

Image Processing using Approximate Data-path Units

by

Madhu Vasudevan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2013 by the
Graduate Supervisory Committee:

Chaitali Chakrabarti, Chair
David Frakes
Sandeep Gupta

ARIZONA STATE UNIVERSITY

December 2013

ABSTRACT

In this work, we present approximate adders and multipliers to reduce data-path complexity of specialized hardware for various image processing systems. These approximate circuits have a lower area, latency and power consumption compared to their accurate counterparts and produce fairly accurate results. We build upon the work on approximate adders and multipliers presented in [23] and [24]. First, we show how choice of algorithm and parallel adder design can be used to implement 2D Discrete Cosine Transform (DCT) algorithm with good performance but low area. Our implementation of the 2D DCT has comparable PSNR performance with respect to the algorithm presented in [23] with ~35-50% reduction in area. Next, we use the approximate 2x2 multiplier presented in [24] to implement parallel approximate multipliers. We demonstrate that if some of the 2x2 multipliers in the design of the parallel multiplier are accurate, the accuracy of the multiplier improves significantly, especially when two large numbers are multiplied. We choose Gaussian FIR Filter and Fast Fourier Transform (FFT) algorithms to illustrate the efficacy of our proposed approximate multiplier. We show that application of the proposed approximate multiplier improves the PSNR performance of 32x32 FFT implementation by 4.7 dB compared to the implementation using the approximate multiplier described in [24]. We also implement a state-of-the-art image enlargement algorithm, namely Segment Adaptive Gradient Angle (SAGA) [29], in hardware. The algorithm is mapped to pipelined hardware blocks and we synthesized the design using 90 nm technology. We show that a 64x64 image can be processed in 496.48 μ s when clocked at 100 MHz. The average PSNR performance of our implementation using accurate parallel adders and multipliers

is 31.33 dB and that using approximate parallel adders and multipliers is 30.86 dB, when evaluated against the original image. The PSNR performance of both designs is comparable to the performance of the double precision floating point MATLAB implementation of the algorithm.

To my parents for giving me the strength to pursue my dreams,
my sister and brother-in-law, for being the constant source of inspiration
and my dear friend Kannan Sha for his unquestionable faith in me.

This work is also affectionately dedicated to the six different
computers that were used to complete this work.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Dr. Chaitali Chakrabarti, for her excellent guidance, patience and kindness and providing a nurturing environment to conduct research. I could not have imagined a better mentor than her. She has been the biggest motivation to me for the last two years and has helped me grow as a professional as well as a better person. This work could not have been completed without her vision, insights and guidance. I would also like to thank NSR CSR0910699 for the financial support.

My sincerest gratitude to my committee members Dr. David Frakes and Dr. Sandeep Gupta for their kind encouragement.

I am grateful to my mentors, Amrit Panda, Yunus Emre and Christine Zwart, for their guidance and encouragement. I am thankful to my colleagues, Anirudh Yellamraju, Chegen Yang, Hsing-Min Chen, Manqing Mao, Ming Yang and Siyuan Wei, for their support and patience towards my questions. I also owe them thanks for creating a fun and relaxed environment in the lab.

I am thankful to my friends, Ashwin Ramanan, Pavithra Ramamoorthy, Preethi Natarajan, Priya Shanmugham and Vinodhini Raveendran for their unconditional support through thick and thin.

I would like to express my gratitude to Casey Gonzalez and Joseph Wagner, for their kindness, generosity and encouragement.

Lastly, I would like to thank my faculty members as well as the administrative staff, at Arizona State University, for providing an environment conducive for learning and professional development.

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER	
1 INTRODUCTION	1
1.1 Thesis Organization	5
2 APPROXIMATE ADDERS	6
2.1 Prior Work.....	6
2.2 Approximate Full Adder Circuits	7
2.2.1 Approximation 1	7
2.2.2 Approximation 2	8
2.2.3 Approximation 3	9
2.2.4 Approximation 4	9
2.2.5 Approximation 5	10
2.3 Area Complexity	10
2.4 Building Parallel Adders.....	11
3 APPROXIMATE MULTIPLIER	13
3.1 Reference Approximate Multiplier.....	13
3.1.1 2x2 Multiplier	13
3.1.2 Building Larger Multipliers.....	14
3.2 Proposed Multiplier	14
3.3 Comparison of Hardware Complexity	17

4	CASE STUDIES	20
4.1	Discrete Cosine Transform (DCT)	21
4.1.1	DCT – Our Implementation.....	22
4.1.2	DCT – Reference Paper Implementation for 16 bits data-path	25
4.1.3	Comparison of Approximation 5 with truncation.....	27
4.1.4	Area Complexity	29
4.1.5	Summary	30
4.2	Gaussian FIR Filter	31
4.2.1	Results	32
4.3	Fast Fourier Transform (FFT)	33
4.3.1	Split Radix FFT.....	33
4.3.2	Results	34
5	IMPLEMENTATION OF SAGA ALGORITHM	36
5.1	SAGA Algorithm.....	36
5.1.1	Determination of Optimal Displacements.....	37
5.1.2	Computing Displacements.....	38
5.1.3	Constructing Intermediate Images.....	39
5.2	Our Implementation	40
5.2.1	Sobel Operator	42
5.2.2	Computing the J matrix	43
5.2.3	Computing $J^T J$ and $J^T(-I_y)$	44
5.2.4	Calculating nodal displacements α_L	44
5.2.5	Calculating displacements α	46

5.2.5	Calculating nodal displacements α_L	46
5.2.6	Linear Interpolation to compute output pixel values	46
5.3	Timing Analysis of our hardware architecture.....	46
5.4	Hardware Synthesis and Performance Results.....	48
6	CONCLUSION	50
6.1	Summary	50
6.2	Future Work	52
	REFERENCES.....	53

LIST OF TABLES

Table	Page
1. Truth Table for accurate and approximate adders for all input combinations [23]...	18
2. Layout of accurate and approximate FA cells [23]	21
3. Layout of accurate and approximate FA cells for our implementation [23]	22
4. Comparison of hardware complexity	29
5. Implementation of DCT coefficients	31
6. Results for DCT – our implementation compared to implementation in [23]	34
7. Comparing the results for DCT – our vs ref implementation for 16 bit datapath	36
8. Results for DCT – truncation vs Approximation 5	38
9. Area Complexity – comparison of # of Fas used in the three configurations	39
10. Results for Gaussian FIR Filter.....	42
11. Results for 32 point 2D FFT	45
12. Timing Analysis of our hardware architecture.....	57
13. Synthesis results of all the hardware modules.....	58
14. Results for SAGA.....	59

LIST OF FIGURES

Figure	Page
1. Conventional Mirror Adder circuit [23]	17
2. Approximation 1 Adder circuit [23].....	17
3. Approximation 2 Adder circuit [23].....	19
4. Approximation 3 Adder circuit [23].....	19
5. Approximation 4 Adder circuit [23].....	20
6. 16 bit parallel adder using 10 accurate FA and 6 approximate FA	21
7. Karnaugh Map of the 2x2 approximate multiplier	23
8. Building a 4x4 approximate multiplier from 2x2 multiplier [24]	24
9. Accurate, reference and proposed approximate 2x2 multiplier	25
10. Building proposed 8x8 multiplier using 2x2 multipliers	26
11. Percentage errors of the proposed and reference 8x8 multiplier	27
12. Schematic for accurate, reference and proposed 2x2 multiplier	28
13. Architecture for first stage butterfly computation of DCT coefficients ...	33
14. Comparison - area of our implementation vs 16, 20 bit reference	37
15. Comparison – PSNR and area our implementation, 16, 20 bit reference ..	40
16. Convolution kernel for 5x5 Gaussian FIR Filter	41
17. Butterfly structure used for SRFFT	44
18. Isophote curvature – SAGA algorithm.....	47
19. Interpolation of input pixel values along the displacements	49
20. Proposed hardware architecture of SAGA	51
21. Block diagram for hardware implementation of Sobel operator	52

22.	Computation of J matrix from $\text{diag}(\mathbf{I}_x)$ and θ	54
23.	Hardware architecture for TDMA1 and TDMA2	55
24.	Timing Analysis of our hardware architecture	58

CHAPTER 1

INTRODUCTION

In order to reduce the area, latency and power of Digital Signal Processing implementations, several techniques have been adopted. The popular ones are reduction in number of computations [1-6], dynamic range adjustment of data-path units or truncation [6-10] and voltage scaling [11-18]. Recently approximate circuits have been proposed in [19-27] which also reduce area, latency and power.

