM) with a small MRED. The simulation results

reveal the advantages of the CABM over other designs in terms of the MRED and PDP-MRED product. The proposed multipliers have been evaluated in image sharpening and JPEG applications. It is shown that M16-5 produces more accurate output than other approximate multipliers by achieving a higher output quality while consuming less power. In addition, for the first time, approximate multipliers are evaluated in the interference nulling calculation of the MIMO baseband receiver. We measured how computation errors can be corrected along with errors caused by channel noise so that the transmitted data can be recovered without additional hardware cost using powerful error detection and correction codes (e.g., LDPC codes) that are already present in the communication system. It is shown that approximate multipliers can often safely replace exact multipliers with relatively low performance degradation.

Chapter 4

Improving the Accuracy and Hardware Efficiency of Neural Networks Using Approximate Multipliers

Given that multipliers are the main computational bottleneck of neural networks (NNs) and a major hardware cost [26], [58], [59], this chapter focuses on the use of approximate multipliers in NNs.

Many approximate multipliers have already been proposed in the literature that decrease the hardware cost while maintaining acceptably high accuracy. We divide the known approximate multipliers into two main categories: (1) deliberately-designed multipliers, which includes designs that are obtained by making some changes in the truth table of the exact designs [13], and (2) CGP-based multipliers, which are designs that are generated automatically using the CGP heuristic algorithm [8]. Note that there are other classes of approximate multipliers that are based on analog mixed-signal processing [60], [61]. However, they are not considered in this thesis since our focus is on digital design, which is more flexible and aggressively scalable in implementation than analog/mixed-signal based designs.

There is typically a trade-off between the accuracy and hardware cost in approximate multipliers, and there is no one best design for all applications. Selecting the appropriate approximate multiplier for any specific application is typically a complex question that requires careful consideration of multiple

alternative designs. The objective of this chapter is to find the approximate multipliers that improve the performance of a NN, i.e. by reducing the hardware cost while preserving an acceptable output accuracy. To the best of our knowledge, this is the first work that attempts to find the critical features in an approximate multiplier that make it superior to others for use in a NN.

Our benchmark multipliers, including 500 CGP-based approximate multipliers and 100 variants of deliberately-designed multipliers, are evaluated for two standard NNs: a MLP that classifies the MNIST dataset [62] and a CNN, LeNet-5 [21], that classifies the SVHN dataset [63]. After each network is trained while using double-precision floating-point exact multipliers, the accurate multipliers are replaced with one approximate design (selected from the set of benchmark multipliers), and then five steps of retraining are performed. This process is repeated for each of the benchmark multipliers, resulting in 600 variants for each of the two considered NNs. The retraining is done for each approximate multiplier only once. Then the inference operation is performed on the NNs to evaluate their output accuracy. Since the simulations always start from the same point, i.e. we run the retraining steps on the pre-trained network (with exact multipliers), there is no randomness and, therefore, the results will be consistent if the simulation is repeated.

4.1 Evaluation of Approximate Multipliers in Neural Networks

This section considers both application-dependent and -independent metrics to evaluate the effects of approximate multipliers in NNs.

4.1.1 Application-independent metrics

Application-independent metrics measure design features that do not change from one application to another. Given that approximate multipliers are digital circuits, these metrics can be either output error or hardware cost metrics. Error function metrics are required for feature selection analysis.

The main four error metrics are the ER, error difference (ED), absolute

Table 4.1: Considered features of the output error function.

Feature	Description
ER	Error rate
Var-ED	Variance of the ED values
Mean-ED	Mean value of the ED values
RMS-ED	Root mean square of the ED values
Var-RED	Variance of the RED values
Mean-RED	Mean value of the RED values
RMS-RED	Root mean square of the RED values
Var-AED	Variance of the AED values
Mean-AED	Mean value of the AED values

error difference (AED), and the relative error difference (RED). We evaluated all 600 multiplier designs using the 9 features extracted from these four main metrics, as given in Table 4.1. All of the considered multipliers were implemented in MATLAB and simulated over their entire input space, i.e. for all $256 \times 256 = 65536$ combinations.

The definitions for most of these features are given in (4.1); those that are not given in (4.1) are evident from the description. Note that E and A in (4.1) refer to the exact and approximate multiplication results, respectively. Also note that the mean/variance-related features in Table 4.1 are measured over the entire output domain of the multipliers (N=65536), i.e. $256 \times 256 = 65536$ cases for the employed 8-bit multipliers.

$$\begin{gathered} ED = E - A \\ RED = 1 - \frac{A}{E} \\ AED = |E - A| \\ RMS_{ED} = \sqrt{\left(\frac{1}{N} \times \sum_{i=1}^{N} (A_i - E_i)^2\right)} \\ Var_{ED} = \frac{1}{N} \times \sum_{i=1}^{N} \left(ED_i - \frac{1}{N} \times \sum_{i=1}^{N} ED_i\right)^2 \end{gathered} \tag{4.1}$$

Note that the variance and the root mean square (RMS) are distinct metrics, as specified in (4.1). Specifically, the variance measures the spread of the

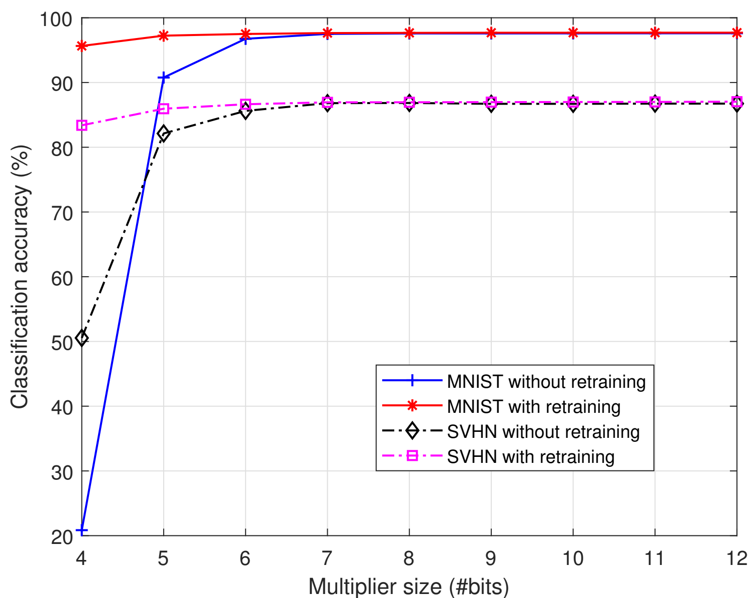


Figure 4.1: Effects of multiplier size on classification accuracy.

data around the mean while the RMS measures the spread of the data around the best fit. In the case of error metrics, the best possible fit is zero.

All of the multipliers were implemented in VHDL and/or Verilog and synthesized using the Synopsys Design Compiler (DC) for the STMicro CMOS 28-nm process to obtain the most important hardware metrics: the power dissipation, the circuit area, and the critical path delay. These hardware metrics are useful for identifying the most hardware-efficient multiplier among those with similar error characteristics.

We also generated 500 approximate multipliers using the CGP algorithm. The Verilog, C, and MATLAB codes for all the designs and their error and hardware characteristics can be found in [64].

4.1.2 Application-dependent metrics

The classification accuracies of the MLP and LeNet-5 networks were evaluated over the MNIST and SVHN datasets, respectively. All 600 of the approximate multiplier designs (100 deliberately-designed and 500 CGP-based) were employed in both NNs and the resulting classification accuracy of the NNs was calculated.

The results show that without performing the retraining steps, the 6-bit multiplier is the smallest design that is able to provide acceptable results. On the other hand, when retraining steps are considered (we performed 5 retraining steps), 4-bit designs can be used with only 2% degradation in classification accuracy compared to 8-bit designs. Note that the 8-bit designs were found to be only 0.04% less accurate than the 12-bit designs.

Interestingly, we observed that almost all of the approximate multipliers result in similar classification accuracies for the MNIST dataset, regardless of the circuit design. This was expected since MNIST is a relatively easy dataset to classify. This bodes well for the use of cheaper, approximate multiplier designs. The SVHN dataset, however, shows drops in classification accuracy more clearly than the MNIST dataset when reduced-width multipliers are used. This might be due to the fact that SVHN data is inherently harder to classify than the MNIST data.

4.1.3 Overfitting

An interesting finding from this work is the observation that a few approximate multipliers have slightly improved the classification accuracy over the exact multipliers. This is a potentially significant result since it means that we can use less hardware and less power and yet get better results. We believe that overfitting in NNs may be the main reason for this interesting result.

Overfitting happens when the network is trained so much that it produces overly complex and unrealistic class boundaries when deciding whether to classify a data point into one class or another [65]. An overfitted network performs well on the training data since it effectively memorizes the training examples, but it performs poorly on test data because it has not learned to generalize to a larger population of data values. Several solutions have been proposed in the literature to avoid overfitting such as dropout [65], weight decay [66], early stopping [67], and learning with noise [68]–[73].

Dropout techniques help to avoid overfitting by omitting some neurons from a NN. More specifically, for each training case, a few neurons are selected and removed from the network, along with all their input and output

connections [65]. Weight decay is another strategy for handling overfitting in which a weight-decay term is added into the objective function. This term reduces the magnitude of the trained weights and makes the network's output function smoother and, consequently, improves the generalization (i.e., a well-generalized NN can more accurately classify unseen data from the same population as the learning data) and reduces the overfitting [66]. Early stopping approaches stop the training process as soon as a predefined threshold value for classification accuracy has been achieved with the objective of preventing too much training and hence overfitting [67].

Last, but not the least, the addition of noise to the synaptic weights of NNs has been found to be a low overhead technique for improving the performance of a NN [69]. The authors in [71] report up to an 8% improvement in classification accuracy by injecting stochastic noise into the synaptic weights during the training phase. The noise injected into the synaptic weights in NNs can be modeled as either additive or multiplicative noise [72], [73], as defined in (4.2), and both have been found to be beneficial.

$$\begin{aligned} Additive\,noise &: W^*_{ij} = W_{ij} + \delta_{ij} \\ Multiplicative\,noise &: W^*_{ij} = W_{ij}\delta_{ij} \end{aligned} \tag{4.2}$$

In (4.2), δ_{ij} denotes the injected noise and W_{ij} denotes the noisy synaptic weight between the ith neuron in layer L and the jth neuron in layer $L+1$. The input of neuron j in layer $L+1$, denoted by n_j, is calculated by

$$n_j = \textstyle\sum_{i=1}^{N_L} x_i \times w_{ij}, \tag{4.3}$$

where N_L is the number of neurons in layer L, and x_i and w_{ij} denote a neuron's output and its connection weight to neuron j, respectively. If the exact multiplication in (4.3) is replaced with an approximate one, the approximate product for multiplicand a and multiplier b is given by

$$M(a, b) = a \times b + \Delta(a, b), \tag{4.4}$$

where the dither (error function) $\Delta(a, b)$ is the function that expresses the dif-

ference between the output of an exact multiplier and that of an approximate multiplier. By combining (4.3) and (4.4) we obtain:

$$
\begin{gathered}
n_j = \sum_{i=1}^{N_L} x_i \times w_{ij} = \sum_{i=1}^{N_L} M(x_i, w_{ij}) \approx \\
\xrightarrow{approximate\ multipliers} \sum_{i=1}^{N_L} M'(x_i, w_{ij}) = \\
\sum_{i=1}^{N_L} \Big((x_i \times w_{ij}) + \Delta(x_i, w_{ij}) \Big) = \\
\sum_{i=1}^{N_L} \Big(x_i \times (w_{ij} + \frac{\Delta(x_i, w_{ij})}{x_i}) \Big) = \sum_{i=1}^{N_L} x_i \times w_{ij}^{*}.
\end{gathered}
\tag{4.5}
$$

Note that the "noise" term, $\Delta(x_i, w_{ij})$ in (4.5) depends on the neuron output x_i and is a different function for each individual design. Hence, we cannot compare the result in (4.5) to the definitions given in (4.2) since $\Delta(x_i, w_{ij})$ is an unknown function that changes for different multipliers. However, we hypothesize that the same argument that adding "noise" onto the synaptic weights, as we did in (4.5), can sometimes help to avoid overfitting in NNs.

To provide experimental support for this hypothesis, we built an analytical approximate multiplier, which is defined as:

$$M'(a, b) = a \times b + \epsilon, \tag{4.6}$$

where ϵ denotes the injected noise. We added Gaussian noise since it is the most common choice in the literature [68]–[70]. We used this noise-corrupted exact multiplier in an MLP (784-300-10, i.e. 784 neurons in the first layer, 300 hidden neurons, and 10 output neurons) and tested it over the MNIST dataset. Fig. 4.2 shows how the accuracy is affected by increasing noise levels. Note that the noise's mean and standard deviation in the noise-corrupted multiplier are the exact multiplication product (EMP) and a percentage of the EMP, respectively. This percentage is given by the term noise level in Fig. 4.2.

Since the added Gaussian noise is stochastic, we ran the simulations 10 times and report the average results. The results in Fig. 4.2 confirmed the results in [68], [73]: adding small amounts of noise can indeed improve the classification accuracy. However, as shown in Fig. 4.2, adding too much noise

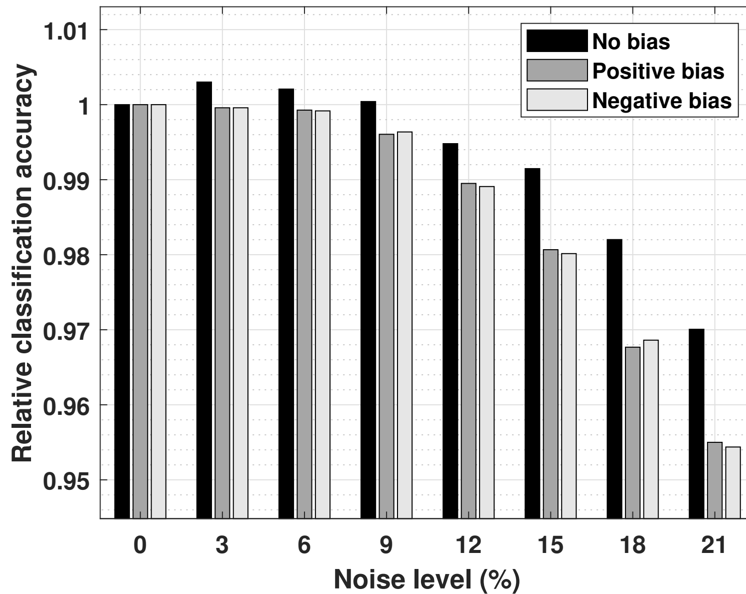


Figure 4.2: MNIST classification accuracy, training and testing with additive Gaussian noise.

will degrade the classification accuracy. Note that the classification accuracies in Fig. 4.2 are normalized to the classification accuracy obtained by using exact multipliers.

Additionally, we injected Gaussian noise with positive and negative offsets in our accuracy analysis in Fig. 4.2 to show the negative effect of biased noise on the classification accuracy. For biased noise, the errors are more likely to accumulate and, therefore, the accuracy drops. The mean is changed to 1.1×EMP and 0.9×EMP to model the positive and negative offsets, respectively.

4.2 Critical Features of Multipliers for NNs

In the previous section, we showed that adding noise to the multipliers can improve the accuracy of a NN. We also modeled the difference between an exact multiplier and an approximate one using the error function $\Delta(x_i, w_{ij})$ of the approximate multiplier, see (4.5). In this section we consider different multipliers to investigate which properties of the error function might make one design superior to others when employed in a NN.

As previously mentioned, the error function depends on the multiplier and is a different function for each individual design. An exact analysis of the error functions for different multipliers is impractical and so instead we sought the relevant features of the error functions using statistical techniques. Nine seemingly relevant features of the error function were identified and listed in Table 4.1. In order to determine the most discriminative features of the error functions, i.e. the features that contribute the most to the performance of an approximate multiplier in a NN, the nine features in Table 4.1 were applied to several standard statistical feature selection tools (as described next).

To be able to run the feature selection algorithms, the multipliers were classified into two categories based on their performance in NNs. We defined a threshold accuracy, A_{th}, and classified the multipliers that produce higher accuracies than A_{th} into "Class 1" while the others drop into "Class 0". Since in the NN accuracy analysis some approximate multipliers produce slightly higher classification accuracies than exact multipliers when employed in NNs, it was convenient to choose $A_{th} = ACC_{Exact}$, which is the NN classification accuracy that is obtained when exact multipliers are employed in the network's structure. Note that the average noise level for the Class 1 approximate multipliers is 2.61%, which is close to the obtained noise level range in Fig. 4.2.

4.2.1 Feature selection

Feature selection is a statistical way of removing less relevant features that are not as important to achieving accurate classification performance. There are many potential benefits to feature selection including facilitating data understanding and space dimensionality reduction [74], [75]. In this article, feature selection algorithms are used to select a subset of the multipliers' error function features that are the most useful for building a good predictor. This predictor anticipates the behavior of an approximate multiplier in a NN.

Scikit-learn is a free machine learning tool that is widely used for feature selection [76]. It accepts an input data array and their corresponding labels to build an estimator that implements a fitting method. We used the three classifiers recursive feature elimination (RFE) [77], mutual information (MI)

[78], and Extra-Tree [79].

The RFE classifier iteratively prunes the least important features from the current set of features until the desired number of features is reached. The ith output of the RFE corresponds to the ranking position of the ith feature, such that the selected (i.e., the estimated best) features are assigned a rank of 1. Note that in RFE, the nested feature subsets contain complementary features and are not necessarily individually the most relevant features [77]. MI is another useful feature selection technique that relies on nonparametric methods based on entropy estimation from the K-nearest-neighbor distances, as described in [78]. Each feature is assigned a score, where higher scores indicate more important features. Finally, the tree-based estimators can also be used to compute feature importance to discard less relevant features. Extra-Tree, an extremely randomized tree classifier, is a practical classifier that is widely used for feature selection [79]. Similar to MI, the ith output of this classifier identifies the importance of the ith feature, such that the higher the output score, the more important the feature.

The results for each of the three mentioned feature selection algorithms are provided in Table 4.2. The results in Table 4.2 show that **Var-ED** is the most important feature according to all three classifiers. **RMS-ED** is another important metric, i.e. the most important metric according to RFE, the second most critical feature in MI, and the third most significant metric in Extra-Tree classifier. Our simulation results show that the average value of the **Var-ED** and **RMS-ED** features for "Class 0" multipliers are 20.21× and 6.42× greater than those of the "Class 1" approximate multipliers, respectively.

Other important features that have a good ranking in the three classifiers are **MEAN-AED** and **VAR-AED**. We also observed that the multipliers that produced better accuracies in a NN than the exact multiplier (Class 1 multipliers) all have double-sided error functions. Thus they overestimate the actual multiplication product for some input combinations and underestimate it for others. Having double-sided EDs seems to be a necessary but not a sufficient condition for better accuracy.

Given that "Class 1" approximate multipliers tend to have smaller **Var-**

Table 4.2: Ranking of the error function features.

Features	Feature ranking		
	RFE	**MI (score/ranking)**	**Extra-Tree (score/ranking)**
ER	5	0.2844 / 7	0.0518 / 9
Mean-ED	1	0.2124 / 8	0.0534 / 8
Var-ED	1	0.3884 / 1	0.2074 / 1
RMS-ED	1	0.3861 / 2	0.1383 / 3
Mean-RED	7	0.1623 / 9	0.0562 / 7
Var-RED	6	0.3309 / 5	0.0993 / 6
RMS-RED	4	0.3253 / 6	0.1235 / 5
Mean-AED	2	0.3655 / 3	0.1452 / 2
Var-AED	3	0.3424 / 4	0.1244 / 4

ED and **RMS-ED** values and the observation that double-sided errors are necessary for a good approximate multiplier, the difference in the error magnitude should be small to meet the **RMS-ED** requirement i.e., having small **RMS-ED** values. Moreover, since the error should be double-sided to have a small variance, these errors should be distributed around zero.

4.2.2 Training the classifier

Now, having found the most important features of the error function of an approximate multiplier, we can use them to predict how well a given approximate multiplier would work in a NN. In this sub-section we explain how to build a classifier that has the error features of an approximate multiplier as inputs and predicts if it belongs to Class 1 or Class 0.

NN-based classifier

The error features of 500 randomly selected multipliers were used to train the NN-based classifier and those of the 100 remaining multipliers were used as the test samples to obtain the classification accuracy of the trained model. We designed a 3-layer MLP with 20 neurons in the hidden layer and 2 neurons in the output layer (since we have two classes of multipliers). The number of neurons in the input layer equals the number of features that are considered for classification. The number of considered multiplier error features that

were used as inputs to the NN-based classifier was varied from 1 up to 9 (for 9 features in total, see Table 4.1). The resulting classification accuracies, plotted in Fig. 4.3, reflect how well the classifier classifies approximate multipliers into Class 1 or Class 0.

Note that when fewer than nine features are selected, the combination of features giving the highest accuracy is reported in Fig. 4.3. The combination of features is selected according to the results in Table 4.2 and is given in Table 4.3.

To choose two features, for example, the candidate features are selected from the top-ranked ones in Table 4.2: (1) **Var-ED** and **Mean-AED** (by Extra-Tree), (2) **Var-ED** and **RMS-ED** (by MI), and (3) **Mean-ED**, **Var-ED**, and **RMS-ED** (by RFE). For these four features (i.e. **Mean-ED**, **Var-ED**, **RMS-ED**, and **Mean-AED**), we consider all six possible combinations and report the results for the combination that gives the highest accuracy. Using the same process as in this example, the feature combinations for which the accuracy is maximized were found and are provided in Table 4.3.

As shown in Fig. 4.3, the highest classification accuracy is achieved when two features are used as inputs to the NN-based classifier, namely Var-ED and RMS-ED. Also, Fig. 4.3 shows that using more than two features does not necessarily result in a higher accuracy.

MATLAB classification learner application

The MATLAB software environment provides a wide variety of specialized applications [80]. In particular, the classifier learner application, available in the "apps gallery", allows us to train a model (classifier) that predicts if a multiplier falls into "Class 0" or "Class 1" when applied to a NN. This application provides the option of choosing a model type, i.e. decision trees, K nearest neighbors, support vector machines (SVMs), logistic classifiers among others. We considered all of these model types (with their default settings) to find the model that most accurately fits the classification problem. Similarly, 500 randomly selected multipliers were used to train the model and the 100 remaining multipliers as the test samples to obtain the classification accuracy

Table 4.3: Feature combinations that give the highest multiplier classification accuracy.

Number of features	Selected combination
1	Var-ED
2	Var-ED, RMS-ED
3	Var-ED, RMS-ED, Mean-AED
4	Var-ED, RMS-ED, Mean-AED, Var-AED
5	Var-ED, RMS-ED, Mean-AED, Var-AED, RMS-RED
6	Var-ED, RMS-ED, Mean-AED, Var-AED, RMS-RED, Mean-ED
7	Var-ED, RMS-ED, Mean-AED, Var-AED, RMS-RED, Mean-ED, Var-RED
8	Var-ED, RMS-ED, Mean-AED, Var-AED, RMS-RED, Mean-ED, Var-RED, ER

of the trained model.

Fig. 4.3 also shows the effect of the number of selected features on the accuracy of each of the three considered classifiers. Note that the SVM- and KNN-based classifiers achieve higher accuracies than the decision tree-based classifier. All three classifiers achieve better accuracies than the NN-based classifier.

Similar to the NN-based classifier, the classifier's accuracy for the combination of features that gives the highest accuracy is shown in Fig. 4.3 when fewer than 9 features are selected. The highest classification accuracy for the SVM- and KNN-based classifiers is achieved when only two features are used as inputs to the classifier: i.e. **Var-ED** and **RMS-ED**. However, the decision tree-based classifier has the highest accuracy when only one feature, **Var-ED**, is considered.

Verifying the classifiers

The trained SVM classifier was verified in the previous subsection by using 100 approximate multipliers, where an accuracy of almost 86% was achieved. In this section, the SVM classifier is used to predict the performance of 14

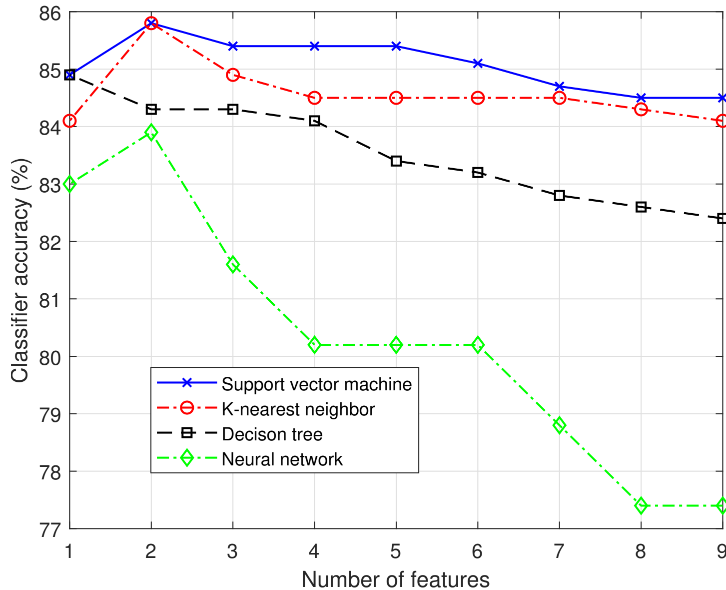


Figure 4.3: Effect of the number of selected features on approximate multiplier classifier accuracy.

representative approximate multipliers in a different benchmark NN. The SVM classifier is selected since it has the best performance compared to the other classifiers, see Fig. 4.3.

Ideally, we would want to verify the classifier using all 600 approximate multipliers. However, the large number of multipliers in a deep NN benchmark and the large number of images in the dataset would make the exhaustive experiment prohibitively time consuming. Therefore, in addition to the 100 previously considered multiplier, 5 multipliers were randomly selected from each class of approximate multipliers, plus the two multipliers that provided the best accuracy when used in a NN to classify the SVHN and MNIST datasets, and the two multipliers that had the worst accuracy for those same datasets. The SVM classifier was used to predict the behavior of each of these multipliers in a given NN benchmark. Then, these multipliers were used in the NN to verify the classifier's accuracy.

AlexNet is considered as the benchmark NN and trained to classify the ImageNet dataset [81]. AlexNet is a CNN with 9 layers: an input layer, 5 convolution layers, and 3 fully-connected layers [23]. Note that training a

Table 4.4: Classification accuracy of AlexNet on the ImageNet LSVRC-2010 dataset.

Class of multipliers	Multiplier	SVM classifier
Class 1	M0: Randomly selected	✓
	M1: Randomly selected	✗
	M2: Randomly selected	✓
	M3: Randomly selected	✓
	M4: Randomly selected	✓
	Best for SVHN	✓
	Best for MNIST	✓
Class 0	M5: Randomly selected	✓
	M6: Randomly selected	✓
	M7: Randomly selected	✓
	M8: Randomly selected	✓
	M9: Randomly selected	✓
	Worst for SVHN	✓
	Worst for MNIST	✓

deep CNN over a big dataset, such as the ImageNet, would be very time consuming. Hence, we used the MATLAB pre-trained model and performed 10 retraining steps (using the approximate multipliers) as an alternative to training the network from scratch.

Table 4.4 shows how the SVM classifier anticipates the performance of each of the 14 multipliers (i.e., the five randomly selected multipliers from each class of approximate multipliers and the four multipliers that provided the best and the worst accuracies when used in a NN to classify the SVHN and MNIST datasets) in AlexNet.

As shown in Fig. 4.3, none of the classifiers is 100% accurate. For instance, the AlexNet implemented with approximate multiplier $M1$ has a worse accuracy than A_{th} (i.e., the accuracy of AlexNet implemented with exact multipliers) even though the multiplier is classified into "Class 1" (see Table 4.4). However, this misclassified multiplier produces an accuracy close to A_{th} and the difference in accuracy (0.41%) is small.

While some multipliers might perform well for one dataset, they might not work well for other datasets. In other words, the performance of a multiplier is application-dependent. To illustrate this claim, we have plotted the Pareto-optimal designs in power-delay product (PDP) for the SVHN dataset using all

600 approximate multipliers in Fig. 4.4(a).

Fig. 4.4(b) shows the performance of the Pareto-optimal multipliers in PDP for the SVHN dataset for the MNIST dataset. Note that a multiplier is considered to be PDP-Pareto optimal if there does not exist any other multiplier which improves the classification accuracy with the same PDP. It is clear from Fig. 4.4 that the Pareto-optimal designs for the two datasets are different.

4.3 Error and Hardware Analysis of Approximate Multipliers

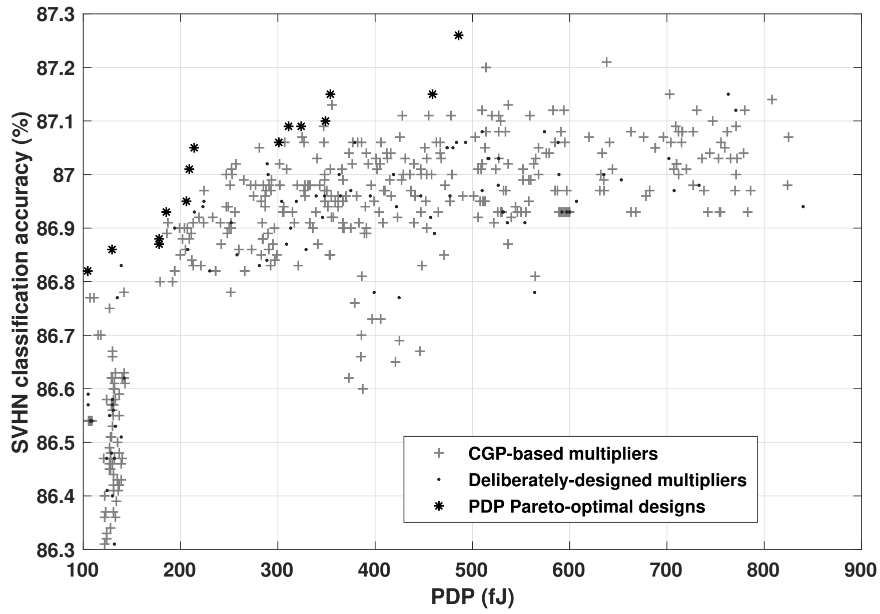
This section analyzes the error and hardware characteristics of approximate multipliers. Based on this analysis, a few designs that have a superior performance in both considered datasets are identified and recommended.

4.3.1 Error analysis

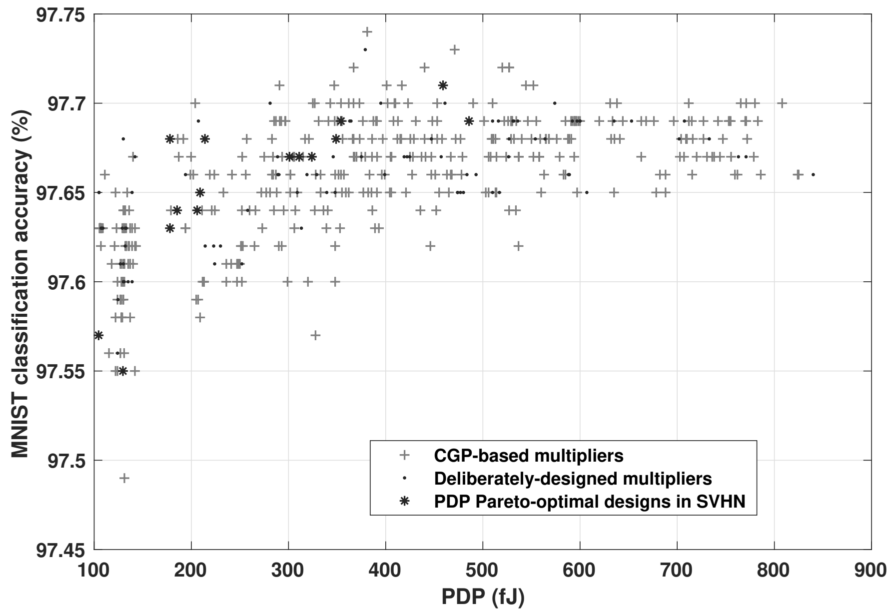
Fig. 4.5 compares "Class 0" and "Class 1" multipliers with respect to four important error features: **Var-ED**, **RMS-ED**, **Mean-AED**, and **Var-AED**. This plot shows how the "Class 1" and "Class 0" multipliers measure differently for the considered features. As shown in Fig. 4.5, "Class 1" multipliers generally have smaller **Mean-AED**, **Var-ED**, **Var-AED**, and **RMS-ED** values, when compared to "Class 0" multipliers. It also shows, in the zoomed-in insets, that some "Class 0" multipliers have smaller **Var-AED**, **RMS-ED**, **Mean-AED**, and/or **Var-ED** values than some "Class 1" multipliers is the reason why some multipliers are misclassified by the classifiers.

4.3.2 Hardware analysis

To further understand the quality of approximate multipliers, we performed a hardware analysis. The main hardware metrics of a multiplier, i.e. power consumption, area, and critical path delay, and PDP, are considered in this analysis. Note that all of the considered multipliers in this work are pure combinational circuits for which the throughput is inversely proportional (a.k.a. reciprocal function) to the critical path delay.

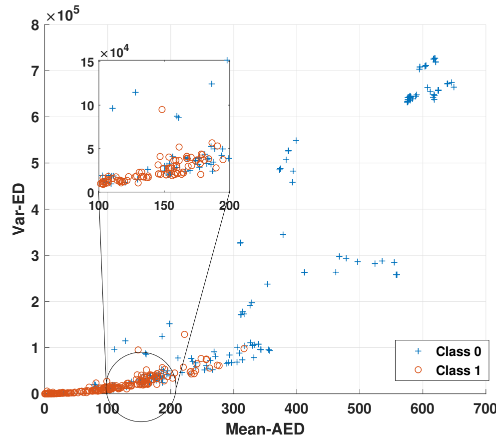


(a) Pareto-optimal design in PDP for the SVHN.

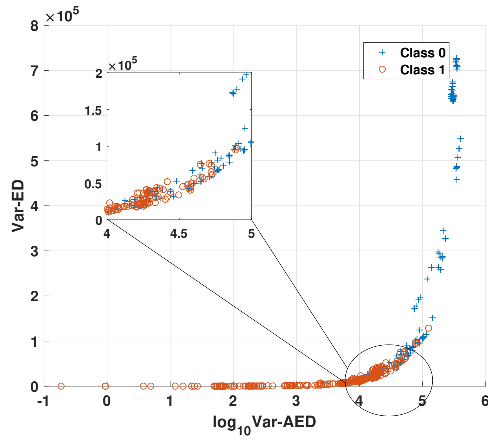


(b) Behavior of SVHN Pareto-optimal multipliers for the MNIST.

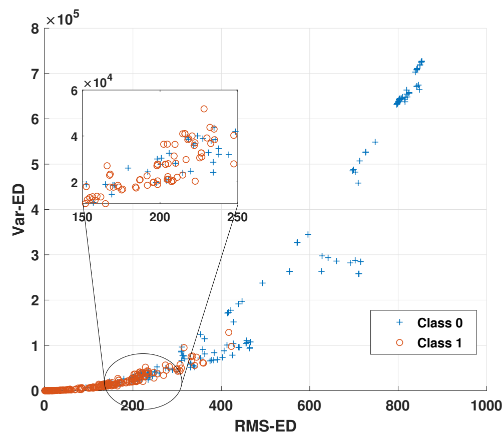
Figure 4.4: Neural network accuracy using the same approximate multipliers for different datasets.



(a) Var-ED vs. Mean-AED.



(b) Var-ED vs. log_{10}(Var-AED).



(c) Var-ED vs. RMS-ED.

Figure 4.5: Classification of Class 0 and Class 1 multipliers based on the most important features.

Fig. 4.6 shows two scatter plots that best distinguish the two classes of approximate multipliers are area vs. delay (Fig. 4.6(a)) and power consumption vs. delay (Fig. 4.6(b)). Note that only the results for the SVHN dataset are shown as the results for the MNIST are almost the same.

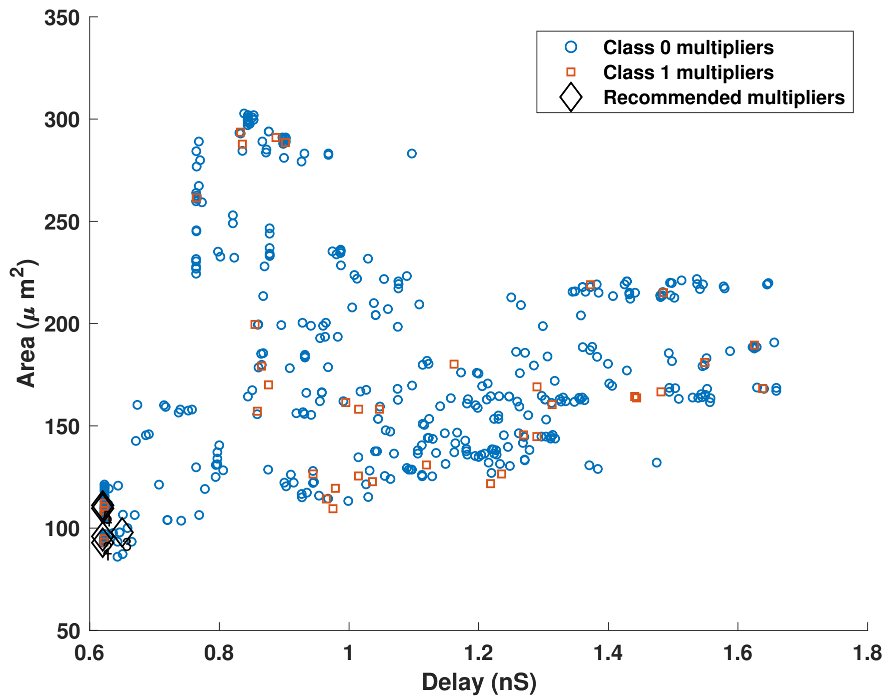
As the results in Fig. 4.6 show, unlike for the error metrics, there is no clear general trend in the hardware metrics. However, the designs with small delay and power consumption are preferred for NN applications, as discussed next.

As approximate multipliers are obtained by simplifying the design of an exact multiplier, more aggressive approximations can be used to further reduce the hardware cost and energy consumption. As previously discussed, some multipliers have almost similar accuracies, while as shown in Fig. 4.4, they have different hardware measures. The main reasons are as follows: (1) The hardware cost of a digital circuit totally depends on how it is implemented in hardware; e.g., array and Wallace multipliers are both exact designs and, therefore, they have the same classification accuracy. However, they have different hardware costs. (2) Classification accuracy of NNs is application-dependent and it depends on the network type, the dataset, learning algorithm, and the number of training iterations.

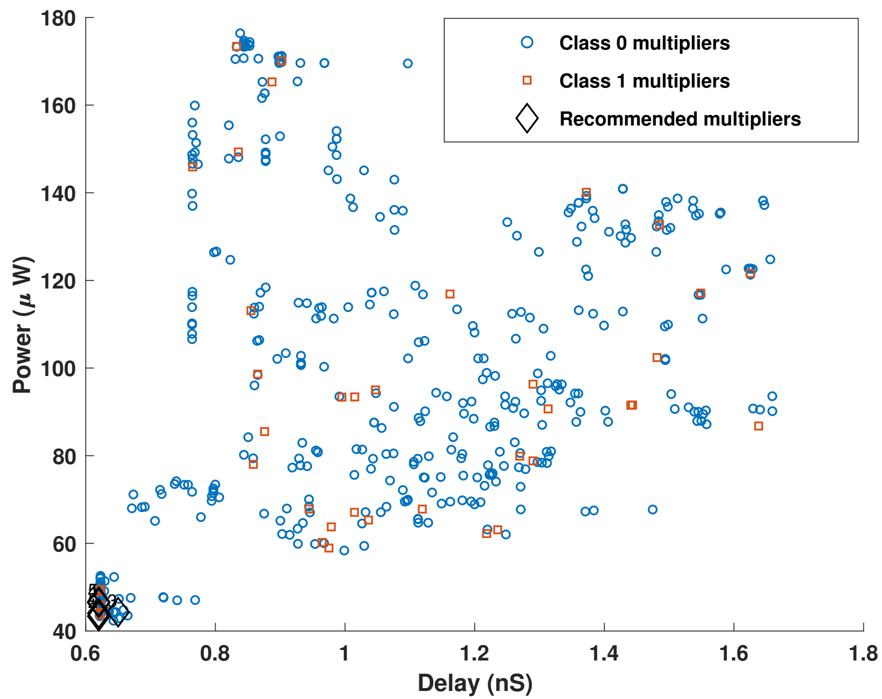
4.4 Recommended Approximate Multipliers

This subsection identifies a few approximate multipliers that have superior performance for both considered datasets. We chose the five best approximate multipliers that produce better accuracies than exact multipliers when used in the two considered NNs: the MLP for the MNIST dataset and LeNet-5 for the SVHN dataset. Note that these five designs were selected and sorted based on their low PDP values.

Table 4.5 lists and Fig. 4.6 shows these multipliers. Their Verilog, C, and MATLAB descriptions can be found online from [64]. Table 4.5 also reports the main hardware characteristics of these designs, i.e. the area, power consumption, delay, and PDP. The results in Table 4.5 indicate that all five chosen approximate multipliers (which are all CGP-based approximate multipliers)



(a) Area vs. delay for Class 1 and Class 0 approximate multipliers.



(b) Power vs. delay for Class 1 and Class 0 approximate multipliers.

Figure 4.6: Hardware comparison between Class 0 and Class 1 approximate multipliers.

Table 4.5: Hardware characteristics of the five best approximate multipliers.

Multiplier	**Area** (μm^2)	**Power** (μW)	**Delay** (nS)	**PDP** (fJ)
Exact	290.98	176.90	0.92	162.74
mul8-350	92.86	43.37	0.62	26.97
mul8-439	95.96	44.03	0.62	27.40
mul8-120	97.92	44.30	0.65	28.62
mul8-183	109.67	46.45	0.62	28.92
mul8-134	111.14	46.69	0.62	29.07

Table 4.6: Error characteristics of the five best approximate multipliers.

Multiplier	**ER**	**VAR-ED**	**RMS-ED**	**Accuracy (%)**	
				MNIST	**SVHN**
Exact	0	0	0	97.69	86.93
mul8-350	99.0	1.246e+4	123.0	97.70	**87.00**
mul8-439	97.8	4.500e+4	275.5	97.71	86.96
mul8-120	98.5	3.954e+4	217.3	97.70	**87.00**
mul8-183	97.2	1.334e+4	135.6	97.70	86.98
mul8-134	93.9	0.768e+4	111.1	**97.72**	86.95

consume less power (at least 73%) than the exact multiplier, while providing slightly higher accuracies (up to 0.18% more) when they are used in NNs. Comparing the average area and PDP shows a significant saving in hardware cost (i.e. 65.20% and 81.74% less area and PDP, respectively) by replacing the exact multipliers with the approximate ones.

The accuracies of the five recommended multipliers when employed in the two NN workloads are reported in Table 4.6. Although not an important error feature, the ER is shown in Table 4.6, together with VAR-ED and RMS-ED, which are two critical error features for the performance of an approximate multiplier in NNs. The results show that the five recommended multipliers all have small VAR-ED and RMS-ED values.

Hardware descriptions (in Verilog) of all of the CGP-based approximate multipliers can be found online in [64]. By using the Verilog code, one can easily obtain the truth table and/or the logic circuit for each design.

VAR-ED and RMS-ED, as the two most critical error features for the performance of an approximate multiplier in NNs, are also given in Table 4.6. The results show that the five recommended multipliers all have small

Table 4.7: Hardware characteristics of an artificial neuron implemented using recommended approximate multipliers.

Multiplier used in the neuron	**Energy**(fJ)	**Area**(μm^2)
Exact	944.08	956.02
mul8-350	269.48	367.52
mul8-439	438.29	475.72
mul8-120	553.68	599.10
mul8-183	631.76	561.40
mul8-134	801.73	583.92

VAR-ED and RMS-ED values, which is consistent with the results in Fig. 4.5.