The ‘almost correct adder’ in [19] is based on the observation that on average, the longest carry propagating sequence is approximately $\log n$, where n is the datapath width. Therefore, the sum output is independent of any carry beyond the $\log n$ sequence and hence the carry need not be propagated. They also provide a methodology to detect and correct errors. The approximate adder in [20] utilizes logic synthesis to design approximate versions of a given function. The algorithm determines the minterms that produce an approximate version of a circuit with smallest number of literals for a given error rate threshold. A bio-inspired imprecise adder and multiplier are proposed in [21] that provide an estimation of the result instead of its real value. An approximate adder and an approximate Booth multiplier (based on the approximate adder) are proposed in [22] and applied to the pipeline stages of a superscalar processor. A prediction unit predicts the error in the early stages of the pipeline and uses the accurate result instead. In [23], several approximations to a full adder circuit are described by applying complexity reduction at transistor level. The DCT implementation results provided in this paper demonstrate that this is a very promising approach. An inaccurate 2×2 multiplier based on Karnaugh Map simplification is proposed in [24], which acts as the building block for

larger multipliers. In [25], a systematic methodology is presented for automatic logic synthesis of approximate circuits, that synthesizes an approximate version of a given RTL specification for a given quality constraint. A statistical error analysis and model is presented in [26] and [27].

A very important consideration in the adoption of these techniques is that they cause minimal degradation to the algorithm performance. Fortunately, image processing algorithms are typically error resilient and such techniques are well suited for these algorithms. However indiscriminate use of these techniques affects the accuracy and thus the algorithm performance. This work attempts to judiciously use approximate adders and multipliers, such that the algorithm performance is only mildly affected. It builds on the approximate adder and multiplier circuits introduced in [23] and [24] and shows how modified versions of these circuits can be successfully used in minimizing the area and latency of key image processing systems. Note that reduced latency can be translated into increased throughput or reduced power consumption through voltage scaling.

First we study approximate full adder (FA) circuits where the approximations are achieved by applying logic complexity reduction at transistor level [23]. These are used to implement multi-bit parallel adders where the approximate FA are used only for the least significant bits. The approximate FAs help reduce the area and latency of the parallel adder but generate accurate results in most cases. This makes them more effective than the popular technique of truncation.

We use approximate parallel adders to implement the 2D Discrete Cosine Transform algorithm which is used in standards such as the Joint Photographic Expert

Group (JPEG) standard, lossy image and video compression, digital watermarking, and face recognition. DCT is also widely used in solving partial differential equations and Chebyshev approximations.

Compared to the implementation in [23], we reduce the width of the data-path from 20 bits to 16 bits, use a mixture of accurate and approximate adders and re-use several computations. We implement the proposed method and the method presented in [23] in hardware. We compare the results generated by the hardware implementation for both methods with those generated by using double precision floating point MATLAB. We find that by applying these three techniques, we are able to achieve up-to 50% reduction in area and an average improvement of 2-7 dB in PSNR compared to [23] for various standard test images.

For a fairer comparison, we also implement the reference algorithm on a 16 bit wide data-path and compare its performance with our implementation. We observe that, our implementation has a 40-45% reduction in area and comparable PSNR performance for the various configurations. We also compare the implementation using a parallel adder with the most aggressive approximate FA with an implementation using truncation, as they both have comparable area. We conclude that approximate adders perform much better compared to truncation especially for more aggressive scenario where several LSBs are truncated. For instance, in a 16 bit parallel adder implementation, use of 4 approximate FAs results in an average PSNR of 34.38 dB which is more than 10 dB higher compared to 23.46 dB obtained by truncating 4 LSBs.

Next, we implement the approximate multiplier presented in [24] in hardware. The approximate multiplier is achieved by Karnaugh map simplification and has a lower area

and latency compared to the accurate 2×2 multiplier. This 2×2 multiplier acts as a building block in constructing larger multipliers. We propose three enhancements to this multiplier to achieve appreciable improvement in performance. We synthesize the accurate, reference and proposed multipliers in hardware using DC Compiler, and evaluate them based on area, latency and power. For the 2×2 multiplier unit, we achieve nearly 28% and 63% reduction in area, and 25% and 63% reduction in power compared to the reference and accurate multipliers, respectively.

We study the performance of the approximate multipliers by using them in the hardware implementation of the Gaussian FIR Filter and the 2D Fast Fourier Transform algorithms. We choose these two algorithms because of their use in large number of DSP systems. FIR filters are used in image processing, digital wireless communication such as Global System for Mobile Communications (GSM), hearing aids, and digital video broadcast. Fast Fourier Transform is considered to be one of the top ten most important algorithms of the twentieth century and finds applications in but not restricted to, communications, astronomy, geology, optics, etc. Specifically, it can be applied for spectral analysis, data compression, solving partial differential equations, polynomial multiplication, etc.

For FFT, we show that our implementation achieves 4.7 dB improvement in PSNR performance with ~5% area overhead compared to [24].

We also propose a hardware architecture of the interpolation algorithm, Segment Adaptive Gradient Angle (SAGA) [29]. We study each of the algorithm blocks and evaluate their sensitivity to data-path precision. We map the algorithm into pipelined hardware blocks and synthesize them using 90 nm technology. We show that a 64×64

image can be processed in 496.48 μ s when clocked at 100 MHz. This implementation achieves an average PSNR of 31.33 dB compared to the original image. The floating point double precision MATLAB implementation of SAGA achieves an average PSNR of 32.14 dB which is comparable to our performance.

1.1. Thesis Organization

The layout of the work is organized as follows. In Chapter 2, we describe the various approximate full adders presented in [23] and present their area complexities. In Chapter 3, we present the approximate multiplier proposed in [24]. We discuss the enhancements proposed to this multiplier and compare the performance of the reference and proposed multipliers. We also compare the hardware complexity of the three multipliers in terms of area, latency and power. In Chapter 4, we apply these approximate adder and multiplier circuits to popular image processing algorithms such as Discrete Cosine Transform, Gaussian FIR Filter, and Fast Fourier Transform. We evaluate the performance of the various approximate implementations against the double precision floating point MATLAB implementation. In Chapter 5, we describe the SAGA algorithm and its hardware implementation in detail. We conclude the thesis in Chapter 6 and provide pointers for further research.

CHAPTER 2

APPROXIMATE ADDERS

In order to reduce the complexity of the data-path unit, approximations have been proposed for the full adder (FA) circuit in [23]. In this chapter, we study these approximations and analyze their error characteristics and area complexity. These approximate adders have been used to implement the Discrete Cosine Transform algorithm in Chapter 4 and the performance of the corresponding implementation evaluated.

2.1. Prior Work

In the approximation method outlined in [23], the Conventional Mirror Adder (MA) circuit shown in Figure 1 is used as the reference design. Here A and B are the primary inputs, Cin is the carry in from the previous stage. Sum' is the complemented Sum signal and Cout' is the complement carry out signal that is sent to the next stage. The approximate circuits are obtained by systematically removing transistors from this accurate full adder. The two rules followed in this process are [23]:

- i. No combinations of the inputs A, B and Cin should result in an open or short circuit,
- ii. The simplified circuit should result in minimum number of errors in Sum and Cout.

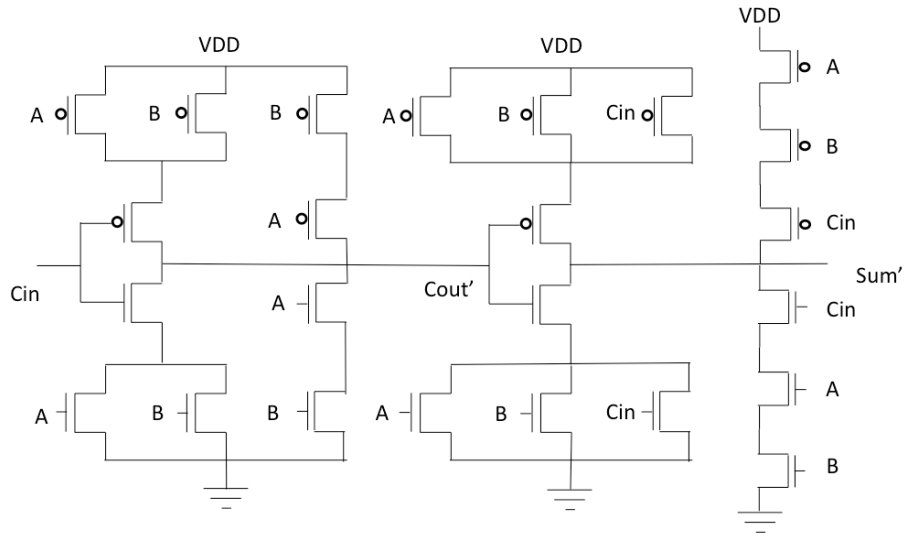


Figure 1. Conventional Mirror Adder circuit [23]

2.2. Approximate Full Adder Circuits

2.2.1. Approximation 1

Approximation 1 has 8 fewer transistors than the reference design. It introduces two errors in the Sum and one error in the Cout outputs. The schematic for this approximation is shown in Figure 2.