An artificial neuron was also implemented using the five recommended approximate multipliers to replace the exact ones. The implemented neuron has three inputs and an adder tree composed of two adders to accumulate the three multiplication products. This is a widely-used technique for the performance analysis of multipliers in NNs [26].

The hardware characteristics for the implemented neuron are given in Table 4.7. The results show that the neurons constructed using the recommended multipliers can be up to 71.45% more energy-efficient than the neuron that uses the exact multiplier while being 61.55% smaller than it.

4.5 Summary

This chapter described the evaluation of a large pool of approximate multipliers, which contained 100 deliberately-designed and 500 CGP-based multipliers, for application in NNs. The exact multipliers in two benchmark networks, i.e. one MLP and one CNN (LeNet-5), were replaced after training with approximate multipliers to see how the classification accuracy is affected. The MLP and the CNN were employed to classify the MNIST and the SVHN datasets, respectively. The classification accuracy was obtained experimentally for both datasets for all 600 approximate multipliers.

The features in an approximate multiplier that tend to make it superior to others with respect to NN accuracy were identified and then used to build a predictor that forecasts how well an approximate multiplier is likely to work in

a NN. This predictor was verified by classifying 114 approximate multipliers based on their performance in LeNet-5 and AlexNet CNN for the SVHN and ImageNet dataset, respectively.

The major findings of this chapter are as follows:

- NNs that use appropriate approximate multipliers can provide higher accuracies compared to NNs that use the same number of exact multipliers. This is a significant result since it shows that better NN performance can be obtained with significantly lower hardware cost while using approximation.

- It appears that using approximate multipliers adds small inaccuracies (i.e., approximation noise) to the synaptic weights and this noise helps to mitigate the overfitting problem and thus improve NN accuracy.

- The most important features that make a design superior to others are the variance of the error distance (Var-ED) and the root mean square of the error distance (RMS-ED).

Although the statistically most relevant and critical features of approximate multipliers are identified in this work, a statistically accurate predictor based on those features cannot guarantee that the best approximate design will be identified: ensuring the best choice of approximate multiplier requires application-dependent experimentation.

Chapter 5

Logarithmic Multiplier and Squaring Circuits

This chapter proposes an energy-efficient leading-one detector (LOD) to speed up and improve the hardware efficiency of approximate logarithmic arithmetic circuits. The main drawback of the logarithmic multipliers that use LODs, like the Mitchell-based multipliers, is that they always underestimate the actual multiplication product. This may cause problems in iterative and repetitive applications, where the errors would accumulate. Hence, a nearest-one detector (NOD) is proposed and used in this chapter to design a logarithmic multiplier that has a more convenient double-sided error distribution. In addition, a logarithmic squaring circuit is proposed and evaluated in this chapter. Although the squaring function can be implemented with a multiplier, the frequent use of squaring in DSP applications led us to investigate it as an independent design.

5.1 Fast and Low-Power Leading-One Detector for More Energy-Efficient Logarithmic Multipliers

In a logarithmic number system (LNS), the binary logarithm of the input operand is computed using a LOD. Then the required operations (either shift or addition/subtraction) are performed more economically in the LNS and, finally, the antilogarithm of the result is computed. The common element in all

logarithmic arithmetic units, which plays a significant role in the performance of the system, is the LOD [53], [82]. Hence, a novel energy-efficient and fast LOD is proposed in this section.

5.1.1 The proposed LOD design

The main idea of the proposed exact LOD is to use an exact n-bit LOD to find the position of the leading one of a $2n$-bit number. The $2n$-bit input number N is divided into two halves, i.e. the more significant half ($N_H = N_{2n-1:n}$) and the less significant half ($N_L = N_{n-1:0}$). An OR gate tree is used to find out whether or not there is a '1' in N_H. The two possible scenarios arise:

- There is a '1' in N_H. In this case, N_H is the input to an exact n-bit LOD. Since the output bit width of a conventional LOD for a $2n$-bit number is $2n$ bits, n zeros can be immediately appended to the LSB lower half of the one-hot encoded result.

- There is no '1' in N_H. In this case, N_L is input to an exact n-bit LOD. As in the previous case, n zeros are appended to the MSB upper half of the result.

The block diagram of the proposed LOD is shown in Fig. 5.1. Note how 16 MSBs are evaluated using an OR gate tree and the final result, i.e. signal *sel*, is used as the select line of a multiplexer. Similar to the scaling technique, *sel* = '1' means that there is at least one '1' among the 16 MSBs (N_H) and, therefore, N_H is the input to the 16-bit LOD. Otherwise, N_L is used as the input to the 16-bit LOD. Note that in Fig. 5.1, the symbol '&' denotes vector concatenation and not logical bit-wise AND.

Although the proposed LOD is only evaluated in a logarithmic multiplier in this chapter, it is applicable to all of the other logarithmic arithmetic units that use a LOD, such as logarithmic squaring and square root functions and logarithmic dividers.

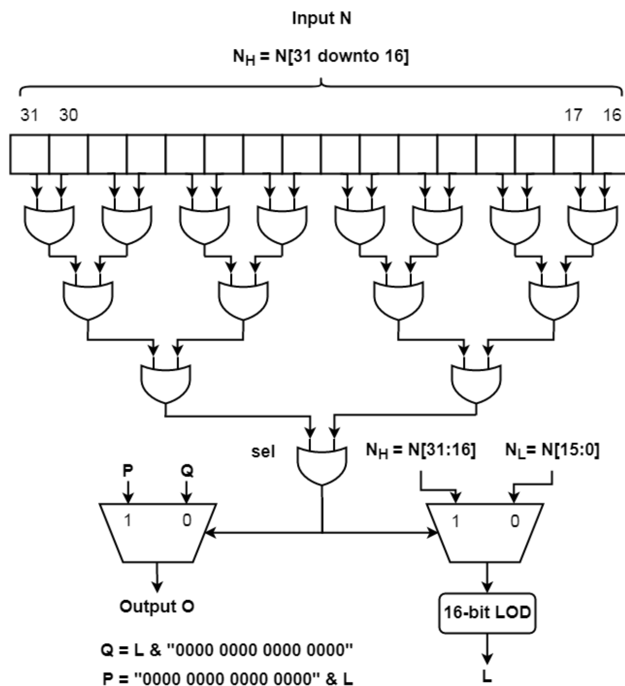


Figure 5.1: Using a 16-bit LOD to find the position of the leading one in a 32-bit number.

Table 5.1: Hardware metrics of five different logarithmic squaring functions.

Multiplier type	**Power** (mW)	**Delay** (nS)	**Area** (μm^2)	**PDP** (fJ)
Conventional Mitchell [47]	0.74	4.25	2257.21	3.14
Mitchell with proposed LOD	0.56	2.96	2173.66	1.65

5.1.2 Hardware analysis of the proposed LOD

This section provides the hardware metrics for the baseline Mitchell multiplier and the Mitchell multiplier with the proposed LOD. Both designs were implemented using the VHDL hardware description language in Vivado and then synthesized using the Synopsys Design Compiler for ST Micro's 28-nm CMOS process. The default settings for Design Compiler were used for all of the simulations to ensure a fair comparison.

The conventional LOD in the Mitchell multiplier was replaced with the proposed design. The predicted values for four key metrics — area, critical path delay, power consumption, and PDP — were then extracted.

As shown in Table 5.1, the Mitchell Multiplier with the proposed LOD is

30.35% faster than the conventional Mitchell multiplier. In terms of area, the proposed LOD results in a smaller multiplier than the standard Mitchell multiplier. Finally, with respect to the energy consumption, the Mitchell multiplier with the proposed LOD reduces the PDP by a factor of almost 0.5×.

5.2 An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing

The proposed logarithmic multiplier in this section can be used as a more accurate baseline design instead of the Mitchell approach. Moreover, the existing techniques in the literature for improving the accuracy of the Mitchell method are also applicable to the proposed method.

5.2.1 Proposed approximation approach

The positive integer N expressed as in (2.8) can be also represented as:

$$N = 2^{k+1}(1-y), \tag{5.1}$$

where $0 < y \leq 1$.

The conventional logarithmic approximation uses 2^k for the given number $2^k \leq N < 2^{k+1}$ (by using a LOD). Instead, we propose the approximation given in Algorithm 1. Let d_1 and d_2 denote the differences $N - 2^k$ and $2^{k+1} - N$, respectively. As shown in Algorithm 1, when $d_1 < d_2$ we underestimate the value of $log_2 N$ as k; otherwise, we overestimate it as $k+1$.

Algorithm 1 Proposed approximation for computing $log_2 N$ given N

```
1:  N = 2^k(1 + x) = 2^(k+1)(1 − y)
2:  d1 = N − 2^k                         ▷ error in underestimate
3:  d2 = 2^(k+1) − N                     ▷ error in overestimate
4:  if d1 < d2 then                      ▷ use underestimate
5:      x = d1/2^k
6:      log2N ≈ k + x
7:  else                                 ▷ use overestimate
8:      y = d2/2^(k+1)
9:      log2N ≈ k + 1 − y
10: end if
```

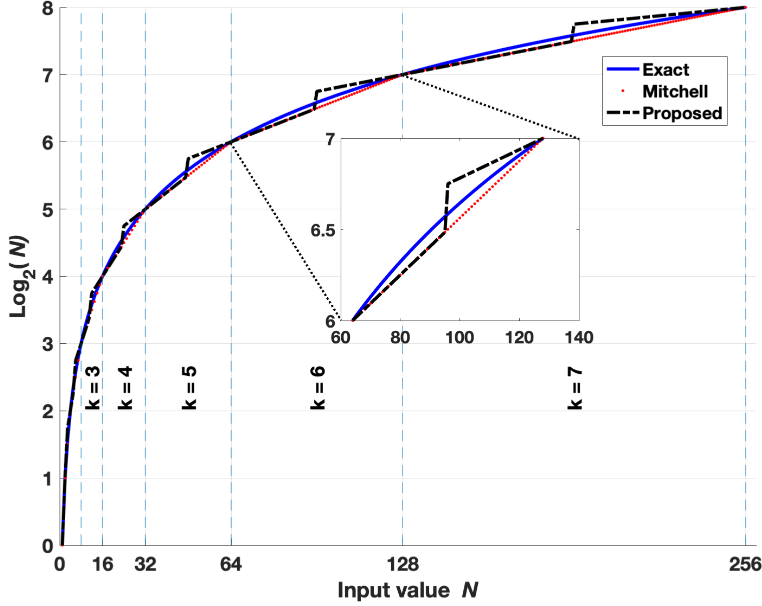


Figure 5.2: Approximation of log_2N.

The exact, Mitchell, and the proposed methods for computing log_2N are plotted in Fig. 5.2. The k parameter values corresponding to the N values, obtained from Algorithm 1, are also shown (between the vertical dotted lines) in Fig. 5.2. Only the intervals for $k \geq 3$ is shown in order to keep the figure easy to read.

The proposed approximation results in more than 6× smaller average error (over the integer range [1, 255]) than the Mitchell method (0.0088 vs. 0.0568), which is due to the double-sided error distribution of the proposed approach.

To further illustrate the error behavior of the proposed method, we compare the mean square error (MSE) values of the proposed and the Mitchell approaches. Mitchell uses the approximation $log_2(1+x) \approx x$. On the other hand, according to Algorithm 1, log_2N can be approximated as:

$$log_2N \approx \begin{cases} k+x, & for\ N = 2^k(1+x), \\ k+1-y, & for\ N = 2^{k+1}(1-y). \end{cases} \tag{5.2}$$

Given the approximated values for log_2N, the MSE_M for the Mitchell method can be calculated as:

$$MSE_M = \frac{1}{8} \times \sum_{k=0}^{7} \left[\frac{1}{2^k} \times \sum_{i=0}^{2^k-1} \left(log_2(1 + \frac{i}{2^k}) - \frac{i}{2^k} \right)^2 \right]. \tag{5.3}$$

The summation over k is provided to cover the entire input range for an 8-bit design. To calculate the MSE for the proposed approach we need to divide the input domain into two intervals. In the first interval, the input operand is closer to the largest power of two smaller than or equal to it. In the second interval, on the other hand, the input operand is closer to the smallest power of two that is larger than it. This can be done for the MSE_P as:

$$\begin{aligned} MSE_P = \frac{1}{8} \times \sum_{k=0}^{7} \left[\frac{1}{2^k} \times \left[\sum_{i=0}^{2^{k-1}-1} \left(log_2(1 + \frac{i}{2^k}) - \frac{i}{2^k} \right)^2 \right.\right. \\ \left.\left. + \sum_{i=2^{k-1}}^{2^k-1} \left(log_2(\frac{2^k + i}{2^{k+1}}) - \frac{2^k - i}{2^{k+1}} \right)^2 \right] \right]. \end{aligned} \tag{5.4}$$

The mathematical proofs for (5.3) and (5.4) are provided in Appendix A.1 and A.2, respectively.

5.2.2 Improved logarithmic multiplier design

High-level description of the ILM design

The proposed ILM first transforms the multiplicand A and multiplier B to the closest powers of two plus an additional term, as given by:

$$A = m_1 + q_1, \tag{5.5}$$

$$B = m_2 + q_2, \tag{5.6}$$

where $m_1 = 2^{k_1}$ and $m_2 = 2^{k_2}$. Hence, the product $A \times B$ can be approximated as:

$$A \times B \approx (2^{k_1+k_2} + q_2 2^{k_1} + q_1 2^{k_2}) + \cancel{q_1 q_2}. \tag{5.7}$$

As shown in (5.7), the three most significant terms are all multiples of powers of two that can be easily implemented as left-shift operations in hardware. In this design, the least significant term ($q_1 q_2$) is ignored and left as the

approximation error. A more detailed description of the ILM is provided in Algorithm 2, where NOD, PE and DEC denote the nearest-one detector, the priority encoder, and the decoder, respectively. Detailed descriptions of these three components are given in the following subsection.

Algorithm 2 Proposed logarithmic multiplication

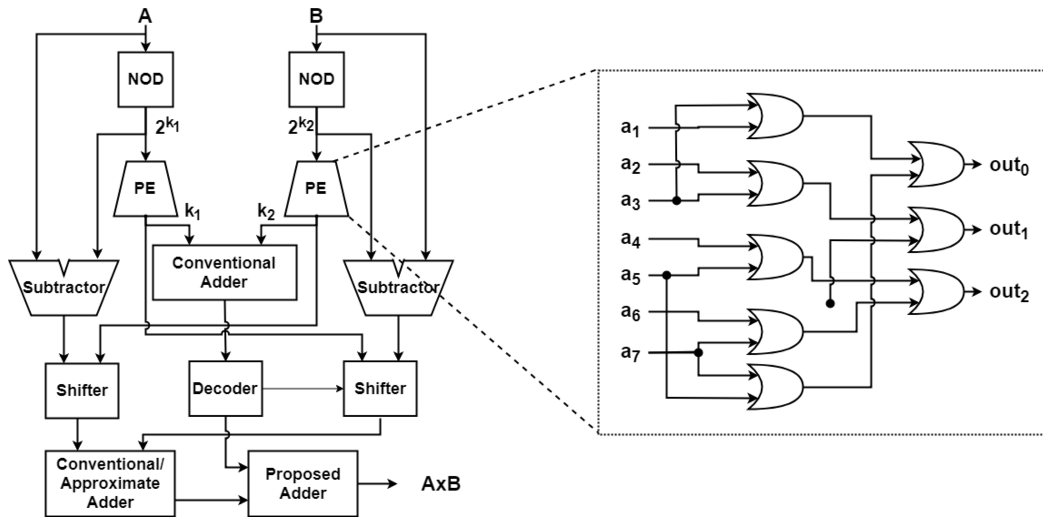
```
1:  procedure M(A, B)
2:      A, B: inputs, γ: approximate output
3:      m_1 ← NOD(A),
4:      k_1 ← PE(m_1),
5:      q_1 ← A − m_1,                          ▷ for steps 3-5 see (5.5)
6:      m_2 ← NOD(B),
7:      k_2 ← PE(m_2),
8:      q_2 ← B − m_2,                          ▷ for steps 6-8 see (5.6)
9:      q_1 2^{k_2} ← q_1 << k_2,
10:     q_2 2^{k_1} ← q_2 << k_1,
11:     2^{k_1+k_2} ← DEC(k_1 + k_2),
12:     γ ← 2^{k_1+k_2} + q_2 2^{k_1} + q_1 2^{k_2}.      ▷ see (5.7)
```

Hardware implementation

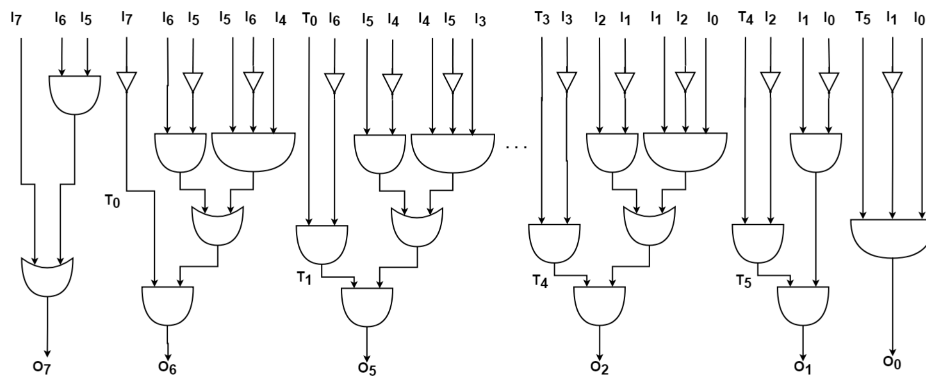
The ILM can be implemented by either: (1) implementing the logic to calculate the nearest powers of two, or (2) using a look-up table (LUT). We decided not to use LUTs as that would increase the memory usage, which is often a serious bottleneck for neural networks applications [26], [83].

The block diagram of the ILM is given in Fig. 5.3(a). The NOD circuits (Figs. 5.3(b) and 5.3(c)) are based on a leading-one detector (LOD) circuit. However, unlike the LOD, the NODs find the nearest power of two to a given input. Similar to some existing LODs [82], [84], the proposed NODs evaluate from the MSB to the LSB.

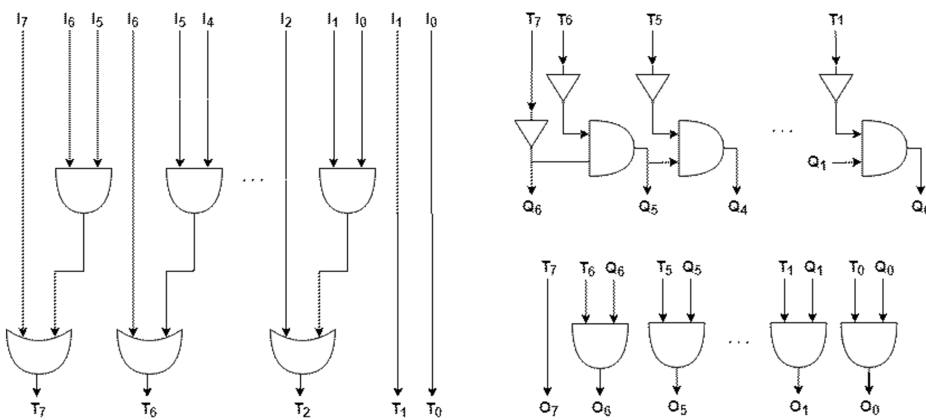
The priority encoder (PE) then determines the number of required shifts based on the NOD's output. The two residue terms q_1 and q_2 are also calculated and shifted according to the k_2 and k_1 values, respectively, and a decoder generates the most significant term, $2^{k_1+k_2}$. Finally, the three resulting terms are summed up to obtain the approximate product. For hardware savings, we used the PE proposed in [50].



(a) Block diagram of the ILM and the priority encoder [50].



(b) Proposed nearest-one detector (NOD) Circuit I.



(c) Proposed nearest-one detector (NOD) Circuit II.

Figure 5.3: The proposed improved logarithmic multiplier (ILM) design.

Figs. 5.3(b) and 5.3(c) depict the design of the two proposed NODs, where I and O are the primary input and output signals, respectively. Normally, nine bits are needed to represent the nearest power of two to an 8-bit input. However, the proposed multiplier is evaluated for NN applications, where large synaptic weights are unlikely to appear and removing them would not significantly influence the performance of NNs. Hence, are designs are simplified by rounding down the output of the NOD to the largest power of two representable in 8 bits, i.e. 128. In other words, up-rounding is not performed if the nearest-power of two is greater than 128.

The NOD design in Fig. 5.3(c) is composed of two stages: (1) an up-rounding stage and (2) a leading one detector. The internal 8-bit signal $T[7:0]$, obtained from the design on the left side of Fig. 5.3(c), is the up-rounded version of the original input I. In fact, $T = I$ at all bit positions unless two successive bits in I, e.g. I_i and I_{i-1}, are '1'. In that case, the bit at one higher position in signal T, i.e. T_{i+1}, will be '1'. For example, the input signal $I =$ "00**11**0010" results in $T =$ "0**1**110010". Note that not all of the numbers require up-rounding. According to Algorithm 1, we only need to overestimate and round up if $d_1 \geq d_2$, i.e., when the closest power of two to the input is greater than the input number. The internal signal T is then used as the input to an LOD, as proposed in [50], to output the one-hot representation of the nearest power of two to the given input. In fact, the two circuits on the right side of Fig. 5.3(c) together form an LOD, as explained in [50].

In order to further improve the hardware efficiency, we also propose a novel adder. This adder is used in the final stage, i.e. the adder that produces $A \times B$ in Fig. 5.3(a). There are three inputs to this adder (i.e., $2^{k_1+k_2}$, $q_1 \times 2^{k_2}$, and $q_2 \times 2^{k_1}$, see step 12 in Algorithm 2), hence an adder tree composed of two adders is required. A conventional 8-bit ripple-carry adder (composed of conventional FAs) is used to add $q_1 \times 2^{k_2}$ and $q_2 \times 2^{k_1}$ and the proposed adder is used to add the result to the third term, $2^{k_1+k_2}$, see Fig. 5.3(a).

Note that the proposed adder is not an approximate design, however it has a simplified structure. Since $2^{k_1+k_2}$ is in a one-hot representation, the structure of the 8-bit adder is modified accordingly. The truth tables for both

Table 5.2: The proposed vs. the conventional full adder.

a	b	c_{in}	Conventional FA		Proposed FA	
			sum	c_{out}	sum	c_{out}
0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	1	0	1	0	1	0
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	1	N/A	N/A
1	1	0	0	1	0	1
1	1	1	1	1	N/A	N/A

Conventional

$$sum = (\overline{a}.\overline{b}.c_{in}) + (\overline{a}.b.\overline{c_{in}}) + (a.\overline{b}.\overline{c_{in}}) + (a.b.c_{in})$$
$$c_{out} = (a.b) + (a.c_{in}) + (b.c_{in})$$

Proposed

$$sum = (\overline{b}.c_{in}) + (\overline{a}.b.\overline{c_{in}}) + (a.\overline{b})$$
$$c_{out} = (a.b) + (b.c_{in})$$

Table 5.3: Hardware comparison of the conventional and proposed full adders.

Full adder	**Power** (μW)	**Delay** (nS)	**Area** (μm^2)	**PDP** (fJ)
Conventional	1.32	0.09	3.42	0.1188
Proposed	**0.59**	**0.08**	**2.28**	**0.0472**

the conventional and the proposed FAs are shown in Table 5.2. Note that the "not applicable" (N/A) entries in Table 5.2 cannot happen because there is only one '1' in one of the inputs. If input A is a one-hot number and the '1' is at bit position i, then it is not possible to have carry in from less significant positions.

The performance of the proposed adder is compared to the conventional FA and the results are given in Table 5.3. Both adders were implemented in VHDL and then synthesized using the Synopsys Design Compiler (DC) for ST Micro's CMOS 28-nm process. As shown, the proposed adder is 33.3% smaller and 60.27% more energy-efficient than the conventional full adder. This can significantly reduce the hardware implementation cost, as discussed in the next section.

To further improve the hardware efficiency, the "conventional adder 2" in Fig. 5.3(a) is replaced with an approximate adder. A modified SOA-k adder is used in which, instead of setting all of the k LSBs to '1', these bits are set alternatively to '1' and '0'Ḃy doing so, the resulting adder can either overestimate or underestimate the result. Therefore, the double-sided error distribution property in the proposed ILM is preserved.

5.2.3 Performance evaluation of the ILM

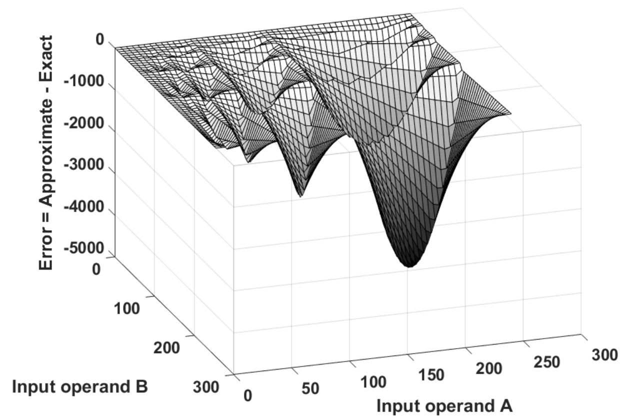
Accuracy metrics

The error for the product of $A = 2^{k_1}(1+x_1)$ and $B = 2^{k_2}(1+x_2)$ depends on k_1 and k_2, i.e. the intervals in powers of two into which the input operands fall. To illustrate this error, the difference between the exact and approximate products is plotted for two designs, the Mitchell and the proposed ILM multipliers, in Fig. 5.4 for integers $A, B \in [0, 255]$ to provide a better visualization of the error behavior. Notice how the error increases as k_1 and k_2 increase.

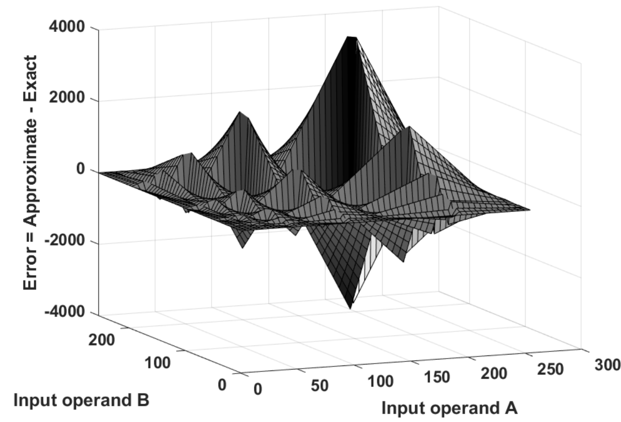
Fig. 5.4 shows the error behavior over the multiplier's entire input domain. However, it has been shown that the trained synaptic weights in NN applications do not have a uniform distribution and they are mostly centered around zero ([85] and [86]), where the multiplier is most accurate. On the other hand, the exact input distribution varies for a NN; it would depend on the application. Therefore, in Table 5.4 the accuracy metrics are reported for the two most common general distributions, the uniform and standard normal. To study the error distribution 10^6 input combinations were generated for multiplications using exact and logarithmic multipliers; the corresponding MRED, average error (AE), and the NMED [87] were then calculated. Note that the error distance is defined as the absolute difference between the exact and the approximate products, P_e and P_a, while the AE is calculated as:

$$AE = \frac{1}{N} \times \sum_{i=1}^{N} (Pa - P_e) \tag{5.8}$$

The parameter k in the ALM-SOA-k in Tables 5.4 and 5.5 indicates the number of LSBs to which approximation is applied. Similarly, IML-L indicates



(a) Error characteristics of Mitchell's multiplier. [50].



(b) Error characteristics of the proposed ILM multiplier.

Figure 5.4: Error visualization in LMs.

Table 5.4: Error metrics of the LMs for general input distributions.

Distribution	Multiplier Type	AE	MRED	NMED
Uniform	Mitchell [47]	589.71	0.0372	0.0091
	ALM-SOA-5[49]	561.48	0.0343	0.0087
	ALM-SOA-9[49]	305.68	0.0873	0.0076
	ILM-0	**0.25**	**0.0275**	**0.0068**
	ILM-5	28.03	0.0296	0.0068
	ILM-9	288.49	0.1069	0.0086
Normal	Mitchell [47]	76.29	0.0346	0.0012
	ALM-SOA-5[49]	53.24	0.0877	0.0010
	ALM-SOA-9[49]	183.66	1.6657	0.0035
	ILM-0	**5.24**	**0.0269**	**0.0008**
	ILM-5	19.26	0.0951	0.0010
	ILM-9	270.60	1.6982	0.0044

that L LSBs are approximated in the proposed ILM.

The results in Table 5.4 show that the proposed ILM-0 is the most accurate design with respect to all the considered error metrics, especially the MRED, for both input distributions. ILM-5, on the other hand, is the second most accurate design when the inputs are uniformly distributed. However, for a normal input distribution, the Mitchell and the ALM-SOA-5 multipliers perform better than ILM-5. Finally, ILM-9 has the worst error behavior as it approximates nine LSBs, as explained in Section IV.B.

The error metrics for the LMs when used in the two considered NN workloads are given in Table 5.5. Instead of assuming a general input distribution, as in Table 5.4, we performed all of the multiplications $x_i \times w_i$ in the NNs using LMs and calculated their error metrics accordingly.

The results in both Tables 5.4 and 5.5 show that the ILM-0 and ILM-5 are the most accurate of all the considered designs in terms of AE, MRED, and NMED for both the uniform and normal distributions of inputs and for the two application-specific NNs. Because the ER is generally high (more than 98% [49], [88]) for LMs due to the approximation in the base-2 logarithm and it does not give any insight as to how close the approximated result is to the exact one, the ER values are not reported here.

Table 5.5: Error metrics of the LMs for the two NN workloads.

NN Type	Multiplier Type	AE	MRED	NMED
MLP(784-128-10) MNIST dataset	Mitchell [47]	2704.39	0.4319	0.1992
	ALM-SOA-5[49]	2506.42	0.3004	0.1873
	ALM-SOA-9[49]	618.52	0.6267	0.1828
	ILM-0	**13.14**	**0.1193**	**0.0296**
	ILM-5	55.31	0.2539	0.0299
	ILM-9	602.43	0.4757	0.0933
Alexnet CIFAR-10 dataset	Mitchell [47]	583.89	0.0759	0.0389
	ALM-SOA-5[49]	577.58	0.0695	0.0385
	ALM-SOA-9[49]	825.81	0.1166	0.0659
	ILM-0	25.13	**0.0300**	0.0087
	ILM-5	**6.73**	0.0303	**0.0083**
	ILM-9	261.66	0.0869	0.0349

Table 5.6: Hardware metrics of the logarithmic multipliers.

Multiplier	Power (μW)	Delay (nS)	Area (μm^2)	PDP (fJ)
Mitchell [47]	66.26	1.42	281.2	94.09
ALM-SOA-5[49]	61.04	**1.39**	255.4	84.84
ILM-0 (NOD I)	53.72	1.68	287.4	90.25
ILM-5 (NOD I)	**50.37**	1.64	255.3	**82.61**
ILM-0 (NOD II)	65.27	1.83	285.9	119.44
ILM-5 (NOD II)	56.90	1.59	**239.9**	90.47

Hardware metrics

The hardware measurements for four key metrics are given in Table 5.6. The design in [89] was considered as the Mitchell multiplier as [47] does not detail any particular hardware implementation. Only the basic block in [89] was implemented, i.e. no iterations as iterative algorithms can be applied to any logarithmic design and they would significantly increase the hardware costs [50].

As shown in Table 5.6, the smallest designs are ILM-5 (using NOD II) and ILM-5 (using NOD I), which are, respectively, 14.68% and 9.21% smaller than the base Mitchell design while being almost 20% more accurate (see Tables 5.4 and 5.5). With respect to delay, the ALM-SOA and Mitchell multipliers are 17.98% and 15.49% faster than the proposed ILM-5 (with NOD I). However, the results in Table 5.6 show that the ILM-5 (NOD I) has the lowest PDP value

among all the considered designs. In term of power consumption, ILM-5 (NOD I) is the most efficient design, consuming 21.18% less power than its closest competitor, ALM-SOA-5. Note that from here on, the NOD I circuit type given in Fig. 5.3(b) is considered in the proposed ILM circuit unless otherwise noted.

The hardware complexity of the LMs, including the proposed design, increases almost linearly with the input size N, while that of the conventional multipliers increases almost quadratically with N [49]. Hence, more significant savings are expected when the proposed multiplier is extended to larger designs (e.g., 64-bit multipliers). However, as eight bits have been shown to be sufficient for NN applications [90]–[92], 8-bit multipliers are considered in this work.

5.2.4 Example application: neural networks

The proposed ILM can be used in various types of NNs; however, only the most common feed-forward NNs are studied in this work. Two NNs are considered to evaluate the LMs. The first one is an MLP that classifies the MNIST dataset and the other one is a CNN that classifies the CIFAR-10 dataset.

Neural network workloads

We used an MLP network with 784 input neurons (one for each pixel of the 28×28 monochrome image), 128 neurons in the hidden layer and 10 output neurons. The outputs are interpreted as the probability of classification into the 10 target classes of the digits 0 to 9 [62]. The MLP uses the soft-limiting sigmoid activation function.

The exact multipliers in the considered MLP are replaced with each of the LMs and the classification accuracy is evaluated. The resulting weights are plotted in Fig. 5.5 to determine if the trained synaptic weights are normally distributed. Since an 8-bit width is used for inference and the most significant bit is the sign bit, the trained synaptic weights are mapped into the range [-127, 127] in Fig. 5.5.

Moreover, AlexNet is used to classify the CIFAR-10 dataset [23]. AlexNet

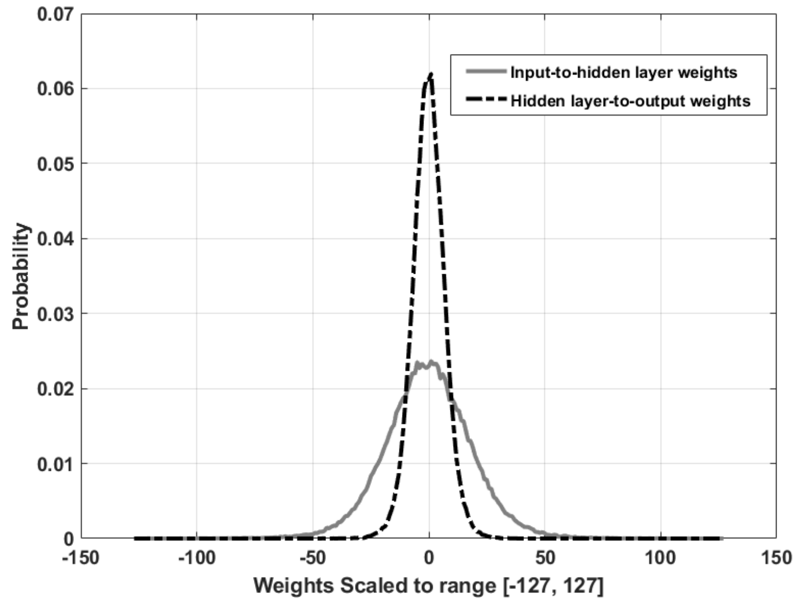


Figure 5.5: Probability distribution of the trained weights for the MLP, mapped into the range of [-127, 127].

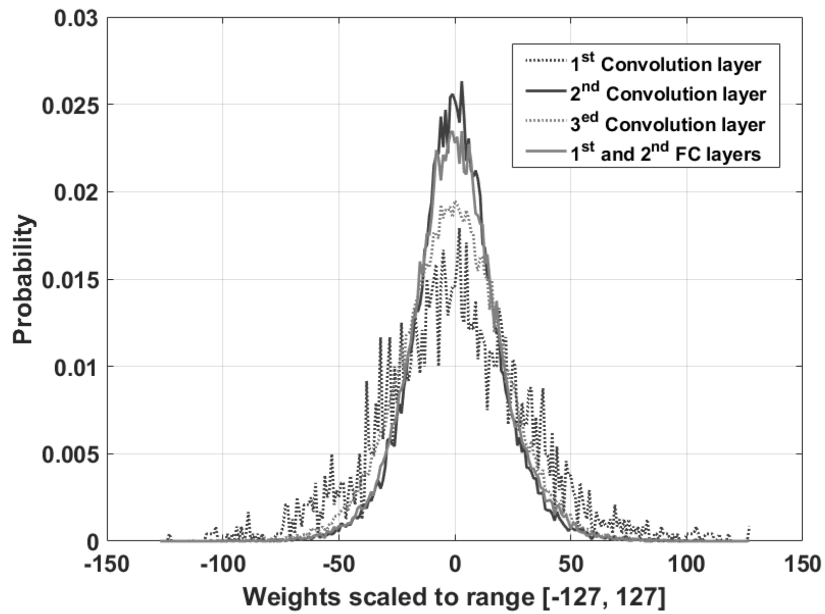


Figure 5.6: Probability distribution of the trained weights for Alexnet, mapped into the range of [-127, 127].

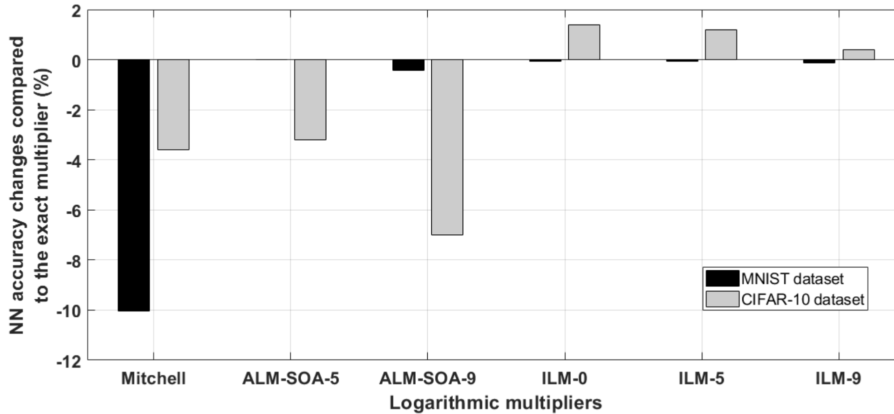


Figure 5.7: Comparison of classification accuracy of the MNIST and CIFAR-10 datasets with logarithmic multipliers.

is a CNN configuration that uses ReLU activation functions and has three convolutional layers, two fully-connected layers, max-pooling layers and average pooling layers. Similarly, the exact multipliers (in the convolutional and fully-connected layers) were replaced in succession with each type of the LMs and the resulting classification accuracy was evaluated for each version. The rectified linear units (ReLUs) is used as the activation function in the neurons in this network. An ReLU has an output of zero if the input is less than zero, otherwise the output is equal to the input [93]. The weights are also plotted in order to investigate the distribution of the trained synaptic weights. The weights are mapped into the range [-127, 127], as shown in Fig. 5.6.

Accuracy analysis

The classification accuracies for both the MNIST and the CIFAR-10 datasets using the various LMs are plotted in Fig. 5.7. Unlike previous studies, such as [8], [26], retraining has not been performed. The proposed designs that use ILM-5 and ILM-9 show only 0.08% and 0.12% accuracy degradation, respectively, for the MNIST dataset compared to the NN that use exact multipliers. The ALM-SOA-5, on the other hand, has the same accuracy as the NN with exact multipliers.

As also shown in Fig. 5.7, all three variants of the proposed ILM increase the classification accuracy for the CIFAR-10 dataset compared to the NN with

Table 5.7: Hardware characteristics of the artificial neuron when implemented using different types of LMs.

Multiplier used in the neuron	**Energy**(fJ)	**Area**(μm^2)
Mitchell [47]	93.85	1019.5
ALM-SOA-5[49]	68.23	915.1
ILM-0 (NOD I)	87.17	1004.8
ILM-5 (NOD I)	**53.32**	894.5
ILM-0 (NOD II)	87.48	995.1
ILM-5 (NOD II)	54.14	**884.7**

exact multipliers. Note that we can get up to 1.4% accuracy improvement by using ILM-0 instead of the exact multipliers. The double-sided signed error distribution and the low error magnitude of the proposed design help mitigate the overfitting issue in Alexnet. In fact, the double-sided errors with lower magnitudes effectively introduce noise into the proposed ILM. Hence, by using the ILM multiplier we are adding noise to the NN. It has already been shown that adding noise is often an effective way of improving the performance of NNs [65], [94]. This result indicates that higher classification accuracies can be obtained with less hardware costs.

Hardware analysis

The hardware characteristics of the LMs were discussed in Section V. In this section, an artificial neuron is implemented using different types of LMs to replace the conventional ones. The implemented neuron has three inputs and an adder tree composed of two adders to accumulate the three multiplication products. This is a widely-used technique for the performance analysis of multipliers in NNs [7], [26].

The hardware characteristics for the implemented neuron are given in Table 5.7. The results show that the neuron that use the proposed ILM-5 (NOD I) has the lowest energy consumption. It is 21.85% more energy-efficient than the neuron using ALM-SOA-5 while being 2.25% smaller. In terms of area, on the other hand, the neuron using ILM-5 (NOD II) is the smallest design.

Truncation needs to be considered in the implementation of the artificial neuron. 8-bit precision is used for each of the three inputs and their corre-

sponding synaptic weights and, therefore, the multiplication product would be 16-bit. However, since the output will be connected to another layer of neurons, truncation to 8 bits is required. The truncation is done by performing hard-limiting, i.e. using the maximum 8-bit number for all output values that need more than 8-bit precision.

5.3 Low-Power Approximate Logarithmic Squaring Circuit Design for DSP Applications

5.3.1 Proposed squaring function

Mathematical modeling

Any positive integer N, as expressed in (2.8), can be also factored as in (5.1). Considering (2.8) and (5.1), the base-2 logarithm of N can be expressed as:

$$\log_2 N = k + \log_2(1 + x) = k + 1 + \log_2(1 - y). \qquad (5.9)$$

Hence, $\log_2 N^2 = 2\log_2 N$ can be written as the summation of the middle and right expressions in (5.9), as given by:

$$\log_2 N^2 = 2k + 1 + \log_2(1 + x - y - xy). \qquad (5.10)$$

The variable y can be obtained as a function of x, i.e. $y = 0.5(1 - x)$, by considering the fact that both (2.8) and (5.1) represent the same value N. Solving for t and substituting the expression into (5.10) results in:

$$\log_2 N^2 = 2k + 1 + \log_2(0.5 + x + 0.5x^2). \qquad (5.11)$$

The least squares method is used to linearly approximate $\log_2(0.5 + x + 0.5x^2)$ in (5.11). This method chooses the coefficients so as to minimize the summed square of residuals. We used the MATLAB *Curve Fitting Toolbox* [95] for this purpose. The resulting best least squares linear fit over $0 \leq x < 1$ is:

$$\log_2(0.5 + x + 0.5x^2) \approx 1.975x - 0.8732. \qquad (5.12)$$

Hence the base-2 logarithm of N^2 can be approximated by replacing the log() function in (5.11) with (5.12). However, to simplify the hardware implementation, the constant 1.975 is rounded to 2, which is a simple left-shift in hardware. The approximation in (5.12) will not remain the best linear fit when the coefficient 1.975 is changed to 2 and, therefore, the other coefficient needs to be adjusted to minimize the approximation error. By trying different values, we found out experimentally that 0.1268 (the constant obtained by replacing (5.12) in (5.11)) needs to be changed to 0.039 to achieve the best MRED for the LESF. This matter is further discussed in the accuracy analysis of the proposed squaring function in Section 5.3.2. The proposed squaring function is therefore given by:

$$N^2 = 2^{\log_2 N^2} \approx 2^{2k+1.975x+0.1268} \approx 2^{2k+2x+0.039}. \tag{5.13}$$

Note that the coefficients that result in the minimum MSE for the approximation in (5.12) are 1.975 and -0.8732, according to the MATLAB *curve fitting* toolbox. However, changing the coefficients to what is used in (5.13) increases the MSE from 0.0026 to 0.0084. Although this increase is notable, the new MSE is still negligible. More importantly, using the modified coefficients significantly simplifies the hardware implementation and still results in a highly-accurate squaring function.