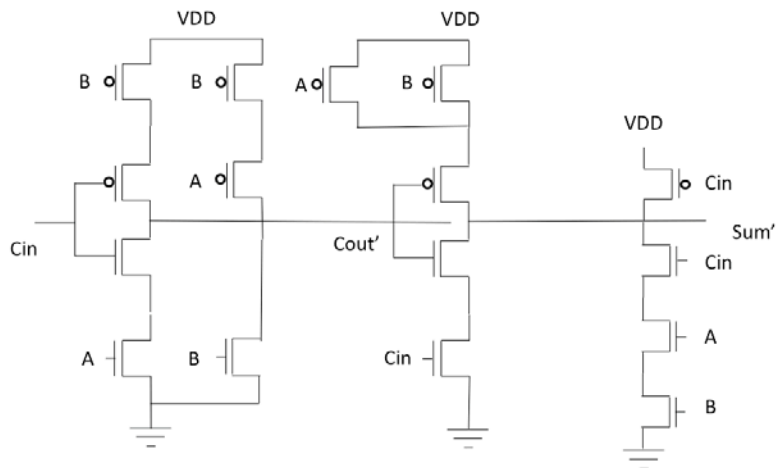


Figure 2. Approximation 1 Adder circuit [23]

TABLE I. TRUTH TABLE SHOWING OUTCOMES FOR ACCURATE AND APPROXIMATE ADDERS FOR ALL POSSIBLE INPUT COMBINATIONS [23]

Inputs			Accurate Outputs		Approximate Outputs							
<i>A</i>	<i>B</i>	<i>Cin</i>	<i>Sum</i>	<i>Cout</i>	<i>Sum1</i>	<i>Cout1</i>	<i>Sum2</i>	<i>Cout2</i>	<i>Sum3</i>	<i>Cout3</i>	<i>Sum4</i>	<i>Cout4</i>
0	0	0	0	0	✓	✓	✗	✓	✗	✓	✓	✓
0	0	1	1	0	✓	✓	✓	✓	✓	✓	✓	✓
0	1	0	1	0	✗	✗	✓	✓	✗	✗	✗	✓
0	1	1	0	1	✓	✓	✓	✓	✓	✓	✗	✗
1	0	0	1	0	✗	✓	✓	✓	✓	✓	✗	✗
1	0	1	0	1	✓	✓	✓	✓	✓	✓	✓	✓
1	1	0	0	1	✓	✓	✓	✓	✓	✓	✓	✓
1	1	1	1	1	✓	✓	✗	✓	✗	✓	✓	✓

2.2.2. Approximation 2

The truth table for Approximation 1 shows that $\text{Sum} = \neg \text{Cout}$ for 6 of the 8 combinations.

This forms the basis for Approximation 2. Since direct assignment leads to a greater input gate capacitance compared to the original reference adder, a buffer is introduced.

The resulting circuit is shown in Figure 3.

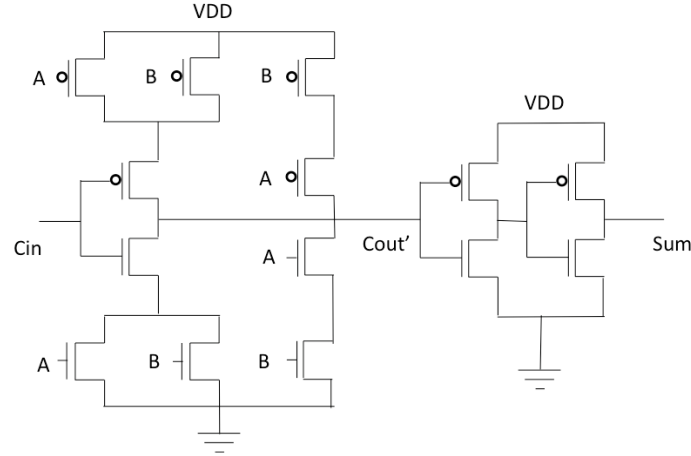


Figure 3. Approximation 2 Adder circuit [23]

2.2.3. Approximation 3

Approximation 3 is obtained by combining approximations 1 and 2. The resulting circuit has 13 fewer transistors compared to the reference adder. This approximation has three errors in Sum and one error in Cout.

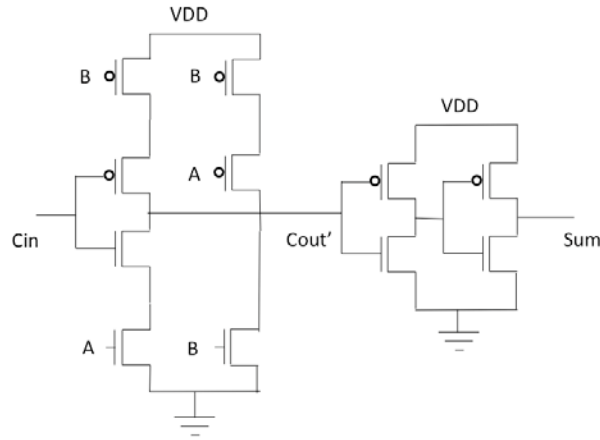


Figure 4. Approximation 3 Adder circuit [23]

2.2.4. Approximation 4

From Table I we see that $Cout = A$ for six out of the eight possible combinations. Approximation 4 is the result of combining this with the expression for Sum from

Approximation 1. The resulting approximation has 11 transistors. This results in three errors in Sum and two errors in Cout.

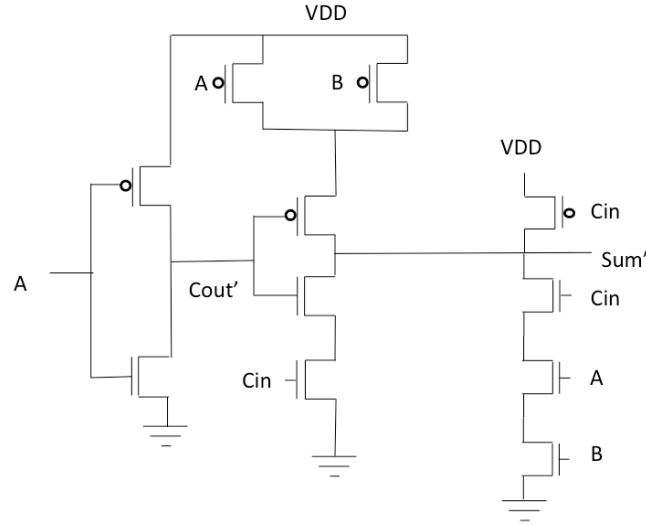


Figure 5. Approximation 4 Adder circuit [23]

2.2.5. Approximation 5

The motivation for Approximation 5 is to avoid carry propagation altogether by making Sum independent of Cin. To achieve this, the Sum output is approximated to B and Cout is approximated to A. This approximation is therefore reduced to an assignment of outputs to inputs and requires no logic gates. This is the most aggressive of all the approximations and yet results in the best performance.

2.3. Area Complexity

All the proposed approximations result in significant reduction in area. The number of transistors and the layout area in IBM 90nm technology for the reference and approximate adders is shown in Table II.

TABLE II. LAYOUT OF ACCURATE AND APPROXIMATE FA CELLS [23]

FA Unit	Number of Transistors	Area (μm^2)
Reference MA	24	40.66
Approximation 1	16	29.31
Approximation 2	14	25.5
Approximation 3	11	22.56
Approximation 4	11	23.91

2.4. Building Parallel Adders

For our study, we implement an approximate 16 bit parallel adder using the ripple carry adder architecture. Here the addition of Least Significant Bits (LSB) is implemented using approximate FAs and the addition of the Most Significant Bits (MSB) is implemented using the reference mirror adder (MA). An approximate parallel adder is shown in Figure 6. Here, the 10 MSB additions are implemented using the reference adder which is accurate, and the 6 LSBs are implemented using approximate FAs. We use this configuration as our reference parallel adder in the rest of the document.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A	C	C	U	R	A	T	E	F	A	A	P	P	R	O	X

Figure 6. 16 bit parallel adder using 10 accurate FA and 6 approximate FA

For the reference 16 bit parallel adder, the total number of transistors is shown in Table III. The area of the different approximate implementations is also shown in the table. This area is computed based on the layout area for accurate as well as approximate

cells given in Table II. We can see that the area reduction obtained using approximate adders is 10-35% compared to the accurate parallel adder.

TABLE III. LAYOUT OF ACCURATE AND APPROXIMATE FA CELLS FOR OUR IMPLEMENTATION [23]

Parallel Adder Unit	Number of Transistors	Area (μm^2)
Reference MA	384	650.6
Approximation 1	336	582.5
Approximation 2	324	559.6
Approximation 3	306	541.9
Approximation 4	306	550.1
Approximation 5	240	406.6

CHAPTER 3

APPROXIMATE MULTIPLIER

In this chapter we build upon the approximate multiplier presented in [24]. We describe the reference multiplier design and propose enhancements to the reference multiplier. We compare the performance of the two approximate designs and the accurate multiplier with respect to area, latency and power.