The signed relative error distance is plotted for the LESF and the baseline Mitchell squaring circuit in Fig. 5.8. This figure shows that unlike Mitchell, LESF has both positive and negative errors.

Hardware implementation

The form of the proposed squaring function in (5.13) does not imply any particular hardware implementation. To address this issue, 2^y, where $y = 2x + 0.039$ needs to be approximated. The least squares method is used again and the best linear fit over $0 \leq y < 1$ according to the MATLAB *Curve Fitting Toolbox* is $2^y \approx 0.9923y + 0.9471$. We modified the two coefficients and implemented $2^y \approx y + 1$ instead. Although this modification increases

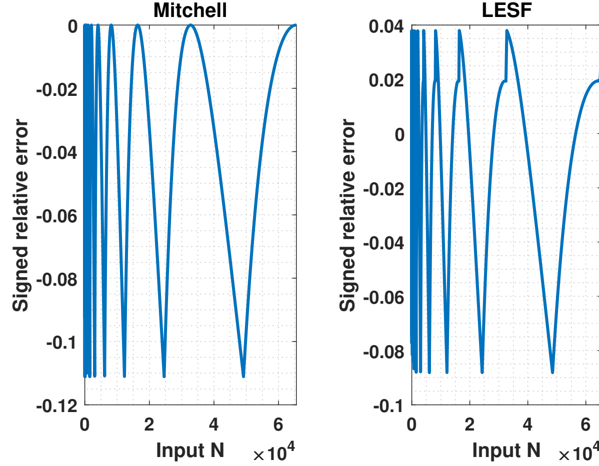


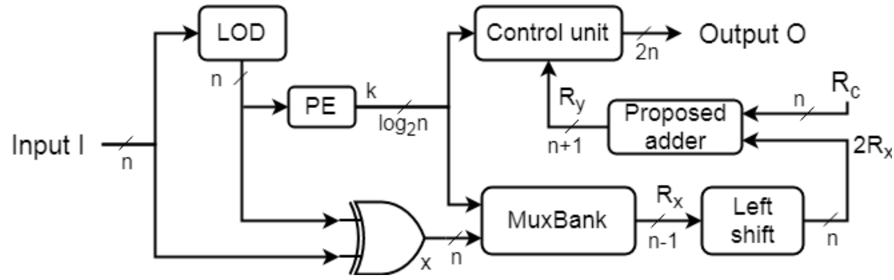
Figure 5.8: Signed relative error for the Mitchell and the LESF squaring circuits.

the MSE from 0.0007 to 0.003, it is still negligible. More importantly, it has a low-cost hardware implementation and results in a highly-accurate squaring function. Finally, LESF can be represented by:

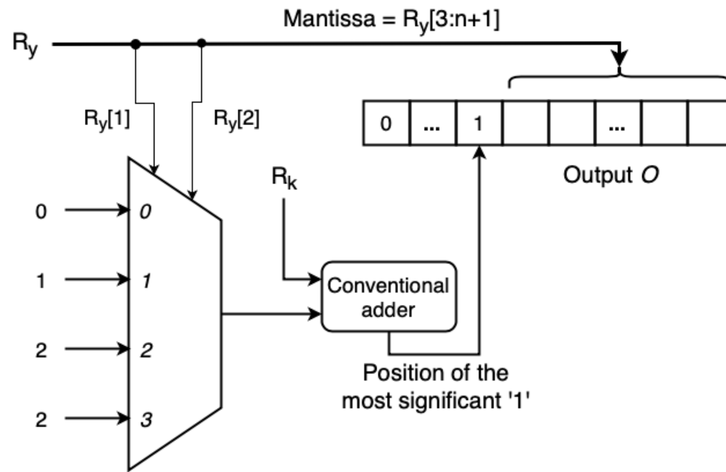
$$N^2 \approx \begin{cases} 2^{2k}(y+1), & y < 1, \\ 2^{2k+1}(y), & 1 \leq y < 2, \\ 2^{2k+2}(y-1), & 2 \leq y < 3. \end{cases} \tag{5.14}$$

For example for $2 \leq y < 3$ in (5.14), $2^y = 2^{(2+(y-2))}$; let $t = y - 2$ and, consequently, $2^y = 2^2 \times 2^t$. Since $0 \leq t < 1$, 2^y can be approximate as $2^2 \times (1 + t) = 2^2 \times (y - 1)$.

Fig. 5.9(a) shows the block diagram of the n-bit LESF. As shown in Fig. 5.9(a), the first step is to find the k and x values. We used the conventional Mitchell approach to find these two parameters. The n-bit output of the LOD is used as the input to the priority encoder (PE), which stores the value of k in $\log_2 n$ bits, R_k. The value of x, on the other hand, is obtained by performing the logical XOR between the original n-bit input I and the LOD's output. Since the output of the LOD uses a one-hot representation, performing the XOR operation does the subtraction [96]. The result of this subtraction needs to be represented in the $(n-1)$-bit register R_x [47]. If I_i, where $i \in \{0, 1, 2, ..., n\}$ is the most significant '1' in I, then $R_x = I_{i-1}I_{i-2}...I_1I_0$. Hence, zeros should



(a) Block diagram of the LESF.



(b) Control unit.

Figure 5.9: Architecture of the low-error squaring function LESF.

be padded to the least significant bits of R_x for $i < n$, e.g. $R_x = 00100000$ for $I = 00001001$. This is done by using multiplexers (*MuxBank* in Fig. 5.9(a)) that use the output of the PE and then append the proper number of zeros accordingly.

The next step is to calculate y, which can be done by adding the constant 0.039 to the shifted version of R_x, $2R_x$, which has the value $2x$. Since $2R_x$ is an n-bit number, the constant 0.039 needs to be represented as the n-bit value R_c. The result of this addition is stored in $(n+1)$-bit register R_y. We implicitly know that R_c contains the fraction part, which lies to the right side of the radix point and, therefore, it can be represented as $R_c = 000010100...00$.

Since R_c in the addition $R_y = 2R_x + R_c$ is a constant, a conventional adder can be replaced by a simpler design. The required function is specified below in Algorithm 3. In Algorithm 3, the two signals in pairs $(R_y[j+2], c_j)$,

Algorithm 3 Addition of $2R_x$ and $R_c = 0.039$

1: Inputs: $2R_x$ and R_c, Output: R_y
2: $R_y[9 : n+1] = 2R_x[8 : n]$
3: $R_y[8] = \neg 2R_x[7]$ ▷ logical NOT
4: $(R_y[7], c_5) = HA(2R_x[6], 2R_x[7])$ ▷ conventional HA
5: $(R_y[6], c_4) = HA(\neg 2R_x[5], c_5)$
6: $(R_y[5], c_3) = HA(2R_x[4], c_4)$
7: $(R_y[4], c_2) = HA(2R_x[3], c_3)$
8: $(R_y[3], c_1) = HA(2R_x[2], c_2)$
9: $(R_y[2], R_y[1]) = HA(2R_x[1], c_1)$

where $j \in \{1, 2, ..., 5\}$, denote the *sum* and $carry_{out}$ signals of a conventional half-adder (HA), respectively. According to Algorithm 3, more savings can be obtained by increasing n. In fact, the second step shows that no calculation required from the 8th bit down toward the least significant bit, i.e. the $(n+1)^{th}$ bit. Note that since this addition is done for the fraction part of the result, index p has the weight of 2^{-p} and, therefore, indexing starts from 1 (the most significant bit) and goes to $n + 1$.

The extra two bits in R_y compared to R_x are used to handle the three conditions in (5.14). As shown in (5.14), y represents the fractional part of the result and, thus, needs to be smaller than 1. Hence, adding two extra bits to the left side of the radix point lets us track the value of y and compare it to the conditions given in (5.14). Finally, the control unit calculates its output O based on the values of R_y and R_k. Fig. 5.9(b) shows the hardware implementation of the control unit. The first two most significant bits in R_y, i.e. $R_y[1]$ and $R_y[2]$, implement the conditions in (5.14). Based on the position of the most significant '1', which is determined by the output of the adder (a.k.a. the exponent) in Fig. 5.9(b), three cases can occur:

- The exponent is so small that there are not enough bit positions to store the $(n-1)$ bits of R_y. In this case, the less significant bits of R_y are discarded.

- The exponent is such that there are just $(n-1)$ bits left in O. In this case, the $(n-1)$ bits of R_y will exactly fit into the available bits in O, see Fig. 5.9(b).

- The exponent is too big to fit into the $(n-1)$ available free bits in O. In this case, after fitting the $(n-1)$ bit of R_y, the other bits are filled with zeros.

For further hardware savings, we used the PE proposed in [50]. This design exploits the fact that the output of the LOD uses a one-hot representation and, therefore, the conventional PE can be simplified. Regarding the LOD, the conventional LOD in [82] is used for all of the designs.

Note that LESF can be used as a more accurate baseline design instead of the Mitchell design. What is more, the existing techniques in the literature for improving the accuracy of the Mitchell design (e.g., the iterative technique in [96]) are also applicable to the proposed design.

5.3.2 Performance evaluation

We sought out competitive squaring circuit designs to permit a performance comparison. The designs in [47] and [49] are logarithmic multipliers, and not squaring circuits. We used the approximation methods in these two references and simplified their hardware implementation (e.g. by removing one of the two LODs in a logarithmic multiplier as there is only one input to a squaring circuit) to create comparable squaring circuits.

Note that there are other types of squaring functions in the literature, such as [97], [98]. Basically, any multiplier design can be simplified and used as a squaring circuit. However, only logarithmic designs were considered in this research. The performance of the squaring functions is evaluated below using both accuracy and hardware metrics.

Accuracy metrics

As mentioned in Section 5.3.1, the constant 0.039 in (5.13) was obtained by empirically trying different values. Fig. 5.10 shows how the accuracy of the LESF, in terms of the MRED, changes with different constant values. As shown in Fig. 5.10, reducing this constant from 1 improves the accuracy of the LESF. However, the minimum MRED, i.e. the maximum accuracy, is obtained

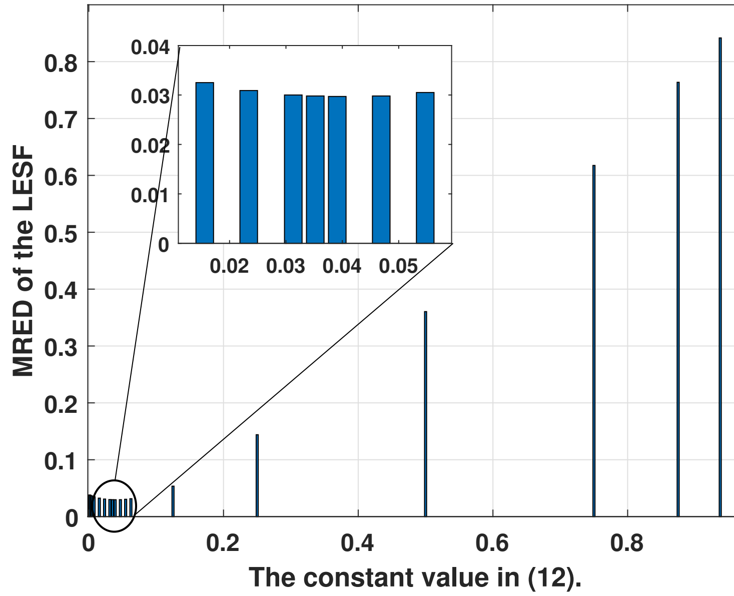


Figure 5.10: Accuracy of the LESF, with respect to the MRED, vs. the constant value in (5.13).

when the constant in (5.13) is in the range [0.035, 0.040], see the inset. Hence, we chose 0.039, which falls into this range and, as mentioned earlier, can be easily implemented in hardware.

All of the considered designs were implemented in MATLAB and their accuracy was evaluated over the entire 16-bit unsigned input domain, i.e. from 0 to 65535. Table 5.8 reports the accuracy metrics for the proposed LESF and other logarithmic squaring functions in the literature. The MRED and the average error AE were then calculated.

The results in Table 5.8 show that the approximate squaring function in [96] with one step of error correction, Sq. Fnc.-1, is the most accurate design, with respect to the MRED. However, using iterative steps for error compensation can be applied to other designs as well at the cost of significant additional hardware, see Section 5.3.2. Hence, Sq. Fnc.-1 aside, LESF is the most accurate design, being 21.39% more accurate than the next most accurate ALM-SOA-9 squaring function. With respect to the AE, LESF seems to be the best design, due to its double-sided error distribution.

Regarding the ALM-SOA squaring function, we tried different numbers of

Table 5.8: Error metrics of five different logarithmic squaring functions.

Squaring Function	**AE**	**MRED**
Mitchell [47]	5.09e+7	0.0384
ALM-SOA-9 [49]	4.87e+7	0.0374
Sq. Fnc.-0. [96]	2.05e+8	0.1137
Sq. Fnc.-1. [96]	2.91e+7	0.0149
LESF	1.44e+7	0.0297

Table 5.9: Hardware metrics of five different logarithmic squaring functions.

Squaring Function	**Power** (mW)	**Delay** (nS)	**Area** (μm^2)	**Normalized PDP×MRED**
Mitchell [47]	8.02e-2	2.12	291.96	0.65
ALM-SOA-9 [49]	7.12e-2	1.79	258.01	0.47
Sq. Fnc.-0. [96]	8.55e-2	1.07	264.22	1.00
Sq. Fnc.-1. [96]	1.95e-1	2.86	562.55	0.79
LESF	5.86e-2	0.81	247.57	0.13

approximation bits in the SOA adder and we found that 9 bits produced the lowest MRED.

Hardware metrics

The hardware measurements for three key metrics, area, critical path delay, and power consumption, are given in Table 5.9. As shown in this table, LESF is the most hardware efficient design, consuming 17.69% less power than the second most power-efficient design ALM-SOA-9 and being 24.29% faster than the second-fastest design Sq. Fnc.-0. As mentioned earlier, iterative techniques significantly increase the hardware cost and this is well reflected in the results of the Sq. Fnc.-1.

The hardware metric PDP and the accuracy metric MRED are multiplied as a single metric in the last column of Table 5.9. The normalized MRED-PDP product is a useful metric as it compares the squaring functions with respect to both hardware and error metrics. As shown in Table 5.9, LESF clearly has the best accuracy-hardware cost trade-off. Although ALM-SOA-9 is the second-best design, it is almost 3.5× less efficient than the LESF with respect to the PDP-MRED product metric.

5.3.3 Example application: square law detector

The AM demodulation using the square law detector algorithm was previously discussed in Chapter 2. Here we use a square-wave signal at 50 Hz for $m(t)$. Let $m(t)$ be any arbitrary message signal (we used a square-wave signal at $50Hz$). MATLAB was used to generate the message signal $m(t)$, the carrier signal $c(t)$ with $f_c = 1\,\text{KHz}$, and the modulated message $s(t)$.

Moreover, MATLAB *lowpass* function is used with the pass-band frequency $f_{pass} = 150\,\text{Hz}$ and a sampling frequency of $f_s = 10\,\text{KHz}$. With these inputs, the *lowpass* function generates a finite impulse response (FIR) filter of order 48. Note that the exact squaring function in (2.4) was replaced with the LESF and other logarithmic squaring functions.

Fig. 5.11 shows the demodulated messages $m'(t)$, according to which the LESF produces the closest waveform to the waveform generated by using the exact squaring function. Sq. Fnc.-1 is the second most accurate design. Finally, the waveforms generated by using the Mitchell and the ALM-SOA-9 squaring functions seem to be equally accurate, and worse than the other two designs.

To numerically compare the performance of the squaring functions, the Euclidean distance between the exact demodulated signal and those obtained by using logarithmic squaring functions were calculated and are reported in Table 5.10. The Euclidean distance $E_{A,B}$ between the two signals A and B measures the straight-line distance between two points in A and B and can be calculated as [99]:

$$E_{A,B} = \sqrt{\textstyle\sum_{i=1}^{S}(A_i - B_i)^2}. \tag{5.15}$$

where S is the number of sample points in the two signals and A_i and B_i denote the samples of the two signals A and B, respectively. According to the results in Table 5.10, the demodulated signal using the LESF is 67.19% closer to the demodulated signal using an exact squaring function compared to the second best design, Sq. Fnc.-1, which is consistent with the results in Fig. 5.11.

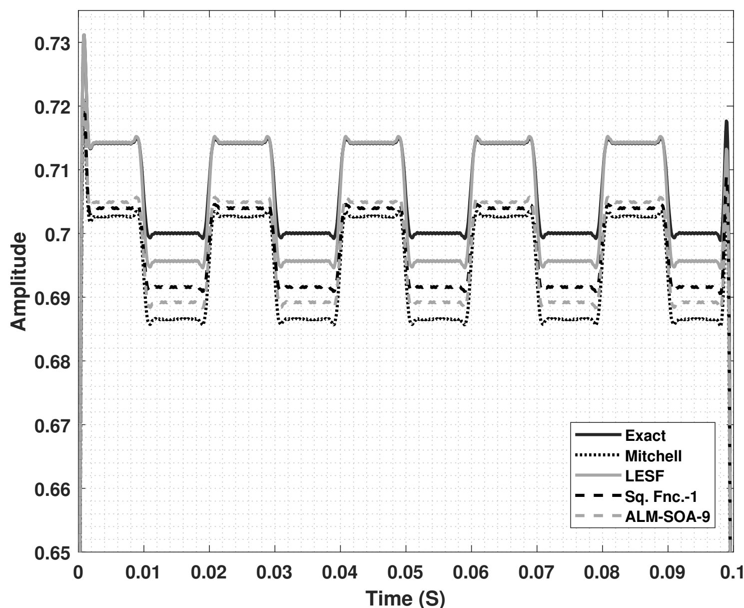


Figure 5.11: Comparison of the demodulated signal $m'(t)$ with exact and logarithmic squaring functions.

Table 5.10: Euclidean distance of five different logarithmic squaring functions.

Squaring Function	Euclidean distance
Mitchell [47]	0.3961
ALM-SOA-9 [49]	0.3185
Sq. Fnc.-0. [96]	2.0441
Sq. Fnc.-1. [96]	0.2960
LESF	0.0971

5.4 Summary

This chapter proposed an exact 16-bit LOD that is used to find the position of the leading-one in a 32-bit number N. The more significant half of N, N_H, is searched for ‘1’. If at least one ‘1’ is found, N_H is input to the 16-bit LOD III; otherwise, the less significant half of N, N_L, is used as the input to LOD. Compared with the original Mitchell multiplier, the proposed LOD reduces the PDP by 24.89% and operates 1.4× faster.

We also proposed a novel approximation method to efficiently compute $\log_2 N$. Using this method, the ILM is designed. The proposed ILM is more

accurate and has the smallest MRED values compared to other logarithmic designs in the literature. Two well-known NNs were considered as benchmark applications, for which the proposed designs show a higher classification accuracy than the other designs. The exact multipliers in both NNs were replaced with LMs and the ILM-5 resulted in the most energy-efficient NN structure. Interestingly, higher classification accuracies are obtained for the CIFAR-10 dataset by using the ILM compared to the use of exact (and other LM) multipliers.

Finally, a low-error squaring function, LESF, is proposed that outperforms the state-of-the-art designs in the literature. LESF is the most hardware efficient design, consuming the least amount of power and while being the fastest design. LESF is also more accurate than the existing designs, in terms of the MRED, except for Sq. Fnc.-1, which uses an iterative error compensation technique. However, this technique can be used to increase the accuracy of any logarithmic squaring function with significant extra hardware cost. The LESF was also shown to be almost 3$\times$ more accurate than the second most accurate design, Sq. Fnc.-1, with respect to the Euclidean distance metric in the square law detector application.

Chapter 6

Logarithmic Multiply-Accumulate Unit and Accelerators

Multiply-accumulate (MAC) units are widely used in the hardware implementation of numerically demanding applications such as machine learning (ML), digital signal processing (DSP), and optimization algorithms. Hence, designing more efficient MAC units can significantly improve the performance of the entire application. Four major considerations in the design of a MAC unit are the performance/computation speed, circuit size, energy consumption, and accuracy. Fortunately, in many ML and DSP applications, the accuracy can often be traded off with the other three design parameters [12], [100]. Hence, approximation techniques can be exploited to accelerate the MAC units, reduce their energy consumption, and make them smaller at the cost of often negligible degradation in the final output quality.

This chapter proposes the first logarithmic MAC (LMAC) unit. Multiplication, the more time-consuming part of the MAC operation, is converted into simple addition in the logarithmic domain and, therefore, the multiplication is accelerated. Novel linear approximations for logarithmic multiplication and addition are considered to simplify the MAC operation. The proposed LMAC benefits from a double-sided error distribution, which can increase the likelihood of error cancellation during the accumulation phase of many applications.

Furthermore, unlike the existing accelerators that simplify the multiplication/addition operations, this chapter proposes the first approximate accelerator that reduces the number of required multiplication operations. The proposed soft-dropping low-power (SDLP) architecture is specifically designed for convolutional neural networks (CNNs) and takes advantage of the spatial dependence between the input image pixels and skips some of the multiplications during the convolution operation and, thereby, reduces the energy consumption of the CNN's inference.

6.1 Design of a Fast and Energy-Efficient Approximate Logarithmic MAC Unit

6.1.1 Proposed logarithmic MAC (LMAC)

We implemented an approximate implementation of the sum of products $AB+CD$ and obtained significant savings in the energy consumption and area compared to the conventional exact implementation, while preserving the accuracy. Then the design is simplified to implement $AB+C$, which is a more common form of the MAC operation.