3.1. Reference Approximate Multiplier

3.1.1. 2x2 Multiplier

The basic building block of the approximate multiplier is the 2x2 multiplier, which multiplies two 2 bit words (a_1a_0) and (b_1b_0). This produces 4 bits of outputs since the largest number generated by a 2x2 multiplication is 9 (1001). In [24], an approximation to the 2x2 multiplier is introduced by approximating this multiplication value to 7, which can be represented with 3 bits (111). By restricting the output of the 2x2 multiplier from 4 to 3 bits greatly reduces the complexity and introduces only a small error – only one in 16 possible combinations is erroneous. Figure 7 shows the modified Karnaugh map of the approximated logic function. The erroneous output is highlighted in red.

a1a0 \ b1b0	b1b0			
	00	01	10	11
00	000	000	000	000
01	000	001	011	010
10	000	011	111	110
11	000	010	110	100

Figure 7. Karnaugh Map of the 2x2 approximate multiplier [24]

3.1.2. Building Larger Multipliers

Larger multipliers are built by using the 2x2 approximate multiplier as the basic building block. The 2x2 module is used to generate the partial products of the larger multiplier. The partial products are then added using an adder tree built with accurate adders. This architecture is illustrated in Figure 8.

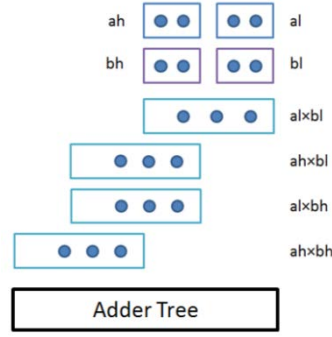
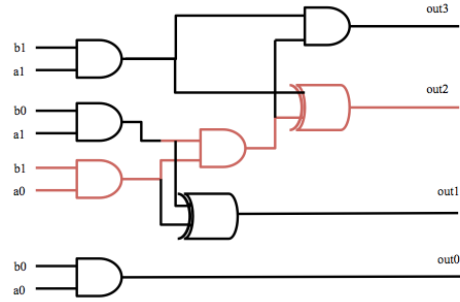


Figure 8. Building a 4x4 approximate multiplier from 2x2 multiplier [24]

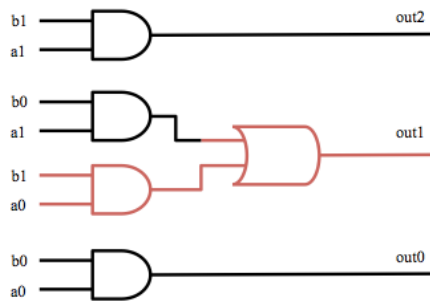
3.2. Proposed Multiplier

We propose three changes to this multiplier. First, we further approximate the 2x2 building block by approximating out_0 to 0. Even though the critical path of the resulting multiplier remains the same, the area has reduced. Figure 9 shows the logic functions of the accurate, reference [24] and proposed 2x2 multiplier.

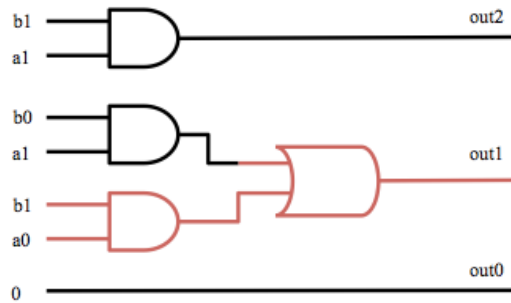
The second enhancement that we propose is the way a larger multiplier is built. The reference multiplier uses the same 2x2 approximate multiplier to compute all partial products. Instead, we introduce three levels of approximation within the larger multiplier. We calculate the least significant partial product with maximum degree of approximation, the middle partial products with medium approximation and the most significant partial product with no approximation.



(a)



(b)



(c)

Figure 9. (a) Accurate (b) Reference approximate and (c) Proposed approximate 2x2 Multiplier. The critical paths have been highlighted in red in all three cases [24].

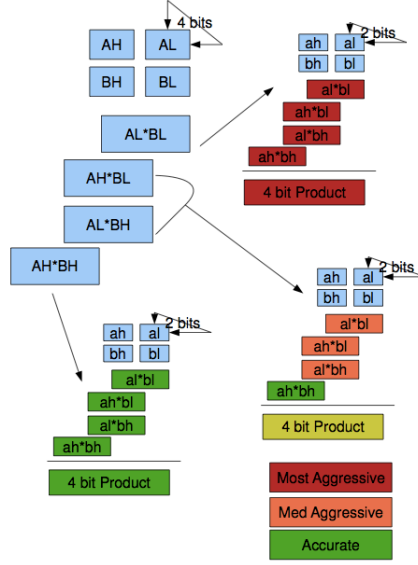


Figure 10. Building proposed 8x8 multiplier using the 2x2 multipliers. The figure shows the varying degree of approximation in the multiplier.

Figure 10 describes the proposed architecture. Compared to the reference multiplier, only the lower partial products are computed with approximation. This improves the accuracy of the proposed multiplier compared to the reference multiplier when two large numbers are multiplied. However, when two small numbers are multiplied, the accuracy drops since we use aggressive approximations to compute the LSB partial product. In image processing applications involving filter computations, one of the inputs to the multiplier is a filter coefficient which is a known constant in many cases. The other input is typically the image pixel value and is a variable. If we have prior information about the filter coefficient, we can adjust the level of approximation within the multiplier. We propose two versions of the approximate multiplier.

1. Version A: Computes the product with medium approximation (using the reference 2x2 multiplier) when the filter coefficient is less than 16.

2. Version B: Computes the product with most aggressive approximation (proposed multiplier) when the filter coefficient is greater than 15.

By introducing two versions of the multiplier, we are able to achieve good accuracy when the filter coefficient is small or large. To validate this claim, we evaluate our 8x8 multiplier against the reference multiplier by sweeping the two inputs A & B from 0-255 and compute the percentage error for each input bin. Figure 11 illustrates the error (in percentage) for the reference and proposed 8x8 multipliers for different input combinations. We see that the proposed multiplier has the same performance as that of the reference multiplier when either (or both) input is small. However, the proposed multiplier performs much better than the reference multiplier for large inputs.

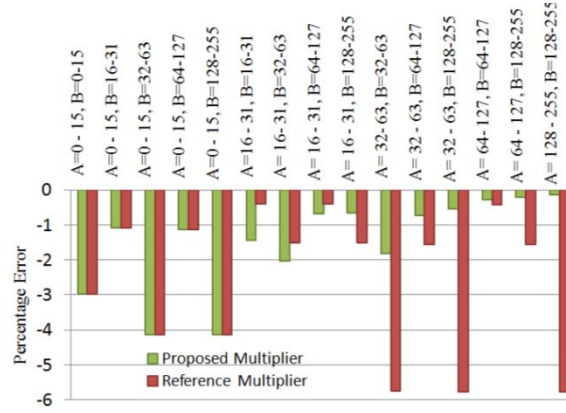
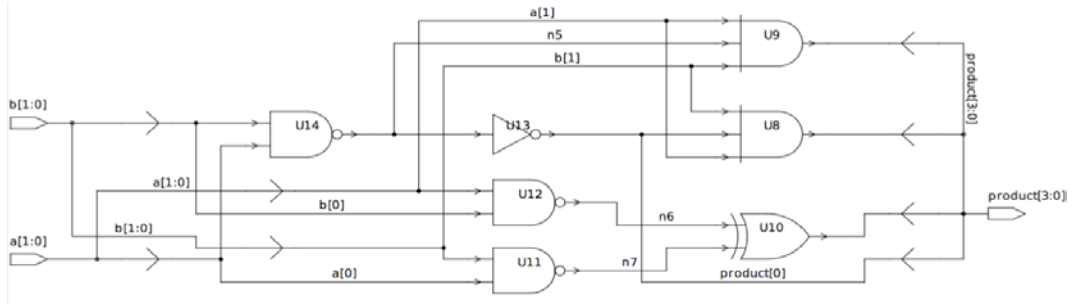


Figure 11. Percentage errors of the proposed and reference 8x8 multiplier for various input combinations

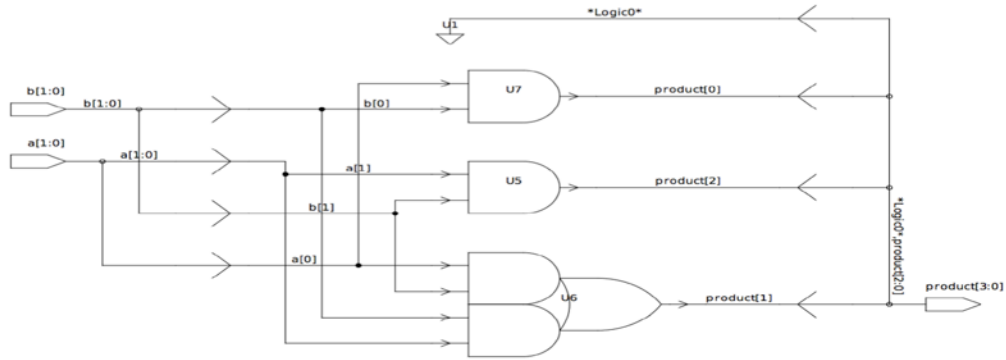
3.3. Comparison of Hardware Complexity

In this section we compare the hardware complexity of the accurate, reference and proposed multiplier for the 2x2 and 8x8 configurations. The metrics used are area, latency and power, obtained on DC Compiler using the SAED 90nm Generic Library