Mathematical Modeling

To implement $AB+CD$, each of the four inputs is factored as given by:

$$\begin{aligned} A &= 2^{k_1}(1+x_1) \\ B &= 2^{k_2}(1+x_2) \\ C &= 2^{k_3}(1+x_3) \\ D &= 2^{k_4}(1+x_4), \end{aligned} \tag{6.1}$$

where x_1, x_2, x_3, and x_4 all lie within the interval $[0, 1)$. Following (2.9) and by using (6.1), the base-2 logarithms of the four inputs A, B, C, and D (i.e., $a = \log_2 A$, $b = \log_2 B$, $c = \log_2 C$, and $d = \log_2 D$) can be calculated as:

$$\begin{aligned} a &= k_1 + \log_2(1+x_1) \\ b &= k_2 + \log_2(1+x_2) \\ c &= k_3 + \log_2(1+x_3) \\ d &= k_4 + \log_2(1+x_4). \end{aligned} \tag{6.2}$$

The least squares method can be used to linearly approximate $\log_2(1+x)$ in (2.9). This method chooses the coefficients so as to minimize the summed squares of the residuals over a relevant input interval. We used the MATLAB *Curve Fitting Toolbox* [95] for this purpose and the resulting best linear fit for the least squares over $0 \leq x < 1$ is:

$$\log_2(1+x) \approx 0.9877x + 0.0634. \tag{6.3}$$

Hence a, b, c, and d can be approximated by replacing the log functions in (6.2) with (6.3). However, to simplify the hardware implementation, the constant 0.9877 in (6.3) should be rounded up to 1. Note that, after changing 0.9877 to 1, the other coefficient needs to be updated to minimize the approximation error over the same input interval. By experimentally trying different values we found that 0.0634 should be changed to $0.0547{=}2^{-5}{+}2^{-6}{+}2^{-7}$ so that all terms are powers of two and can be easily implemented in hardware. Therefore (6.2) can be rewritten as:

$$\begin{aligned} a &\approx k_1 + x_1 + 0.0547 \\ b &\approx k_2 + x_2 + 0.0547 \\ c &\approx k_3 + x_3 + 0.0547 \\ d &\approx k_4 + x_4 + 0.0547. \end{aligned} \tag{6.4}$$

Changing the coefficients from (6.3) to what is used in (6.4) increases the root mean square error (RMSE) from 0.0255 to 0.0258. This increase in the RMSE is small and negligible in many applications.

Now, having found efficiently implementable approximations for a, b, c, and d, the partial products $P_1 = AB$ and $P_2 = CD$ can be calculated as:

$$\begin{aligned} P_1 &= 2^{(a+b)} = 2^{e_1} \\ P_2 &= 2^{(c+d)} = 2^{e_2}, \end{aligned} \tag{6.5}$$

where $e_1 = a + b$ and $e_2 = c + d$. Let e be the maximum of e_1 and e_2. $P = P_1 + P_2$ can then be calculated by:

$$P = 2^e(1 + 2^{-|e_1 - e_2|}). \tag{6.6}$$

Since e_1 and e_2 are not necessarily integers, it is easier to implement (6.6) in the logarithmic domain. The base-2 logarithm of P, p, is then given by:

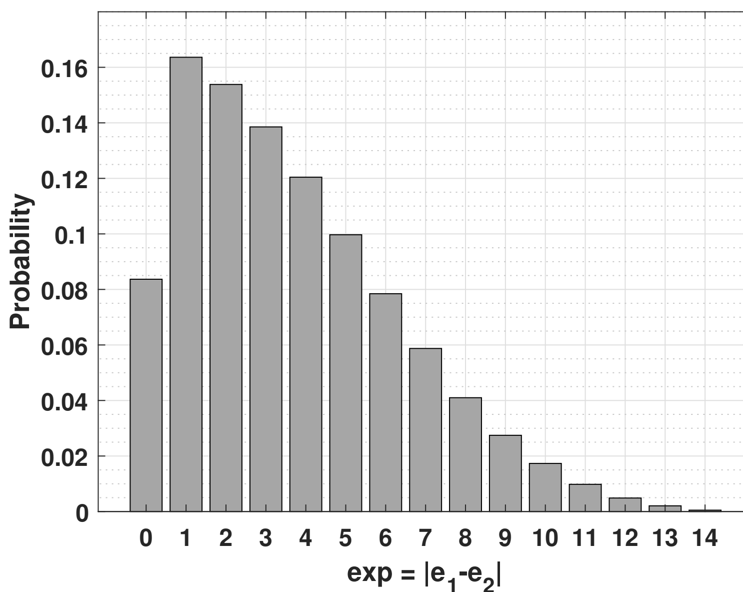


Figure 6.1: Distribution of $exp = |e_1 - e_2|$.

$$p = e + \log_2(1 + 2^{-|e_1 - e_2|}), \tag{6.7}$$

The term $t = 2^{-|e_1-e_2|}$ lies within $(0, 1]$ and, therefore, the linear approximation in (6.3) can be used to simplify $\log_2(1 + t)$ in (6.7). To do so, we need to calculate t, which is a challenging task. Thus the least squares method is used again to linearly approximate the entire function, i.e. $\log_2(1 + 2^{-|e_1-e_2|})$.

The distribution of the values of $exp = |e_1 - e_2|$ plays a significant role when finding the best coefficients of the linear approximation. Thus we experimentally computed the distribution of exp by generating 10 million uniform random quartets of inputs for A, B, C, and D. This distribution is shown in Fig. 6.1. Note that each of these four inputs is assumed to be an integer x in the range $0 \leq x \leq 255$. The value 255 is the maximum value of a pixel color intensity in many image file formats.

Selecting larger exp values covers a wider range of the entire input domain, however our simulation results in the MATLAB *Curve Fitting Toolbox* show that using a larger exp increases the approximation error. According to Fig. 6.1 almost 90% of the exp values are less than or equal to 7 and, therefore, the interval $0 \leq exp \leq 7$ is considered for our linear approximation. Using the

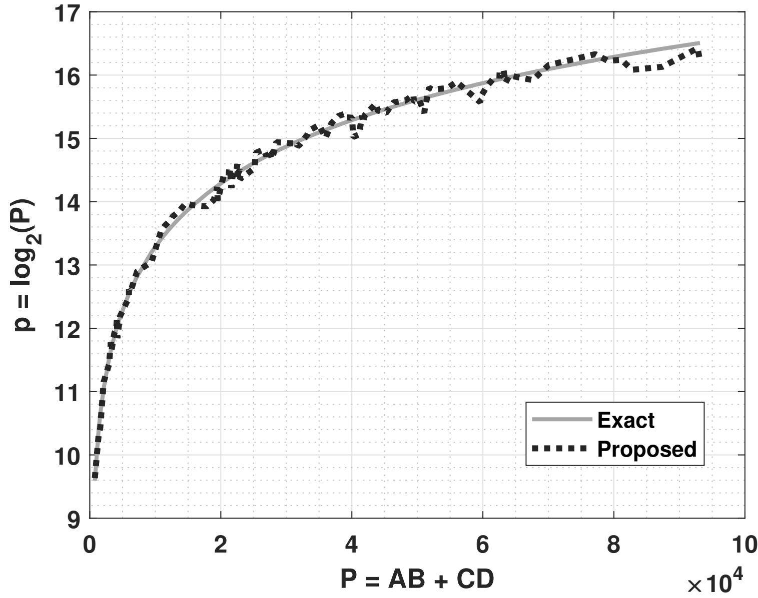


Figure 6.2: Comparison of the outputs of the exact and the LMAC in (6.9).

MATLAB *Curve Fitting Toolbox*, the best linear fit for approximating $\log_2(1+2^{-|e_1-e_2|})$ within the given range is found to be:

$$\log_2(1+2^{-|e_1-e_2|}) \approx -0.1178(|e_1-e_2|) + 0.655. \tag{6.8}$$

To simplify the hardware implementation, the coefficient 0.1178 is raised to 0.1250, which is a power of two. Clearly, by changing 0.1178 to 0.1250, using the original value of the second coefficient (0.6550) no longer guarantees the smallest RMSE. Thus we experimentally tried different values and determined that 0.6550 should be changed to $0.6875{=}2^{-1}{+}2^{-3}{+}2^{-4}$. All three of the required terms that sum up to 0.6875 are powers of two. This modification only slightly increases the RMSE of the original linear approximation in (6.8) from 0.1170 to 0.1183. Hence, the base-2 logarithm of the final product $P = AB + CD$, p, can be linearly approximated by:

$$p = e - \frac{1}{8}(|a+b-c-d|) + 0.6875. \tag{6.9}$$

Fig. 6.2 compares the proposed approximation in (6.9) with the exact value of p for 100 randomly selected inputs from the entire input domain.

The two important features of the proposed approximation method in (6.9) are: (1) the double-sided error distribution, which is evident from Fig. 6.2; and (2) the speed-up in calculating $AB+CD$. As opposed to other MAC units that have to wait for the partial products P_1 and P_2 to calculate the final result, the LMAC can directly calculate the final result once the base-2 logarithms of the four inputs are found.

Another common form of the MAC operation is to calculate $AB+C$ rather than $AB+CD$. Given that $AB+C$ is $AB+CD$ when $D=1$, the proposed design can be easily extended to calculate $AB+C$ as well. With $D=1$, the base-2 logarithm of D, $d=0$. Hence, (6.9) can be rewritten as:

$$p = e' - \frac{1}{8}(|a+b-c|) + 0.6875. \qquad (6.10)$$

where e' is the maximum of $(a+b)$ and c.

Note that choosing between $AB+CD$ and $AB+C$ depends on the available hardware and the most convenient form for the specific application. For example, let $M=[m_1, m_2]$ and $N=[n_1, n_2]$ be two 1×2 arrays of numbers. Then $Q = MN^T = \sum_{i=1}^{2} m_i n_i$, where N^T is the transposed array N, can be calculated either sequentially by using only one multiplier and one adder, or in parallel by using two multipliers and one adder. The multiplier and the adder in the sequential implementation are used twice and, therefore, it would be slower than the parallel implementation, while being more resource-efficient.

Hardware Implementation

To implement the approximation in (6.9), a 16-bit (i.e., half precision) FP number representation is used.

The first step is to find the base-2 logarithm of the input operands, by following (6.4). The k and x values in (6.4) are the values of the exponent and fraction fields of each of the four inputs, respectively. Once the k and x values in (6.4) are found, the base-2 logarithm of the final output p can be calculated using (6.9). Using the same analogy as in (6.1) and (6.4), the final product P can be represented as:

$$\log_2(P) = \log_2\bigl(2^{k_p}(1 + x_p)\bigr) \approx k_p + x_p + 0.0547. \tag{6.11}$$

Considering that (6.11) and (6.9) represent the same value, we obtain:

$$k_p + x_p = e - \frac{1}{8}(|a + b - c - d|) + 0.6328. \tag{6.12}$$

where k_p and x_p are the integer and the fraction parts of the right-hand side of (6.12), respectively.

Fig. 6.3 shows the block diagram of the resulting implementation of LMAC. Note that k_p and x_p fill the exponent and fraction fields of the final output. In fact, Fig. 6.3 shows the hardware implementation of (6.12). The two inputs a and b are added by using a conventional exact adder to generate e_1. Similarly, inputs c and d are added to generate e_2. Then a $\max(e_1, e_2)$ unit compares e_1 and e_2 and outputs the greater input value as e. The constant 0.6328 is added to e in the next step and, finally, the $0.125 \times |e_2 - e_1|$ (obtained by shifting $|e_2 - e_1|$ to the right by three bits) is subtracted from that sum.

Note that the $\max(e_1, e_2)$ unit also generates signal sel, which is used to correctly choose between e_1 and e_2 for calculating $-|e_1 - e_2|$. It selects the smaller input from e_1 and e_2 and sends it to the subtractor. For example, if $e = e_2$, then e_1 is passed to the subtractor.

To calculate $AB + C$, the hardware implementation in Fig. 6.3 needs to be slightly modified. Since $e_2 = c + d = c$ in the $AB + C$ calculation, the adder that sums up c and d can be completely removed. The rest of the design will remain the same.

6.1.2 Evaluation of the LMAC

In order to analyze the accuracy of the proposed MAC unit, the MRED measure is considered [12]. We randomly generated 10 million quartets of values for A, B, C, and D and calculated the sum of products $AB + CD$ by using exact, LMAC, and other approximate MAC units.

For the hardware metrics, on the other hand, we implemented the exact and approximate MAC units in VHDL and synthesized them using the Synopsys

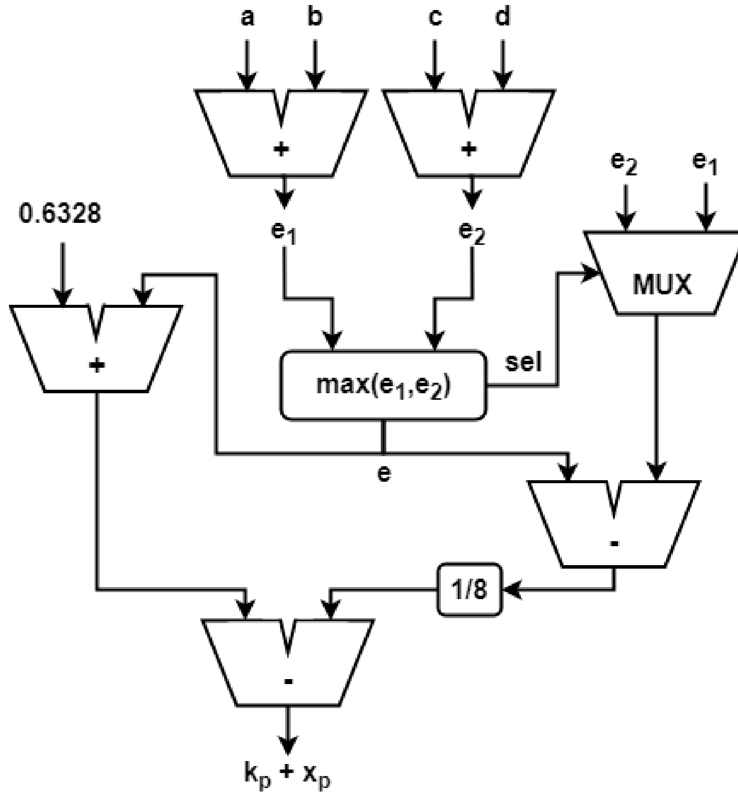


Figure 6.3: Block diagram of the LMAC unit.

Design Compiler (DC) for the STMicro CMOS 28-nm process to obtain the power dissipation, the circuit area, the critical path delay, and power-delay product (PDP). Table 6.1 shows the accuracy metric MRED and the four hardware measures.

The *CFPU* uses the mantissa from one of the inputs for the output mantissa rather than multiplying the two mantissas, known as the mantissa discarding technique [101]. We tried discarding the mantissa of the first and second inputs and we obtained similar MREDs. The other design in Table 6.1 is the truncated MAC, *Trun*, in which seven of the ten mantissa bits are set to zero [13].

As shown in Table 6.1, LMAC is the second most accurate MAC unit in terms of MRED, while being the most hardware-efficient design four hardware cost metrics. LMAC is 26.42% faster than the second fastest design CFPU and 2.5× faster than the conventional exact design. In terms of area, LMAC is the smallest design and it also consumes the least amount of power compared

Table 6.1: Accuracy and hardware measures of the exact, logarithmic, and other approximate MAC units.

MAC type	MRED	Area (μm^2)	Power (μW)	Delay (nS)	PDP (fJ)
Exact	-	895.8	817.5	3.65	2983.8
LMAC	0.106	118.6	122.3	1.42	173.66
CFPU [101]	0.312	145.6	130.4	1.93	251.67
Trun [13]	0.082	259.2	193.2	2.48	479.13

to the exact and other approximate designs. The results in Table 6.1 show a 17.18× reduction in energy consumption compared to the conventional exact MAC unit: the PDP is reduced from 2983.8 to 173.66.

As the results in Table 6.1 show, LMAC and the CFPU MAC units have notably smaller hardware footprints and can operate faster compared to the exact and the truncated MAC designs. Hence, we can conclude that multiplication has a significant impact on the hardware cost, and this is why doing it in the LNS (which is a simple addition) is more efficient.

6.1.3 Example application: image sharpening

To evaluate the effectiveness of the LMAC unit, we consider the image sharpening application. Image sharpening algorithms based on spatial filters are widely used in image processing to enhance the sharpness of an image without producing halo artifacts [102].

One image sharpening algorithm that uses approximate arithmetic is proposed in [11]. The same algorithm is used in this work. We also used the PSNR metric to measure the quality of the sharpened image using the exact MAC and those using an approximate MAC unit. The four sharpened images and their corresponding PSNR values are shown in Fig. 6.4.

As shown in Fig. 6.4, LMAC generates the highest output quality (i.e., the greatest PSNR value), more accurately than the truncated MAC, which had a smaller MRED. We believe this is due to significant cancellation of the bipolar approximation errors produced by the LMAC.

(a) Reference for PSNR

(b) PSNR = 26.58

(c) PSNR = 13.52

(d) PSNR = 23.18

Figure 6.4: Sharpened images using (a) exact MAC, (b) LMAC, (c) CFPU, and (4) Trun.

6.2 Approximate Accelerators for CNN-based Image Classifiers that Rely on Pixel Spatial Dependence

Reducing the bit precision and using approximate multipliers are both useful approximation techniques [8], [26], [90]–[92], [103]. However, we follow a different strategy and propose a hardware-efficient approximate architecture that accelerates the convolution operation in the convolutional layers of CNNs by skipping some multiplications.

The three key observations that motivate the proposed soft-dropping low-power (SDLP) approximate accelerator are as follows:

- Using approximate multipliers has a relatively small negative impact on the classification accuracy [8], [26], [103]. Hence, the output of the convolutional layer does not have to be fully accurate. The inaccuracy can be caused by providing inaccurate inputs to the convolutional layer and/or by using inexact techniques for performing the convolution operation.

- Removing some neurons often does not significantly impact the classification accuracy of CNNs and, therefore, the computation cost can be reduced [65]. This dropout technique shows that prior knowledge of the network is not required and the dropped out neurons can be chosen randomly.

- The convolutional layer is often followed by a pooling layer (to obtain either an average or a maximum) [29], which means that the exact locations of the extracted features in the convolutional layers are not as important as their location relative to the other features [30]. Hence, the exact values of the convolutional operation are not as important as their relative values.

SDLP is a proposed approximate accelerator that skips performing the multiplication operations for some neurons in the convolutional layers (referred to as SNs, short for skipped neurons) and uses the information of their adjacent neurons instead. Clearly, as the number of SNs increases, the implementation cost decreases at the cost of more accuracy degradation. Unlike references [104], [105], we do not determine the neurons that make the least important contribution to the network's output quality since that would be computation-intensive and not efficient in terms of hardware implementation. Additionally, the critical neurons would be different depending on the network's structure and dataset. We simply use the neurons in every other column of the convolution matrix as candidate SNs. More details are provided in Section 6.2.1.

The existing approximate accelerators for CNNs use one or more of the following techniques:

- Static fixed-point arithmetic: Use a reduced-precision fixed-point rep-

resentation for the operands rather than a full-precision floating-point representation [91], [106].

- Dynamic fixed-point: Use a dynamic fixed-point representation, where different scaling factors are used to process different parts of the network [107].
- Extreme quantification with binary weights: Train and evaluate CNNs that use extremely compact data representations (i.e., binary weights) [108], [109].
- Stochastic computing (SC): Use a random sequence of bits to represent the numbers. Stochastic arithmetic are performed with small circuits [110], [111].
- Approximate computing (AC): Use approximate arithmetic blocks (mainly for more resource-hungry multipliers) that exploit the inherent error resiliency of CNNs to reduce the hardware cost [8], [26].
- Weight pruning: Remove a set of weights (based on their magnitude [112] or the energy consumption of a node [113]) to overcome the over-fitting issue [114], [115].
- Dropout: Randomly select a few neurons and drop them from the network's structure at each training iteration, which helps to reduce the over-fitting issue [65].

Among these techniques, neuron dropout and weight pruning are the only ones that reduce the number of required computations in CNNs, while the other techniques just simplify the original number of calculations to reduce the area and energy consumption.

One of the main disadvantages of the weight pruning techniques is that they require detailed knowledge of the network in order to identify the best set of weights to prune. The best set of weights to prune depends on both the dataset and the network's structure; thus the set cannot be easily implemented in hardware. With respect to the random dropout technique, selecting

random neurons to drop not only does not reduce the hardware cost, but it also increases the hardware complexity. In fact, we still need to keep the hardware for all of the neurons and, moreover, add additional hardware for neurons random selection. The dropout technique is meant to be used for the training process of CNNs using software and the technique cannot be easily implemented in hardware. The SDLP, on the other hand, reduces the number of computations; however, it does not need to run an intensive analysis to find the best set of neurons that can be safely skipped in the multiplication operations.

Another unique feature of the SDLP is that all of the above-mentioned techniques can be also applied to it. The SDLP still performs multiplication and addition operations and, therefore, can benefit from the reduced precision, stochastic arithmetic unit, approximate arithmetic blocks, weight pruning, and the other aforementioned techniques. However, this has not been considered here since those other techniques were not the focus of this project. Moreover, the idea of using the spatial dependence between the input image pixels can be extended to include other inputs for which there are no sharp changes in the adjacent input components, such as audio signals.

6.2.1 Proposed approximate accelerators

Spatial Dependence Analysis

The SDLP accelerator was originally intended to be used in CNNs that classify images. Images are made up of rectangular arrays of pixels of different colors. In the design of the SDLP, we benefit from the fact that an image has smoothly varying color intensities. In fact, although pixels have different colors (represented with numerical values in a digital system), there are usually few sharp changes in the color component values of the adjacent pixels.

To verify the above-mentioned claim, several images are analyzed in this section. Since the CIFAR-10 dataset (short for Canadian Institute For Advanced Research) [116] is used in our image classification task, the sample images are chosen from this dataset. We randomly selected one hundred im-

Table 6.2: Average PSNR of reconstructed images by using approximate pixel values.

Approximation scheme	**PSNR**			
	$n = 64$	$n = 128$	$n = 256$	$n = 512$
LNP	35.85	32.83	30.07	27.34
RNP	37.16	33.54	30.15	27.14
LRP	39.99	36.08	32.31	30.16

ages from each of the 10 categories in the CIFAR-10 dataset and performed a through analysis on them. The images in the CIFAR-10 dataset are 32×32 and, therefore, there are 1024 pixels in each of the red, green, and blue planes. The experiment on the spatial dependence of the pixels is done in three steps. First, n pixels are randomly selected from the 1024 pixels of each plane. Once the pixels are selected, the same pixels are used in all three planes. The second step is to approximate the value of these n pixels with respect to their adjacent pixels. Finally, the images constructed from approximate values are compared with the original ones and the approximation quality is evaluated by using the well-known peak signal-to-noise ratio (PSNR) metric.