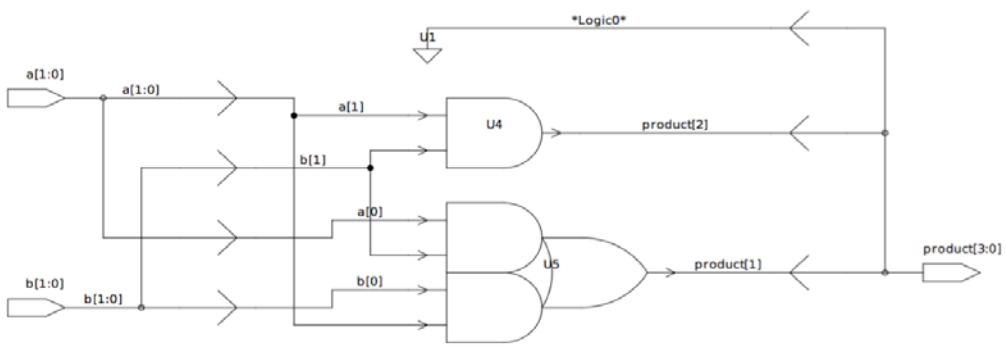
(optimized for power). The schematics of the synthesized 2x2 accurate, reference and approximate multiplier are shown in Figure 12.



(a)



(b)



(c)

Figure 12. Schematic generated after synthesis for (a) Accurate, (b) Reference Approximate and (c) Proposed 2x2 Multiplier

Table V presents the area, latency and dynamic power for the different multipliers. We can see that the proposed 2x2 multiplier is 63.16% and 27.58% smaller in area and 63.21% and 25.86% lower in power compared to the accurate and reference multiplier designs, respectively. The proposed 8x8 multiplier has a 3% overhead compared to the 8x8 reference multiplier due to the introduction of accurate 2x2 multipliers and has 10.25% smaller area compared to the 8x8 accurate multiplier. The area reduction is significantly smaller than the results quoted in [24]. Part of the reason could be that in [24], the larger multiplier and adder trees are built using the RTL-Compiler (RC Compiler), which leads to an optimized design.

The power savings for the 8x8 proposed multiplier is ~2% and ~8.2% compared to the 8x8 reference and accurate multiplier respectively. Thus, while the accuracy of the proposed multiplier is comparable or better than that of the reference multiplier, the overhead incurred in terms of area and latency is negligible.

TABLE IV. COMPARISON OF HARDWARE COMPLEXITY

Multiplier	Area – combinational (nm ²)	Latency (ns)	Dynamic Power (uW) for V=1.2V
2x2 Accurate Multiplier	52.53	0.69	6.70
2x2 Reference Multiplier	26.72	0.53	3.325
2x2 Proposed Multiplier	19.35	0.53	2.465
8x8 Accurate Multiplier	3712.20	5.25	344.22
8x8 Reference Multiplier	3251.40	5.15	322.33
8x8 Proposed Multiplier	3331.58	5.15	316.27

CHAPTER 4

CASE STUDIES

In this chapter, we implement image processing kernel algorithms such as the Gaussian FIR Filter, 2D DCT and 2D FFT using approximate adders and multipliers. The algorithms have been implemented on customized hardware architecture using Verilog VHDL and synthesized using Design Compiler from Synopsys. The area, latency and power estimations are obtained using the Design Compiler for the SAED 90nm Generic Library (optimized for power).

The performance of the algorithms is evaluated using Peak Signal to Noise Ratio (PSNR). The ground truth corresponds to double precision floating point MATLAB implementation. For an image of size $M \times N$, if $I(i,j)$ is the value obtained from MATLAB, at location (i,j) , and $O(i,j)$ is the value obtained from the fixed-point hardware implementation of the algorithm, and MAX is the maximum possible value obtained from MATLAB implementation, then PSNR is defined as:

$$MSE = \frac{1}{MN} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} [I(i,j) - O(i,j)]^2$$

$$PSNR = 10 \log_{10} \frac{MAX^2}{MSE}$$

Therefore, PSNR represents the noise introduced due to fixed-point implementation as well as use of approximate circuits. Six test images of size 512x512 were used for evaluating performance of the system. They are Baboon, Barbara, Boat, House, Lena and Peppers [19].

This chapter is organized as follows. Use of the approximate FA circuits to

implement the Discrete Cosine Transform algorithm is presented in section 4.1. Use of the reference and proposed approximate multipliers in the implementation of the Gaussian FIR filter and 2D Fast Fourier Transform is described in section 4.2 and 4.3, respectively.

4.1. Discrete Cosine Transform (DCT)

The 2D 8-point DCT is implemented by applying 1D DCT along the rows and then applying 1D DCT along the columns. An N point 1D forward DCT can be expressed as

$$X(m) = u(m) \sqrt{\frac{2}{N}} \sum_{i=0}^{N-1} x(i) \cos \frac{(2i+1)m\pi}{2N}, \text{ for } m = 0, 1, \dots, N-1,$$

where $u(m) = \left\{ 1 \text{ for } m = 0 \left| \frac{1}{\sqrt{2}} \text{ otherwise.} \right. \right\}$. All the cosine coefficients are scaled by

32. The multiplication with the cosine coefficient is replaced with additions and shifts as shown in Table V.

TABLE V. IMPLEMENTATION OF DCT COEFFICIENTS

Coefficient	Value	Implementation
a	16	$1 \ll 4$
b	15	$(1 \ll 4) - 1$
c	14	$(1 \ll 4) - (1 \ll 2)$
d	11	$(1 \gg 2) + 1 (1 \ll 1) + (1 \ll 3)$
e	9	$1 + 1 \ll 3$
f	6	$(1 \ll 1) + (1 \ll 2)$
g	3	$1 + (1 \ll 1)$

4.1.1. DCT – Our Implementation

The 8-point DCT can be simplified into odd and even coefficients as shown below.

$$\begin{pmatrix} W0 \\ W2 \\ W4 \\ W6 \end{pmatrix} = \begin{pmatrix} d & d & d & d \\ b & f & -f & -b \\ d & -d & -d & d \\ f & -b & b & -f \end{pmatrix} \begin{pmatrix} x0 + x7 \\ x1 + x6 \\ x2 + x5 \\ x3 + x4 \end{pmatrix}$$

$$\begin{pmatrix} W1 \\ W3 \\ W5 \\ W7 \end{pmatrix} = \begin{pmatrix} a & c & e & g \\ c & -g & -a & -e \\ e & -a & g & c \\ g & -e & c & -a \end{pmatrix} \begin{pmatrix} x0 - x7 \\ x1 - x6 \\ x2 - x5 \\ x3 - x4 \end{pmatrix}$$

Several computations can be reused as in the butterfly structure shown in Figure 13.

The parallel adders used in the first stage butterfly structure that compute $y_0, y_1, y_2, \dots, y_7$, are chosen to be accurate, since these results are used in the rest of the computations. The figure also elaborates how the coefficients $W_0, W_1, W_2, W_3, W_4, W_5, W_6$, and W_7 are computed. All parallel adders used in the latter stages are approximate. The data-path of our system is 16 bits. The approximate 16 bit parallel adder uses approximate adders for the 6 LSBs and uses accurate adders for the 10 MSBs.