Approximating the exact values is done using three different approximation schemes: (1) using the value of the left-neighbor pixel (LNP), (2) using the value of the right-neighbor pixel (RNP), and (3) using the average value of the left and right neighbor pixels (LRP). The average PSNR values for the randomly selected pixels over the one thousand CIFAR-10 dataset images are reported in Table 6.2 for different numbers of approximated pixels, n.

As the results in Table 6.2 show, LRP achieves the highest PSNR. However, the PSNR for the LNP and RNP are still relatively high values even for very large n values, indicating good image quality. Hence, it can be concluded that each pixel provides a reasonable estimate of the values of its neighbor pixels. Clearly, increasing the number of approximated pixels worsens the PSNR, as reflected in Table 6.2.

Figure 6.5: Inputs and outputs to a convolutional layer followed by an average pooling layer.

Approximate Matrix Multiplication

SDLP benefits from the correlation between the adjacent pixels in an image and skips the exact MAC operation for some neurons (a.k.a. the SNs). A convolutional layer followed by an average pooling layer is shown in Fig. 6.5.

In Fig. 6.5, X denotes the input to the convolution layer, Y is the output of the convolutional layer, which is used as the input to the pooling layer, and Z contains the output values of the pooling layer.

In the light of what we described above and what has been reported in the literature, we conclude:

- Successfully applying approximate multipliers to the convolutional layers in previous publications [8], [26], [103] shows that the elements of matrix Y do not have to be fully accurate.

- The major benefits of using the dropout technique [65] are that not all of the neurons in the convolution layer, i.e. matrix Y, are required for acceptable CNN operation and, therefore, some neurons can be safely removed.

- The adjacent pixels in matrix Y will be merged into only one value in matrix Z as a result of the pooling operation [30]. Consequently, it is not necessary to have exact values in Y and, furthermore, Z does not need to be fully-accurate.

SDLP exploits the fact that "having the exact values at the output of the convolutional layer is not necessary" and this realization allows the convolution operation to be simplified by skipping the exact MAC operation for some neurons.

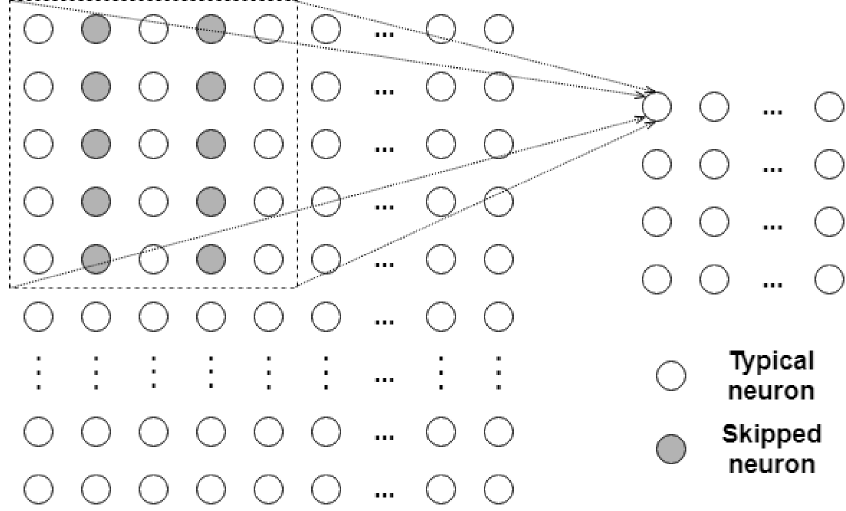


Figure 6.6: An example of the SDLP approximate accelerator for filter size of 5×5.

An example of the SDLP for a filter size of 5×5 is shown in Fig. 6.6. The colored neurons in this figure indicate the SNs. As shown in the figure, the neurons in every other column are chosen to be SNs. Note that we did not perform any optimization on the number of SNs or how they are selected. This could be studied in a future work and as an optimization problem.

As mentioned before, each SN uses only the information of their adjacent neurons in the SDLP. This is done using three scenarios, as explained below.

The left-neighbor approximation (SDLP-LNA) In this scenario, the value of the neuron on the left side of each SN is used as the SN's value.

Typically, the convolution operation with a $k \times k$ spatial filter requires k^2 multiplication and $k^2 - 1$ additions. The additions can be performed using an s-stage adder-tree structure, where $s = \lceil log_2(k^2) \rceil$. Assuming that the additions at each stage are done in parallel in a time τ_a, the entire addition process takes time $s\tau_a$. Similarly, all of the multiplications can be performed in parallel at once in a time τ_m and, therefore, the entire convolution operation can be completed in time:

$$\tau_{Conv} = \tau_m + \left\lceil log_2(k^2) \right\rceil \tau_a. \tag{6.13}$$

SDLP-LNA, on the other hand, reduces the number of multiplications to

$k(k - \lfloor k/2 \rfloor)$. Note that SDLP-LNA does not reduce the number of required additions, however, it speeds up the addition process. By using the same value for each SN as its neighbor neuron, the addition can be converted into faster and less energy-consuming single left-shift operation, reducing the addition time τ_a to τ_s. By doing so, the entire convolution operation in SDLP-LNA can be completed in time:

$$\tau_{SDLP-LNA} = \tau_m + \tau_s + \left\lceil log_2\big(k^2 - k\lfloor \frac{k}{2} \rfloor\big) \right\rceil \tau_a. \tag{6.14}$$

Note that in (6.14), although the number of required multiplication operations is reduced, the execution time τ_m remains the same. This is due the parallel execution of the multiple multiplication operations. Then adding the $k\lfloor k/2 \rfloor SN$s with their adjacent neurons can be performed using single left shifts (because they have similar values and, consequently, a left shift models multiplication with two) that takes time τ_s. Finally, the remaining $k^2 - k\lfloor k/2 \rfloor$ terms can be summed up with an adder tree, which takes time $\lceil \log_2\big(k^2 - k\lfloor k/2 \rfloor\big) \rceil \tau_a$.

Given that $\tau_s << \tau_a$, SDLP-LNA can result in significant improvement in the latency of the convolutional layers of a CNN, especially for larger bit precision. Note that increasing the bit precision makes the addition process even slower than the shift operation. According to our simulations for an 8-bit design, the shift operation is 6.8× faster than the addition operation. For a 16-bit design, on the other hand, the shift operation is 14.1× faster than the addition operation. Note that these results are obtained by implementing the circuits in VHDL and synthesizing them by using the Synopsys Design Compiler (DC) for ST Micro's 28-nm CMOS process.

The right-neighbor approximation (SDLP-RNA) This scenario is very similar to the SDLP-LNA and the only difference is that it uses the value of the neuron on the right side of each SN as the SN's value. Consequently, the reduction in the number of multipliers and the achieved speedup is similar to that for SDLP-LNA.

The average of the left and right neighbors approximation (SDLP-

LRA) In this scenario, the average value of the left and right neighbor neurons to each SN is used as the SN's value. Hence, it is more complex than the two previously discussed scenarios. However, it is still more hardware-efficient than the conventional architecture as it reduces the number of multiplication to $k(k - \lfloor k/2 \rfloor)$, similar to SDLP-LNA.

Regarding the number of required additions, SDLP-LRA needs to take the average value of some neurons and, therefore, it may seem that it increases the depth of the adder-tree. However, we successfully managed this issue by reforming the required additions and proposing a novel compact adder. This technique is explained below using an illustrative example.

There are nine terms to be added after the multiplication in a 3×3 filter; $sum = a + b + c + d + e + f + g + h + i$. Following the given architecture in Fig. 6.6, let b, e, and h be the three SNs in this example. Normally b, e, and h should be calculated first as $b = 0.5(a + c)$, $e = 0.5(d + f)$, and $h = 0.5(g + i)$. Assuming that the additions are preformed all at once in parallel, calculating b, e, and h can be completed in time $\tau_1 = \tau_a + \tau_s$. Then τ_1 needs to be added to $\tau_2 = \big(\lceil \log_2(9) \rceil\big)\tau_a$ to give the total addition time for the nine values. This clearly increases the number of required adders and the latency. We propose simplifying this addition as $sum = 1.5sum_{temp}$, where $sum_{temp} = (a + c + d + f + g + i)$. By doing so, six terms are added to generate sum_{temp}, then sum_{temp} is the shifted to the right by one bit and, finally, it is added to its initial value before shift to produce the result for the final summation. The required time for completing the entire convolution operation by using the SDLP-RNA scenario can be calculated as:

$$\tau_{SDLP-LRA} = \tau_m + \left\lceil log_2\big(k^2 - k\lfloor \frac{k}{2} \rfloor\big) \right\rceil \tau_a + \tau_s + \tau_a. \tag{6.15}$$

Similarly, $\tau_s << \tau_a << \tau_m$, however, τ_a is multiplied by a coefficient, which makes it significant in size compared to τ_m, especially for larger k values.

To further improve both the hardware and timing efficiency of the SDLP-LRA accelerator, a specialized compact adder is also proposed. As mentioned above, the SDLP-LRA calculates $sum = 1.5sum_{temp} = sum_{temp} + 0.5sum_{temp}$.

This is a special case of addition due to the correlation between its inputs, i.e. one input is half of the other one. Let $\alpha =$ "$a_3a_2a_1a_0$" be the 4-bit binary representation of sum_{temp}. Hence, the second input to this adder would be $\beta =$ "$a_3a_3a_2a_1$". Note that sign extension is considered as we might be dealing with negative values. Such an adder is designed in this Chapter and its detailed hardware implementation is provided in Table 6.3. Ideally, we would add the expressions for all of the *sum* and *carry* bits of the conventional adder as well; but it consists of a large number of terms (i.e., sum-of-products). Hence, it is not reported for readability and simplicity. However, only one of the output bits is calculated and shown in Table 6.3 for comparison purposes.

General 4-bit inputs $\alpha =$ "$a_3a_2a_1a_0$" and $\beta =$ "$b_3b_2b_1b_0$" are used in Table 6.3 to better distinguish the compact and the conventional adders. However, the compact adder is simplified considering that $b_0 = a_1$, $b_1 = a_2$, $b_2 = a_3$, and $b_3 = b_2$ (as the sign bit needs to be extended). Although 8-bit adders are used in the hardware implementation, a 4-bit adder is assumed in Table 6.3. This is only for the sake of simplicity and more clarity in this Chapter. One can readily extend this adder to larger designs by using the following Boolean rules:

$$\begin{aligned} s_i &= a_i \oplus b_i \oplus c_{i-1} \\ c_i &= a_i b_i + a_i c_{i-1} + b_i c_{i-1}, \end{aligned} \tag{6.16}$$

where c_i and s_i denote the carry and sum signals at position i, where $i \in \{1, 2, ..., 7\}$ and $c_0 = 0$. The compact adder would be faster than the conventional one and, therefore, (6.15) can be rewritten as:

$$\tau_{SDLP-LRA} = \tau_m + \left\lceil log_2\big(k^2 - k\lfloor\frac{k}{2}\rfloor\big)\right\rceil \tau_a + \tau_s + \tau'_a. \tag{6.17}$$

where $\tau'_a < \tau_a$ denotes the critical path delay of the proposed compact adder. The relation between τ'_a and τ_a depends on the length of the adder and, therefore, it is not measured separately. However, the hardware metrics for the entire design (i.e., adders included) are reported in Section 6.2.2.

Table 6.3: The AND-OR-Invert (AOI) logic implementation of the proposed compact adder.

***Sum* bits of the proposed compact adder**	***Carry* bits of the proposed compact adder**
$s_0 = a_0\bar{a}_1 + a_1\bar{a}_0$	$c_0 = a_0a_1$
$s_1 = (a_2\bar{c}_1) + (a_2a_0) + (\bar{a}_2a_1\bar{a}_0)$	$c_1 = a_1a_2 + a_1a_0$
$s_2 = (a_3a_2a_1) + (a_3\bar{a}_2\bar{a}_1) + (a_3a_1\bar{a}_0) + (\bar{a}_3a_2\bar{a}_1) + (\bar{a}_3\bar{a}_2a_1a_0)$	$c_2 = (a_3a_2) + (a_2a_1) + (a_3a_1a_0)$
$s_3 = (a_3a_2) + (a_2a_1) + (a_3a_1a_0)$	$c_3 = a_3$
s_1 and c_1 bit of the conventional adder	
$s_1 = (a_1a_0b_1b_0) + (a_1\bar{a}_0\bar{b}_1) + (a_1\bar{b}_1\bar{b}_0) + (\bar{a}_1a_0\bar{b}_1b_0) + (\bar{a}_1\bar{a}_0b_1) + (\bar{a}_1b_1\bar{b}_0)$ $c_1 = (a_1b_1) + (a_1a_0b_0) + (b_1b_0a_0)$	

6.2.2 Evaluation of the SDLP Accelerators

To evaluate the SDLP approximate accelerator, we implement a CNN in MATLAB. The network architecture is shown in Fig. 6.7. The network is trained from scratch using the conventional exact architecture. Then the inference CNN design is implemented, where the convolutional layers are replaced with our own SDLP-based convolutional layers.

As shown in Fig. 6.7, the employed CNN has three convolutional layers, each followed by a ReLU activation (which is not shown in the figure) and an average pooling layer. The final layers consist of two fully-connected layers, a ReLU activation layer, and a soft-max layer. The *softmax* layer assigns probabilities to each class in a multi-class classification CNN. In fact, in a classification problem with C classes, p_i $\big(i \in \{1, 2, ..., C\}$ and $\sum_{i=1}^{C} p_i = 1\big)$ anticipates the likelihood of the input image belonging to class i. The specification of each layer, such as the number of filters and filter sizes, are also indicated in Fig. 6.7. Only the convolutional layers are modified by using the SDLP accelerator; the other layers are kept exact.

We divide the performance analysis of the approximate SDLP accelerator in this section into two parts: (1) accuracy metrics and (2) hardware metrics.

Accuracy metrics

An important feature of any approximate design is its impact on the accuracy. Hence, the CNN in Fig. 6.7 is used to classify the CIFAR-10 dataset.

The only two approximate architectures for NNs that reduce the number of required operations are the dropout and weight pruning techniques, neither

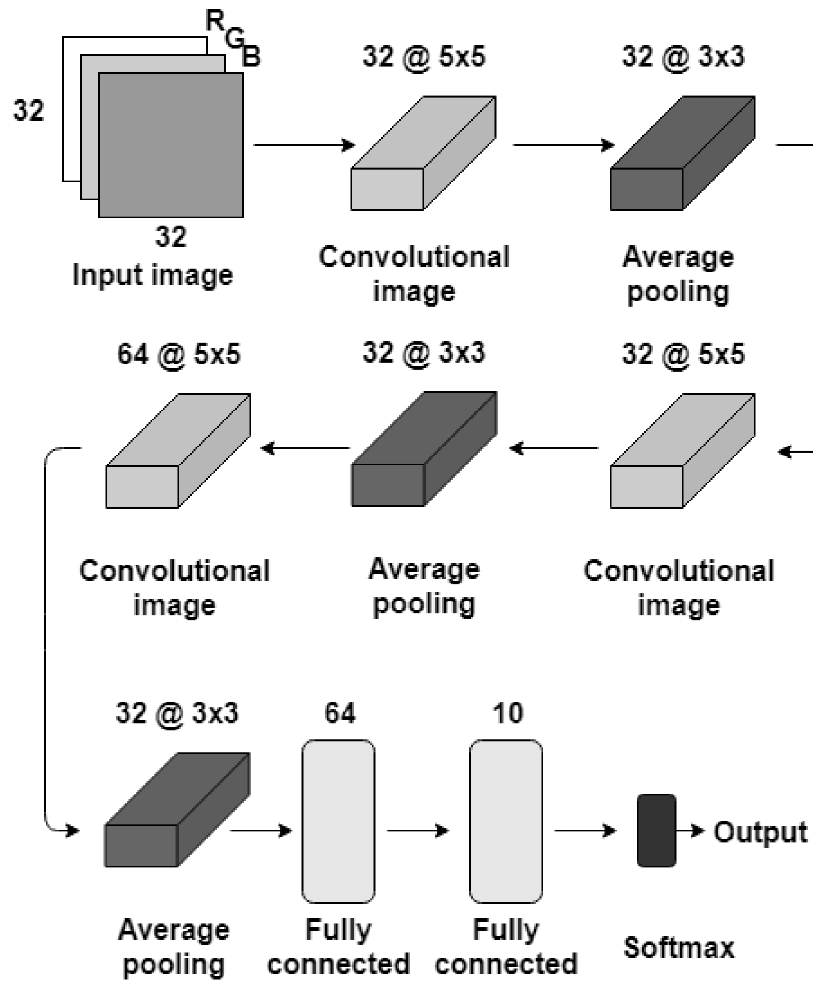


Figure 6.7: Architecture of the employed CNN.

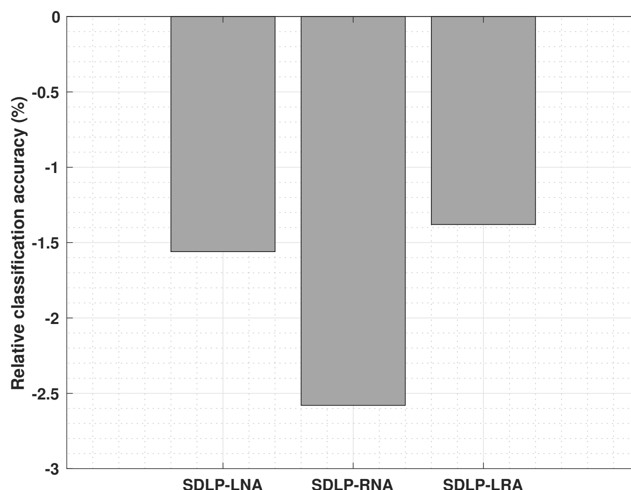


Figure 6.8: Effects of the SDLP accelerator on the classification accuracy on CIFAR-10 dataset.

of which are applicable to a hardware-level implementation. Moreover, as mentioned earlier, all of the existing techniques in the literature are applicable to the SDLP as well. For these reasons, we only compare the SDLP with the conventional exact architecture.

The impact of SDLP with the three different approximation scenarios on the classification accuracy of the CIFAR-10 dataset is illustrated in Fig. 6.8.

Fig. 6.8 shows how much each of the three variants of the SDLP is less accurate than the conventional exact architecture. As shown in the figure, SDLP-LRA achieves the highest accuracy. However, the other two variants (i.e., SDLP-LNA and SDLP-RNA) are only slightly less accurate than the SDLP-LRA. According to Fig. 6.8, the maximum accuracy loss in the SDLP accelerator is only 2.58%.

Hardware metrics

The convolution operation between two 3×3 matrices was implemented in VHDL and then synthesized using the Synopsys Design Compiler (DC) for ST Micro's 28-nm CMOS process.

The four main hardware metrics for any digital system are: area, latency, power consumption, and PDP. These metrics are reported in Table 6.4 for

Table 6.4: Comparison of the hardware cost between the exact and the approximate variants of the SDLP accelerators.

Approximation scheme	**Area** (μm^2)	**Latency** (nS)	**Power** (mW)	**Energy** (fJ)
Exact	1952.68	1.56	1.46	2277.6
SDLP-LNA	1271.98	1.41	0.95	1339.5
SDLP-LRA	1325.83	1.49	1.01	1504.9

different variants of the SDLP accelerator. Note that since SDLP-LNA and SDLP-RNA have the same architecture, we only report the hardware results for SDLP-LNA.

As shown in Table 6.4 and, as expected, SDLP-LNA is more hardware-efficient than the SDLP-LRA. According to Table 6.4, SDLP-LNA and SDLP-LRA accelerate the convolution operation by 9.6% and 4.48%, respectively, compared to the conventional exact design. They are also 1.7× and 1.5× more energy-efficient, respectively, according to the PDP metric.

Although SDLP-LNA is much more hardware-efficient than SDLP-LRA, it is only 0.18% less accurate than it.

6.3 Summary

This chapter proposed the first logarithmic approximate MAC unit, which we call LMAC. As opposed to other approximate MAC units that take advantage of approximate multipliers and adders, LMAC uses the least squares method to linearly approximate the nonlinear functions in the logarithmic domain. According to our simulation results, LMAC achieves the second-best MRED, the lowest energy consumption, and the highest throughput for $AB + CD$ calculation. Moreover, evaluation of LMAC and other approximate MAC units in an image sharpening application shows that LMAC generates the best objective output quality, in terms of the PSNR.

We also proposed three variants of the SDLP accelerators for CNN-based image classifiers. The SDLP approximates the MAC operation by skipping the

exact multiplication operation for some neurons (a.k.a. SNs) while performing the convolution operation and, instead, uses the values of their adjacent neurons. The SNs in this chapter are simply chosen as illustrated in Fig. 6.6, i.e., the neurons in every other column. Three scenarios are used for predicting the values of SNs: (1) using the values of the neurons on the left-hand side (SDLP-LNA), (2) using the values of the neurons on the right-hand side (SDLP-RNA), and (3) using the average value of the right and left neighbor neurons (SDLP-LRA). Our simulation results show that the maximum accuracy degradation for CIFAR-10 dataset is only 2.58%, obtained by using SDLP-RNA. The minimum accuracy loss, on the other hand, is 1.38% that is obtained by SDLP-LRA. The SDLP-LNA seems to be the best variant of the SDLP as it is only 0.18% less accurate than the most accurate variant SDLP-LRA, while being 10.99% and 41.18% more energy-efficient than the SDLP-LRA and the conventional exact architecture, respectively.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This dissertation begins with an introduction to computation-intensive applications including DSP, NNs, and MIMO receivers. Power- and resource-hungry arithmetic operation in these applications are studied and the existing methodologies for designing approximate arithmetic circuits are reviewed in detail in Chapter 2.

In Chapter 3, an initial approximate 4:2 compressor that introduces a rather large error to the output is proposed. However, the number of faulty rows in the compressor's truth table is significantly reduced by encoding its inputs using generate and propagate signals. Based on this improved compressor, two 4×4 multipliers are designed with different accuracies and then are used as building blocks for scaling up to larger multipliers. It is shown, for the first time, that the approximate multipliers can be safely used in the interference nulling calculation of the MIMO baseband receiver, where the channel codes can compensate for the approximation errors as well as the channel noise.