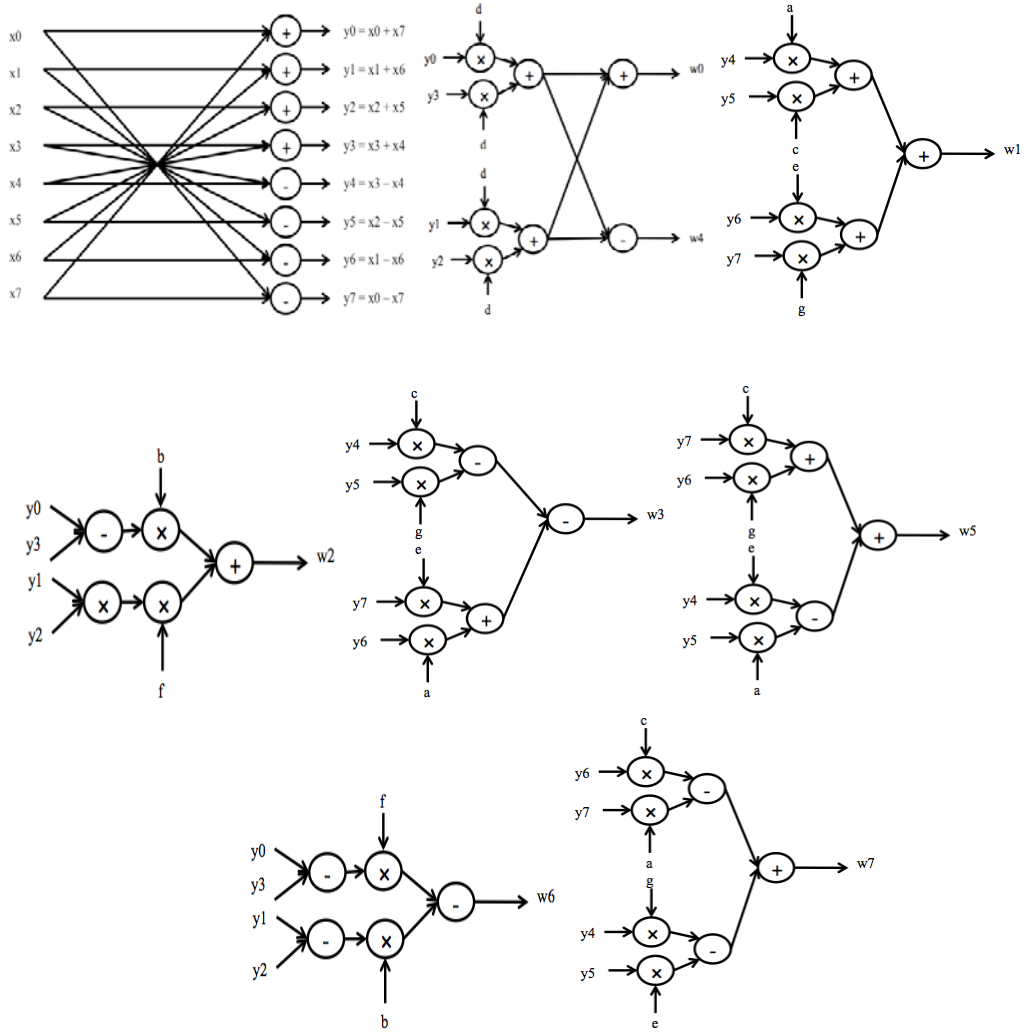


Figure 13. Architecture for first stage butterfly and computation of DCT coefficients

The PSNR results are obtained after applying forward-backward 2D DCT on six standard images and the performance of our DCT is compared with that in [23]. Note that the DCT implementation in [23] does not use the butterfly structure shown in Figure 13 and implements the algorithm in a straightforward manner using matrix vector multiplication. It uses a data-path that is 20 bits wide and all the 20 bit parallel adders are approximate. The PSNR performance results for various configurations with varying

degrees of approximation are presented in [23]. For comparison, we use the configuration that keeps the PSNR performance of the two implementations comparable. This corresponds to a 20 bit parallel adder that uses approximate FAs to add 9 LSBs and accurate FAs to add 11 MSBs. It should be noted that the PSNR performance of both systems can be improved by increasing the number of MSBs that are added accurately.

The PSNR results for the six standard images are presented in Table VI. Note that the PSNR results quoted for [23], given in Table VI, correspond to those obtained using our implementation of the algorithm in [23]. To validate our implementation, we compared the PSNR results of our implementation of the algorithm in [23] with the values quoted in [23]. The PSNR values obtained for the Lena image of our implementation are within ± 1 dB with the results presented in [23] and so our implementation is representative of their design.

TABLE VI. RESULTS FOR DCT – OUR IMPLEMENTATION (DATA-PATH = 16 BITS) COMPARED TO IMPLEMENTATION IN [23] (DATA-PATH = 20 BITS)

Image	PSNR (dB)											
	<i>Accurate</i>		<i>Approximation 1</i>		<i>Approximation 2</i>		<i>Approximation 3</i>		<i>Approximation 4</i>		<i>Approximation 5</i>	
	Our	[23]	Our	[23]	Our	[23]	Our	[23]	Our	[23]	Our	[23]
Baboon	34.61	30.71	29.10	24.27	25.15	23.44	25.97	18.49	22.70	15.63	27.78	26.74
Barbara	34.71	31.01	28.95	24.46	25.40	23.04	25.79	18.42	22.59	16.01	27.64	25.54
Boat	34.53	31.53	29.00	24.56	25.44	23.25	25.83	18.19	22.37	16.02	27.67	26.13
House	33.56	30.76	28.89	25.20	25.56	22.80	26.18	18.77	22.44	15.89	27.41	24.57
Lena	34.79	32.08	29.02	24.69	25.56	23.19	25.94	18.45	22.22	15.92	27.77	26.07
Peppers	35.03	32.36	29.13	24.89	25.72	22.84	26.13	18.29	22.59	15.96	27.92	26.37
Average	34.54	31.41	29.01	24.68	25.47	23.09	25.97	18.43	22.48	15.90	27.70	25.90

From Table VII, we see that our implementation performs better than the reference algorithm even though our data-path width is 16 bits compared to 20 bits in [23]. For parallel adder implementation using accurate FA, and FA based on approximations 1 and 2, we obtain an average improvement of 3.13 dB, 4.33 dB and 2.38dB. The biggest improvement is obtained for approximations 3 and 4 with 7.54 dB and 6.58 dB, respectively, and the smallest improvement is obtained for approximation 5 of 1.8 dB. In general, our implementation has better performance because we use accurate adders to compute the first stage of the algorithm. Also, our implementation has lower area since several computations are reused.

4.1.2. DCT – Reference Paper Implementation for 16 bits data-path

We also implement the DCT algorithm presented in [23] for a 16 bit wide data-path. The aim is to compare the results of the two implementations by keeping the data-path width of the system constant. In the 16 bit parallel adder, we use approximate adders for the 6 LSBs and use accurate adders for the 10 MSBs. Table VII compares the results obtained for our implementation against the 16 bit implementation of the reference algorithm.

TABLE VII. COMPARING THE RESULTS FOR DCT – OUR IMPLEMENTATION VS REFERENCE

IMPLEMENTATION (DATA-PATH = 16 BITS)

Image	PSNR (dB)											
	<i>Accurate</i>		<i>Approximation 1</i>		<i>Approximation 2</i>		<i>Approximation 3</i>		<i>Approximation 4</i>		<i>Approximation 5</i>	
	Our	Ref	Our	Ref	Our	Ref	Our	Ref	Our	Ref	Our	Ref
Baboon	34.61	30.67	29.10	29.76	25.15	28.07	25.97	25.54	22.70	22.33	27.78	28.78
Barbara	34.71	30.97	28.95	29.55	25.40	28.06	25.79	24.95	22.59	22.01	27.64	29.08
Boat	34.53	31.48	29.00	30.23	25.44	28.53	25.83	25.39	22.37	22.25	27.67	29.74
House	33.56	30.71	28.89	29.60	25.56	27.59	26.18	23.37	22.44	21.98	27.41	29.58
Lena	34.79	32.02	29.02	30.19	25.56	28.42	25.94	25.08	22.22	22.30	27.77	30.00
Peppers	35.03	32.30	29.13	30.38	25.72	28.29	26.13	25.18	22.59	22.40	27.92	30.21
Average	34.54	31.36	29.01	29.95	25.47	28.16	25.97	24.92	22.48	22.21	27.70	29.56

From Table VII, we see that while our implementation has better performance for the accurate case, it has worse performance for approximations 2 and 5 and comparable performance for approximations 1, 3 and 4. In our implementation, outputs of the first stage, y_4 , y_5 , y_6 and y_7 are quite small and therefore use of approximate adders in the LSBs of subsequent stages lead to decrease in quality performance.

Figure 14 shows a comparison of area of our implementation against the 16 bit and 20bit implementations of the reference algorithm, for the different approximations. The area estimates are computed from the layout area of various FA cells given in Table II. The area assessment corresponds to an estimate of the area occupied by the FAs and does not consider the routing overhead. It can be seen that our implementations show significant improvement in terms of area. For the various approximations, we achieve ~34% to 39% reduction in area, compared to [23] even when both data-paths are 16 bits

wide. For the various approximations, we achieve 39-50% reduction in area compared to the 20 bit implementation of the reference algorithm.

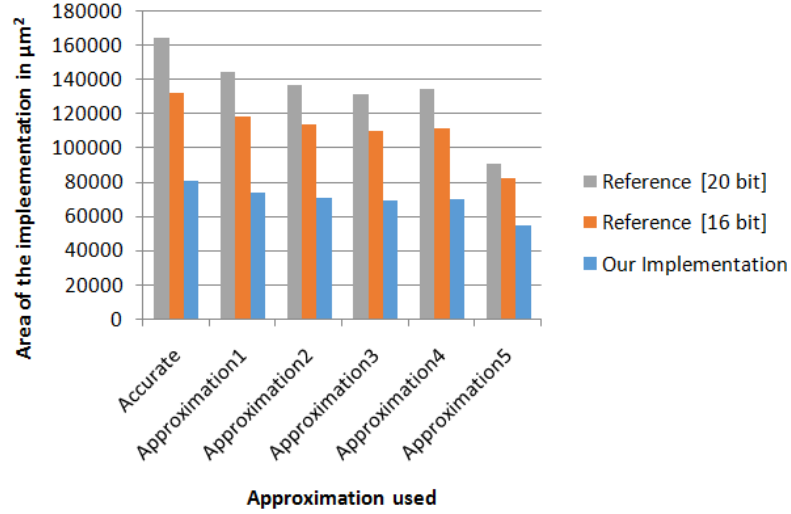


Figure 14. Comparison of area of our implementation vs 16bit and 20bit reference algorithm [23] implementations

4.1.3. Comparison of Approximation 5 with truncation

In this section, we incorporate truncation into our implementation of the DCT algorithm and compare it with the results obtained by using Approximation 5. Note that FAs based on Approximation 5 are built by re-routing the input signals and require no logic gates. If the routing overhead is ignored, then the area of an implementation that employs truncation by L bits and an implementation that approximates addition of L LSBs should be comparable.

In our implementation of truncation, the outputs of the first stage computations are truncated. The computation of the coefficients $W_0, W_1 \dots W_7$ remains the same as discussed before. All parallel adders used in this configuration are accurate. We study the impact of truncating 2, 4 and 6 LSBs on the quality performance. For an L-bit truncation,

the internal precision of the system is 16-L bits. Thus, for 2 bit, 4 bit and 6bit truncation, the internal precision is 14 bits, 12 bits and 10 bits, respectively.

The PSNR results for DCT computation using 2 bit, 4 bit and 6 bit truncation as well as 2 bit, 4 bit and 6 bit approximation using Approximation 5 FAs are shown in Table VIII. We see that the performance of the algorithm drops sharply with increase in degree of truncation. While 2 bit truncation results in some loss of performance, truncation of 6 bits results in significant loss in performance and is unacceptable. Use of Approximation 5 FAs on the other hand demonstrates very small degradation in PSNR performance for 2 and 4 bit approximation. For instance, use of 4 approximate FAs results in an average PSNR of 34.38 dB which is more than 10 dB higher compared to 23.46 dB obtained by truncating 4 LSBS of the 16 bit parallel adder.

TABLE VIII. RESULTS FOR DCT – FOR 2, 4 AND 6 BIT TRUNCATION AND APPROXIMATION USING APPROXIMATION 5

Image	PSNR (dB)						
	Accurate	2 bits		4 bits		6 bits	
		Truncation	Approx 5	Truncation	Approx 5	Truncation	Approx 5
Baboon	34.61	31.3	34.87	23.31	34.40	13.62	27.78
Barbara	34.71	31.50	35.03	23.57	34.51	14.03	27.64
Boat	34.53	31.31	34.86	23.46	34.34	13.10	27.67
House	33.56	30.71	33.87	23.19	33.61	13.47	27.41
Lena	34.79	31.51	35.14	23.51	34.62	13.72	27.77
Peppers	35.03	31.8	35.35	23.70	34.77	13.93	27.92
Average	34.54	31.36	34.85	23.46	34.38	13.65	27.70

4.1.4. Area Complexity

In this section we compare the total area of the three implementations by comparing the number of FAs used in each case. The algorithm used in the reference paper uses two hundred and two 20 bit parallel adders to implement 1D forward-reverse DCT while our implementation uses only one hundred and twenty four 16 bit parallel adders (sixteen are accurate and hundred and eight are approximate 16 bit parallel adders) to implement the same. Therefore, the area of our implementation decreases significantly with respect to the reference algorithm. Table IX lists the number of accurate and approximate FAs used in each configuration.

TABLE IX. AREA COMPLEXITY – COMPARISON OF # OF FAS USED IN THE THREE CONFIGURATIONS

Reference Implementation (data-path = 20 bits) 9 approximate LSBs		Reference Implementation (data-path = 16 bits) 6 approximate LSBs		Our Implementation (data-path = 16 bits) 6 approximate LSBs	
# of accurate FA	# of approximate FA	# of accurate FA	# of approximate FA	# of accurate FA	# of approximate FA
2222	1818	2020	1212	1336	648

Figure 15 plots the area and PSNR for the competing implementations. We use the average PSNR of all images obtained for each configuration for this analysis. Our design goal is to achieve high performance at low area cost and thus a system whose performance lies in the upper left corner on this plot is most desirable. From Figure 15, we can see that our implementation achieves this design goal. Our implementation generates better or comparable PSNR performance at much smaller area for all approximations. Other than Approximation 5, the area reduction using our algorithm is significant for all other configurations. For instance, for both 16 bit and 20 bit

implementation of the reference algorithm we achieve ~34% to 50% reduction in area with minimal or no loss in PSNR performance.

Configurations where the reference paper implementation performs better with respect to PSNR performance can be compensated by reducing the degree of approximation in our implementation. For instance, our accurate implementation has higher PSNR with comparable area compared to 16 bit implementation of [23]. Due to the efficient re-usage of components in our algorithm, we can afford a lesser level of approximation to improve PSNR performance.

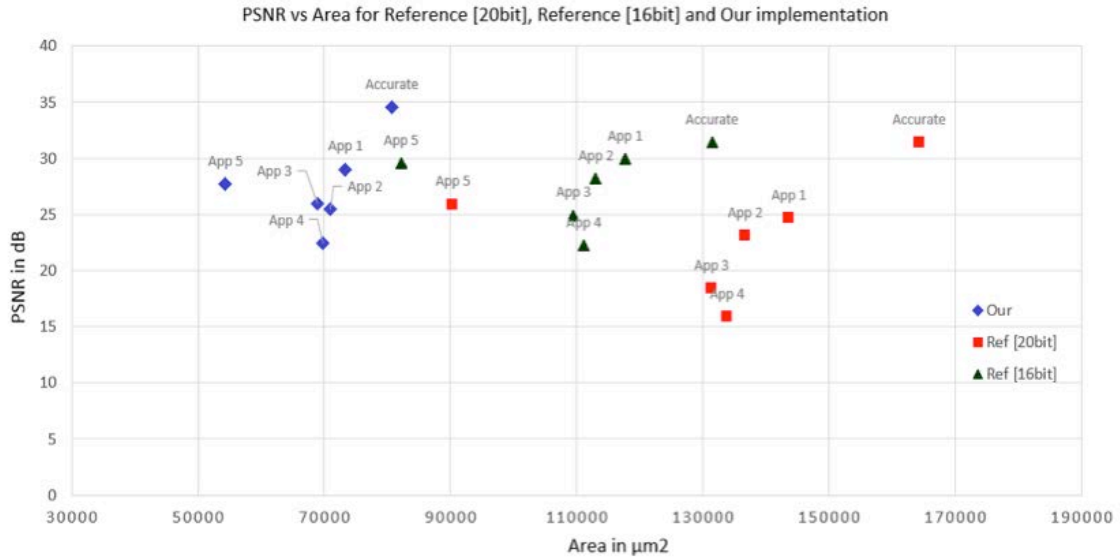


Figure 15. PSNR vs area for various approximations for Ref [20bit], Ref [16bit] and Our implementations

4.1.5. Summary

We can see that compared to [23], we have achieved comparable or slightly worse performance for each approximation as well as the accurate case. We can see that Approximation 5 performs surprisingly well for all cases, with virtually zero overhead.

This can be attributed to its very low mean error value of 0.5. Overall, use of approximate adders and the butterfly structure in Figure 1, helped achieve up to 50% reduction in area compared to [23].

Through this study, we draw the following conclusions:

1. Selective implementation of accurate as well approximate adders can lead to a more optimized solution in terms of both accuracy and area (therefore power).
2. Keeping the early stages of the algorithm accurate provides scope for aggressive approximation in later stages.
3. Significant reduction in area as well as power can be obtained by reusing several computations.
4. Approximation 5 based method has significantly better performance than truncation and should be used in place of truncation.

4.2. Gaussian FIR Filter

For our study, we implement the 5x5 Gaussian Filter for $\sigma = 1$ as shown in Figure 16.

The filter coefficients as well as the inputs are unsigned numbers.

$\frac{1}{256} \times$	1	4	7	4	1
	4	16	26	16	4
	7	26	41	26	7
	4	16	26	16	4
	1	4	7	4	1

Figure 16. Convolution kernel for 5x5 Gaussian FIR Filter

We use an 8x8 multiplier to implement the Gaussian filter. Since the coefficients of the Gaussian filter are hardwired, we use Version A (medium aggression) multiplier to implement the multiplication with coefficients 4 and 7. The Version B (aggressive approximation) multiplier is used to implement the multiplication with coefficients 16, 26 and 41. The internal precision used for both adders and multipliers is 16 bits. All the adders used in the system are accurate; the multipliers are either accurate or approximate based on reference [24] or the proposed method.

4.2.1. Results

The performance of the filter using accurate, reference [24] and the proposed approximate multiplier are compared with respect to PSNR. The ground truth is obtained by running MATLAB simulations in double precision floating point. Table X presents the results for the six standard images – Baboon, Barbara, Boat, House, Lena and Peppers – each of size 512x512.