A challenging topic that has never been addressed previously in the literature is finding the features of an approximate multiplier that make it a superior design in NNs. This problem is investigated in Chapter 4, where 600 approximate multipliers are considered and the critical features in the superior approximate multipliers are identified by using a statistical feature selection approach. We found out that the most important features that make

an approximate multiplier superior to others are the variance of the error distance (Var-ED) and the root mean square of the error distance (RMS-ED); better designs tend to have smaller Var-ED and RMSE-ED values. Although the statistically most relevant and critical features of approximate multipliers are identified in this work, ensuring the best choice of approximate multiplier requires application-dependent experimentation.

Multiplication and its special case, the squaring operation, in the LNS are discussed in Chapter 5. An important step in every logarithmic arithmetic circuit is finding the base-2 logarithm of the input operand(s). This is done in the literature by using LODs. Chapter 5 proposes an exact 16-bit LOD that is used to find the position of the leading one in a 32-bit number. Compared with the original Mitchell approximate multiplier, the proposed LOD reduces the PDP by 24.89% and makes the Mitchell multiplier 1.4× faster. A NOD is also proposed for the first time that causes the LMs have a double-sided error distribution. The proposed ILM using the NOD obtains a higher classification accuracy for the CIFAR-10 dataset than the other LMs. We attribute this higher accuracy to the double-sided noise that is introduced into the NN evaluation by the ILM.

Finally, MAC units and accelerators are studied in Chapter 6. The first fully-logarithmic MAC (LMAC) unit is proposed. As opposed to other approximate MAC units that take advantage of approximate multipliers and adders, LMAC uses the least squares method to linearly approximate the nonlinear functions in the logarithmic domain. LMAC is evaluated in an image sharpening application where it is shown to generate the best output quality, in terms of the objective PSNR metric. Chapter 6 also proposed the SDLP architecture that is used to accelerate the convolution operations in CNNs. It benefits from the spatial dependence between the input image pixels and skips the exact MAC for some neurons. Our simulation results show that the SDLP slightly reduces the CNN classification accuracy by 0.18%, while being 41.18% more energy-efficient than a CNN constructed using a conventional exact MAC.

7.2 Future work

There are other areas where approximate computing techniques could be beneficial. In general, error-resilient applications and those that are inherently noisy, such as having noisy inputs, could be good candidates for approximate computing. In such applications, the errors introduced by approximation techniques are unavoidably combined with the noise that already exists in the system and the system might be able to manage these two sources of inaccuracy. An example of this is when the data is protected, end-to-end, through the use of error-correcting codes.

A very interesting application could be the hardware-efficient design of LDPC code decoders. The demand for higher data rates and more reliable communication standards, such as in IEEE 802.3an, 802.11n, 802.15, 802.16, ETSI 2nd Gen. DVB, 3GPP LTE (4G) and ITU-T G.9960 and G.709 [30], [72], [117] is pushing next-generation standards toward error correction schemes allowing high throughput decoding with near Shannon limit performance [117]. Currently, LDPC codes are known to be the best candidates to meet these competing requirements [117]–[119]. However, LDPC decoders are very complex even for a short length codeword [117]. Hence, one could take advantage of approximate computing as a potential solution to overcome this concern.

Furthermore, the LMAC unit and the SDLP architecture proposed in this research project can be applied to other applications as well. Considering the increasing complexity of ML applications, including NNs, the LMAC can be used to significantly reduce the hardware implementation cost of these algorithms. The SDLP, on the other hand, is specifically designed for CNNs that classify images. The main idea of the SDLP, i.e. exploiting the spatial dependence between the inputs, can be extended to other types of CNNs and, further, to other applications (e.g., image and signal processing).

References

[1] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *18th IEEE European Test Symposium (ETS)*, IEEE, 2013, pp. 1–6.

[2] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

[3] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," *Proceedings of the 50th Annual Design Automation Conference*, p. 113, 2013.

[4] M. S. Ansari, A. Mahani, J. Han, and B. F. Cockburn, "A novel gate grading approach for soft error tolerance in combinational circuits," *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–4, 2016.

[5] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, no. 5, pp. 1694–1702, 2017.

[6] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[7] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 145–150, 2016.

[8] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, "Design of power-efficient approximate multipliers for approximate artificial neural networks," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, 2016.

[9] S. Draghici, "On the capabilities of neural networks using limited precision weights," *Neural networks*, vol. 15, no. 3, pp. 395–414, 2002.

[10] J. L. Holi and J.-N. Hwang, "Finite precision error analysis of neural network hardware implementations," *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 281–290, 1993.

[11] M. S. Lau, K.-V. Ling, and Y.-C. Chu, "Energy-aware probabilistic multiplier: Design and analysis," *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 281–290, 2009.

[12] M. S. Ansari, H. Jiang, B. F. Cockburn, and J. Han, "Low-power approximate multipliers using encoded partial products and approximate compressors," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 3, pp. 404–416, 2018.

[13] H. Jiang, C. Liu, L. Liu, F. Lombardi, and J. Han, "A review, classification, and comparative evaluation of approximate arithmetic circuits," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 4, p. 60, 2017.

[14] J. Yang, G. Zhu, and Y.-Q. Shi, "Analyzing the effect of JPEG compression on local variance of image intensity," *IEEE Transactions on Image Processing*, vol. 25, no. 6, pp. 2647–2656, 2016.

[15] M. Shah, "Future of JPEG XT: Privacy and security," PhD thesis, 2016.

[16] H. Taub and D. L. Schilling, *Principles of Communication Systems*. McGraw-Hill Higher Education, New York, NY, 1986.

[17] J.-M. Chung, J. Kim, and D. Han, "Multihop hybrid virtual mimo scheme for wireless sensor networks," *IEEE Transactions on vehicular Technology*, vol. 61, no. 9, pp. 4069–4078, 2012.

[18] H. Zhong, W. Xu, N. Xie, and T. Zhang, "Area-efficient min-sum decoder design for high-rate quasi-cyclic low-density parity-check codes in magnetic recording," *IEEE Transactions on Magnetics*, vol. 43, no. 12, pp. 4117–4122, 2007.

[19] C. Teuscher, *Turing's connectionism: an investigation of neural network architectures*. Springer Science & Business Media, 2012.

[20] B. C. Csáji, "Approximation with artificial neural networks," *Faculty of Sciences, Etvs Lornd University, Hungary*, vol. 24, p. 48, 2001.

[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[22] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural computing and applications*, pp. 1–31, 2018.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

[24] L. C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected CRFS," *arXiv preprint arXiv:1412.7062*, 2014.

[25] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[26] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, "Energy-efficient neural computing with approximate multipliers," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 2, p. 16, 2018.

[27] A. Van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," in *Advances in neural information processing systems*, 2013, pp. 2643–2651.

[28] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 160–167.

[29] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 111–118.

[30] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[31] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," *International Conference on Artificial Neural Networks and Machine Learning*, pp. 281–290, 2014.

[32] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGA: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[33] S. L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, 2004.

[34] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," *Design, automation and test in Europe*, pp. 1250–1255, 2008.

[35] X. Yang, Y. Xing, F. Qiao, Q. Wei, and H. Yang, "Approximate adder with hybrid prediction and error compensation technique," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 373–378, 2016.

[36] N. Zhu, W. L. Goh, G. Wang, and K. S. Yeo, "Enhanced low-power high-speed adder for error-tolerant application," *International SoC Design Conference*, pp. 323–327, 2010.

[37] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi, “Approximate xor/xnor-based adders for inexact computing,” *13th IEEE International Conference on Nanotechnology*, pp. 690–693, 2013.

[38] 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*, vol. 57, no. 4, pp. 850–862, 2010.

[39] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, “Design of voltage-scalable meta-functions for approximate computing,” *Design, Automation & Test in Europe*, pp. 1–6, 2011.

[40] P. Kulkarni, P. Gupta, and M. Ercegovac, “Trading accuracy for power with an underdesigned multiplier architecture,” in *IEEE International Conference on VLSI Design*, IEEE, 2011, pp. 346–351.

[41] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, “Low-power high-speed multiplier for error-tolerant application,” in *Electron Devices and Solid-State Circuits (EDSSC), 2010 IEEE International Conference of*, IEEE, 2010, pp. 1–4.

[42] C.-H. Lin and C. Lin, “High accuracy approximate multiplier with error correction,” in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, IEEE, 2013, pp. 33–38.

[43] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, “Design and analysis of approximate compressors for multiplication,” *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984–994, 2015.

[44] C. Liu, J. Han, and F. Lombardi, “A low-power, high-performance approximate multiplier with configurable partial error recovery,” *Design, Automation and Test in Europe Conference and Exhibition*, pp. 1–4, 2014.

[45] H. Jiang, J. Han, F. Qiao, and F. Lombardi, “Approximate radix-8 Booth multipliers for low-power and high-performance operation,” *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2638–2644, 2016.

[46] Z. Vasicek and L. Sekanina, “Evolutionary approach to approximate digital circuits design,” *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 432–444, 2015.

[47] J. N. Mitchell, “Computer multiplication and division using binary logarithms,” *IRE Transactions on Electronic Computers*, no. 4, pp. 512–517, 1962.

[48] H. Saadat, H. Bokhari, and S. Parameswaran, “Minimally biased multipliers for approximate integer and floating-point multiplication,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2623–2635, 2018.

[49] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, 2018.

[50] M. S. Kim, A. A. Del Barrio, R. Hermida, and N. Bagherzadeh, "Low-power implementation of Mitchell's approximate logarithmic multiplication for convolutional neural networks," *23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 617–622, 2018.

[51] M. S. Kim, A. A. D. B. Garcia, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient Mitchell's approximate log multipliers for convolutional neural networks," *IEEE Transactions on Computers*, 2018.

[52] D. De Caro, N. Petra, and A. G. Strollo, "Efficient logarithmic converters for digital signal processing applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 58, no. 10, pp. 667–671, 2011.

[53] J. Y. L. Low and C. C. Jong, "Unified Mitchell-based approximation for efficient logarithmic conversion circuit," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1783–1797, 2015.

[54] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, "Design and analysis of approximate compressors for multiplication," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984–994, 2014.

[55] L. Qian, C. Wang, W. Liu, F. Lombardi, and J. Han, "Design and evaluation of an approximate Wallace-Booth multiplier," *IEEE international symposium on circuits and systems (ISCAS)*, pp. 1974–1977, 2016.

[56] S. Venkatachalam and S.-B. Ko, "Design of power and area efficient approximate multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1782–1786, 2017.

[57] K. Bhardwaj, P. S. Mane, and J. Henkel, "Power-and area-efficient approximate Wallace tree multiplier for error-resilient systems," in *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, IEEE, 2014, pp. 263–269.

[58] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini, "Fast neural networks without multipliers," *IEEE Transactions on Neural Networks*, vol. 4, no. 1, pp. 53–62, 1993.

[59] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[60] E. H. Lee and S. S. Wong, "Analysis and design of a passive switched-capacitor matrix multiplier for approximate computing," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 261–271, 2016.

[61] S. Gopal, P. Agarwal, J. Baylon, L. Renaud, S. N. Ali, P. P. Pande, and D. Heo, "A spatial multi-bit sub-1 V time-domain matrix multiplier interface for approximate computing in 65-nm CMOS," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 3, pp. 506–518, 2018.

[62] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*,

[63] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, vol. 2011, no. 2, p. 5, 2011.

[64] *Evoapprox8b - approximate adders and multipliers library*, `http://www.fit.vutbr.cz/research/groups/ehw/approxlib/`, Accessed: 2019-08-31.

[65] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[66] C. S. Leung, H.-J. Wang, and J. Sum, "On the selection of weight decay parameter for faulty networks," *IEEE Transactions on Neural Networks*, vol. 21, no. 8, pp. 1232–1244, 2010.

[67] Y. Shao, G. N. Taff, and S. J. Walsh, "Comparison of early stopping criteria for neural-network-based subpixel classification," *IEEE Geoscience and Remote Sensing Letters*, vol. 8, no. 1, pp. 113–117, 2011.

[68] Y. Luo and F. Yang, "Deep learning with noise," *hp://www. andrew. cmu. edu/user/fanyang1/deep-learning-with-noise. pdf*, 2014.

[69] N. Nagabushan, N. Satish, and S. Raghuram, "Effect of injected noise in deep neural networks," *International Conference on Computational Intelligence and Computing Research*, pp. 1–5, 2016.

[70] T. He, Y. Zhang, J. Droppo, and K. Yu, "On training bi-directional neural network language model with noise contrastive estimation," *10th International Symposium on Chinese Spoken Language Processing*, pp. 1–5, 2016.

[71] A. F. Murray and P. J. Edwards, "Enhanced MLP performance and fault tolerance resulting from synaptic weight noise during training," *IEEE Transactions on Neural Networks*, vol. 5, no. 5, pp. 792–802, 1994.

[72] J. Sum, C.-S. Leung, and K. Ho, "Convergence analyses on on-line weight noise injection-based training algorithms for mlps," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 11, pp. 1827–1840, 2012.

[73] K. Ho, C.-S. Leung, and J. Sum, “Objective functions of online weight noise injection training algorithms for MLPs,” *IEEE Transactions on Neural Networks*, vol. 22, no. 2, pp. 317–323, 2011.

[74] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror, “Result analysis of the NIPS 2003 feature selection challenge,” *Advances in Neural Information Processing Systems*, pp. 545–552, 2005.

[75] I. Guyon and A. Elisseeff, “An introduction to variable and feature selection,” *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[76] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[77] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, “Gene selection for cancer classification using support vector machines,” *Machine Learning*, vol. 46, no. 1-3, pp. 389–422, 2002.

[78] A. Kraskov, H. Stögbauer, and P. Grassberger, “Estimating mutual information,” *Physical Review E*, vol. 69, no. 6, p. 066 138, 2004.

[79] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Machine Learning*, vol. 63, no. 1, pp. 3–42, 2006.

[80] *MATLAB classification learner app. toolbox*, `https://www.mathworks.com/help/stats/classificationlearner-app.html`, Accessed: 2019-08-31.

[81] *Imagenet large scale visual recognition challenge (ilsvrc)*, `http://www.image-net.org/challenges/LSVRC/,2015`.

[82] K. H. Abed and R. E. Siferd, “CMOS VLSI implementation of a low-power logarithmic converter,” *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1421–1433, 2003.

[83] G. Srinivasan, P. Wijesinghe, S. S. Sarwar, A. Jaiswal, and K. Roy, “Significance driven hybrid 8T-6T SRAM for energy-efficient synaptic storage in artificial neural networks,” *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 151–156, 2016.

[84] K. Kunaraj and R. Seshasayanan, “Leading one detectors and leading one position detectors-an evolutionary design methodology,” *Canadian Journal of Electrical and Computer Engineering*, vol. 36, no. 3, pp. 103–110, 2013.

[85] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, “Weight uncertainty in neural networks,” *arXiv preprint arXiv:1505.05424*, 2015.

[86] Y. Xie, S. Liao, B. Yuan, Y. Wang, and Z. Wang, "Fully-parallel area-efficient deep neural network design using stochastic computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 12, pp. 1382–1386, 2017.

[87] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Transactions on computers*, vol. 62, no. 9, pp. 1760–1771, 2013.

[88] M. S. Ansari, B. Cockburn, and J. Han, "A hardware-efficient logarithmic multiplier with improved accuracy," *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 922–925, 2019.

[89] Z. Babić, A. Avramović, and P. Bulić, "An iterative logarithmic multiplier," *Microprocessors and Microsystems*, vol. 35, no. 1, pp. 23–33, 2011.

[90] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[91] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *International Conference on Machine Learning*, pp. 1737–1746, 2015.

[92] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[93] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[94] C. Wang and J. C. Principe, "Training neural networks with additive noise in the desired signal," *IEEE Transactions on Neural Networks*, vol. 10, no. 6, pp. 1511–1517, 1999.

[95] *MATLAB curve fitting toolbox*, `https://www.mathworks.com/help/curvefit/curve-fitting.html`, Accessed: 2019-08-31.

[96] A. Avramović, Z. Babić, D. Raič, D. Strle, and P. Bulić, "An approximate logarithmic squaring circuit with error compensation for DSP applications," *Microelectronics Journal*, vol. 45, no. 3, pp. 263–271, 2014.

[97] J. P. Langlois and D. Al-Khalili, "Carry-free approximate squaring functions with $\mathcal{O}(n)$ complexity and $\mathcal{O}(1)$ delay," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, no. 5, pp. 374–378, 2006.

[98] M.-H. Sheu and S.-H. Lin, "Fast compensative design approach for the approximate squaring function," *IEEE Journal of Solid-state Circuits*, vol. 37, no. 1, pp. 95–97, 2002.

[99] H. Anton, *Elementary Linear Algebra*. John Wiley & Sons, Ltd, Hoboken, NJ, 2019.

[100] T. Yang, T. Sato, and T. Ukezono, "A low-power approximate multiply-add unit," *2nd International Symposium on Devices, Circuits and Systems (ISDCS)*, pp. 1–4, 2019.

[101] D. Peroni, M. Imani, and T. Rosing, "Runtime efficiency-accuracy trade-off using configurable floating point multiplier," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[102] C. C. Pham and J. W. Jeon, "Efficient image sharpening and denoising using adaptive guided image filtering," *IET Image Processing*, vol. 9, no. 1, pp. 71–79, 2014.

[103] U. Lotrič and P. Bulić, "Applicability of approximate multipliers in hardware neural networks," *Neurocomputing*, vol. 96, pp. 57–65, 2012.

[104] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient neuromorphic systems using approximate computing," *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, pp. 27–32, 2014.

[105] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: An approximate computing framework for artificial neural network," *Design, Automation & Test in Europe Conference & Exhibition*, pp. 701–706, 2015.

[106] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[107] P. Gysel, "Ristretto: Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1605.06402*, 2016.

[108] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[109] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[110] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han, "A stochastic computational multi-layer perceptron with backward propagation," *IEEE Transactions on Computers*, vol. 67, no. 9, pp. 1273–1286, 2018.

[111] Y. Liu, L. Liu, F. Lombardi, and J. Han, “An energy-efficient and noise-tolerant recurrent neural network using stochastic computing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[112] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[113] T.-J. Yang, Y.-H. Chen, and V. Sze, “Designing energy-efficient convolutional neural networks using energy-aware pruning,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5687–5695.

[114] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, “Pruning convolutional neural networks for resource efficient inference,” *arXiv preprint arXiv:1611.06440*, 2016.

[115] R. Yu, A. Li, C. F. Chen, J. H. Lai, V. I. Morariu, X. Han, M. Gao, C. Y. Lin, and L. S. Davis, “Nisp: Pruning networks using neuron importance score propagation,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9194–9203, 2018.

[116] A. Krizhevsky, V. Nair, and G. Hinton, “The CIFAR-10 dataset,” *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[117] T. Brack, M. Alles, T. Lehnigk-Emden, F. Kienle, N. Wehn, N. E. L’Insalata, F. Rossi, M. Rovini, and L. Fanucci, “Low complexity LDPC code decoders for next generation standards,” *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, 2007.

[118] J. Andrade, G. Falcao, V. Silva, and L. Sousa, “A survey on programmable LDPC decoders,” *IEEE Access*, vol. 4, pp. 6704–6718, 2016.

[119] T. Richardson and S. Kudekar, “Design of low-density parity check codes for 5G new radio,” *IEEE Communications Magazine*, vol. 56, no. 3, pp. 28–34, 2018.

Appendix A

A.1 Proof of (5.3)

A positive integer N can be written as given by (2.8). On the other hand, according to the Mitchell method, $log_2(1+x) \approx x$. Hence, the approximation error E_N for N can be calculated as follows:

$$E_N = log_2(1+x) - x. \tag{A.1}$$

Since $2^k \leq N < 2^{k+1}$, N can be rewritten as $N = 2^k + i$, where $i \in \{0, 1, 2, ..., (2^k - 1)\}$. Therefore, by using (2.8), x can be represented by:

$$x = \frac{N}{2^k} - 1 = \frac{i}{2^k}. \tag{A.2}$$

The MSE can then be calculated for all of the $2^k \leq N < 2^{k+1}$ values as:

$$MSE_{2^k \leq N < 2^{k+1}} = \frac{1}{2^k} \times \sum_{i=0}^{2^k-1} \left(log_2(1 + \frac{i}{2^k}) - \frac{i}{2^k} \right)^2. \tag{A.3}$$

The summation over k in (5.3) is provided to cover the entire input range for an 8-bit design.

A.2 Proof of (5.4)

For the proposed method, the domain of N, $2^k \leq N < 2^{k+1}$, needs to be evenly divided into two intervals. In the first half, N is closer to 2^k and in the second half it would be closer to 2^{k+1}. The approximation function for each of these intervals is given in (5.2).

Keeping in mind that $N = 2^k + i$ where $i \in \{0, 1, 2, ..., (2^k - 1)\}$, the errors for the first half, i.e. $i \leq 2^{k-1} - 1$, are similar to those in the Mitchell method, and therefore, (A.3) is valid. For the second half, i.e. $i \geq 2^{k-1}$, the second part of (5.2) should be used. Based on (2.8) and (5.2), y can be represented as:

$$y = 1 - \frac{N}{2^{k+1}} = \frac{2^k - i}{2^k + 1}. \tag{A.4}$$

Thus, the total error can be calculated in a similar way to (A.3), as given by:

$$\begin{aligned} MSE_{2^k \leq N < 2^{k+1}} = \frac{1}{2^k} \times \Bigg[\sum_{i=0}^{2^{k-1}-1} \left(log_2(1 + \frac{i}{2^k}) - \frac{i}{2^k} \right)^2 \\ + \sum_{i=2^{k-1}}^{2^k-1} \left(log_2(\frac{2^k + i}{2^{k+1}}) - \frac{2^k - i}{2^{k+1}} \right)^2 \Bigg]. \end{aligned} \tag{A.5}$$

The summation over k can also be added to cover the entire input range for an 8-bit design, see (5.4).