TABLE X. RESULTS FOR GAUSSIAN FIR FILTER

Image	PSNR (dB)		
	<i>Accurate Multiplier</i>	<i>Reference Multiplier</i>	<i>Proposed Multiplier</i>
Baboon	51.16	49.39	49.33
Barbara	47.16	47.21	47.16
Boat	51.16	48.82	48.75
House	51.15	44.70	44.65
Lena	51.52	47.29	47.24
Peppers	51.16	47.95	47.89
Average	50.55	47.56	47.50

The performance of the reference multiplier [24] and the proposed multiplier is almost the same. This is because our multiplier uses almost the same set of approximations as the reference multiplier [24] for coefficients 4 and 7. There is an average drop of 3.04 dB drop in accuracy compared to the accurate multiplier. While our synthesized implementation shows only a 10% reduction in area due to lack of optimization, a ~40% reduction in area is achieved when we compare the gate count of the accurate and proposed multipliers. This estimate is closer to the area reduction reported in [24].

4.3. Fast Fourier Transform

4.3.1. Split Radix FFT

We implement 2D 32-point FFT by first applying 1D FFT along the rows and then 1D FFT along the columns. The 1D FFT is implemented using the Split Radix FFT (SRFFT) algorithm as described in [28]. The FFT coefficients given by

$$X_k = \sum_{n=0}^{N-1} x_n (W_N)^{nk}, \text{ where } W_N \triangleq \cos \frac{2\pi}{N} - j \sin \frac{2\pi}{N},$$

are decomposed into

$$X_{2k} = \sum_{n=0}^{N/2-1} (x_n + x_{n+(N/2)}) (W_N)^{2nk}$$

$$X_{4k+1} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+(N/2)}) - j(x_{n+(N/4)} - x_{n+(3N/4)})] W_N^n W_N^{4nk}$$

$$X_{4k+3} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+(N/2)}) + j(x_{n+(N/4)} - x_{n+(3N/4)})] W_N^{3n} W_N^{4nk}$$

Thus the N point FFT is implemented by one FFT of length $N/2$ and two FFTs of length $N/4$. The motivation for this decomposition is the fact that the radix-2 algorithm is more efficient for even terms and the radix-4 for odd terms. Because SRFFT combines radix-2 and radix-4 algorithms as one, it uses a slightly different butterfly structure shown in Figure 17.

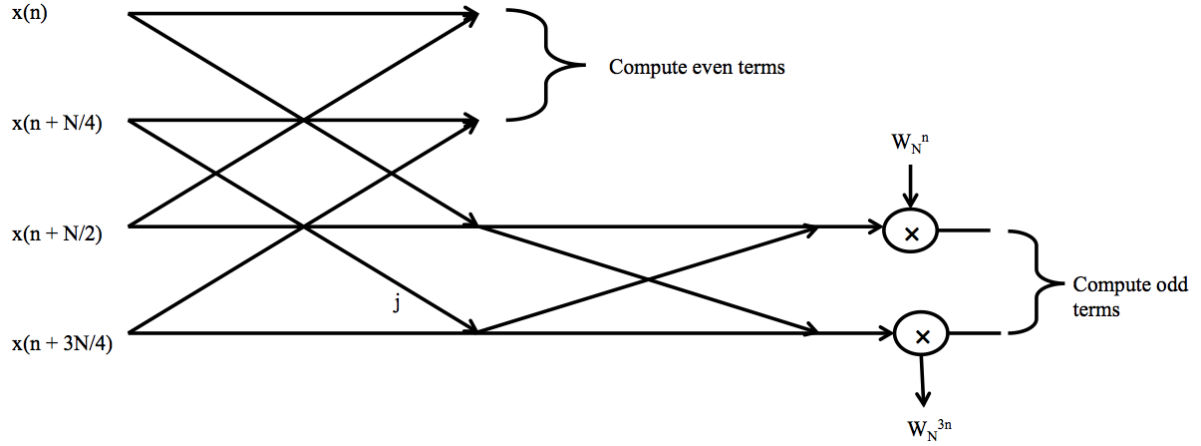


Figure 17. Butterfly structure used for SRFFT [28]

In our implementation, both the inputs and the FFT coefficients are 8 bits wide and the internal precision of the adders and multipliers is 16 bits. All adders used in the implementation are accurate. All the FFT coefficients are scaled by a factor of 128 before multiplication. Since all these coefficients are larger than 16, only Version B of the proposed multiplier is used to implement this algorithm. The output of the first 1D FFT is 12 bits wide and that of second 1D FFT is 16 bits wide.

4.3.2. Results

Table XI provides a comparison of the results obtained for the forward-reverse 2D FFT when implemented using accurate, reference and proposed multipliers. The PSNR is computed by comparing the results obtained from the Verilog HDL implementation with

the ground truth obtained by computing the 2D forward-reverse FFT in MATLAB using floating point precision.

TABLE XI. RESULTS FOR 32 POINT 2D FFT

Image	PSNR (dB)		
	<i>Accurate Multiplier</i>	<i>Reference Multiplier [24]</i>	<i>Proposed Multiplier</i>
Baboon	42.81	37.62	41.46
Barbara	42.71	36.11	41.37
Boat	42.90	38.39	41.84
House	42.29	33.58	40.82
Lena	42.89	36.97	41.59
Peppers	42.79	37.74	41.56
Average	42.73	36.74	41.44

Our implementation has significantly better performance than the one using the reference multiplier in [24] and comparable performance with the accurate multiplier implementation. When two large numbers are multiplied, calculating all the partial products with same level of approximation leads to a steeper drop in accuracy. Varying the degree of approximation within the large multiplier by introducing accurate multiplier to compute the MSB partial product and using aggressive approximation for the LSB partial product leads to better accuracy. This has resulted in our method achieving an average of 4.7dB improvement in PSNR compared to [24] with a very small area overhead.

CHAPTER 5

IMPLEMENTATION OF SAGA ALGORITHM

SAGA is an edge-directed interpolation based image enlargement algorithm that has low complexity and fast execution time compared to state-of-the-art methods [29]. In this chapter, we describe our attempt at designing a hardware architecture for SAGA. In section 5.1, we describe the SAGA algorithm and then in section 5.2 we describe our implementation of pipelined hardware blocks to realize the algorithm. We conclude the chapter by discussing the PSNR performance, and area, latency and power readings of the hardware implementation.

5.1. SAGA Algorithm

SAGA is an interpolation algorithm that linearly interpolates along isophotes rather than along the image lattice. Image isophotes can be described as all points lying on curves of constant luminous intensity. They play an important role in image reconstruction and any errors introduced in these curves can degrade the image quality. SAGA asserts that the tangent lines to the curvature of the isophote provide better approximations on the image grid. It provides a systematic way to calculate these displacements (α and β) along the isophotes. Figure 18 illustrates an isophotic curve and the intersection of the tangent line to the image lattice. The displacements α and β describe the interpolation relationship between the isophote intersections and the image pixel locations. We describe the procedure to compute the displacements and the interpolation process next.

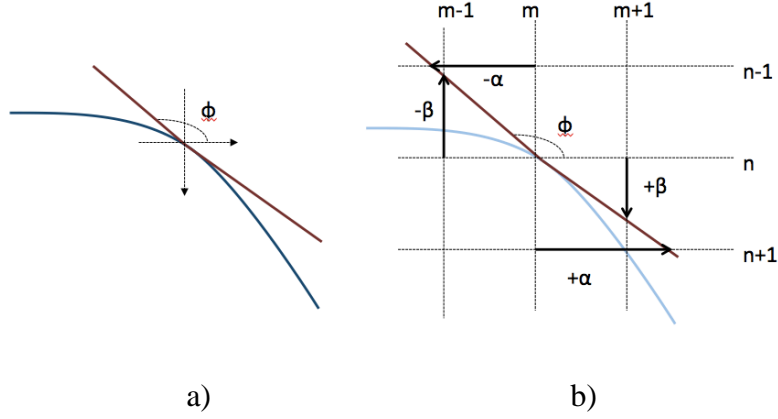


Figure 18. a) The curve shown indicates the isophote curvature. b) The intersection of the tangent line with the integer image. [29]

5.1.1. Determination of Optimal Displacements

The optimal displacements that describe the isophotes for a given row of pixels are determined by minimizing the cumulative squared intensity-matching error, given by $E(\alpha) = \|\text{diag}(I_x) \alpha + I_y\|^2$, where I_x and I_y are the partial derivatives for each pixel in the row obtained using the Sobel operator and $\text{diag}(I_x)$ is a diagonal matrix with I_x as the diagonal entry. This relationship describes the displacement at each node – a node partitions the row (or column) of pixels into smaller segments, where each segment is described by a single isophote. The total number of nodes, also called the stiffness parameter ‘k’, can be fixed or variable. ‘k’ is related to the set of nodes L, as $L \approx M/k$, where M is the number of pixels in a row. The complete set of displacements corresponding to the row can be obtained by interpolating the nodal displacements, $\alpha = \Theta \times \alpha_L$.

Θ is an interpolation band matrix of the form [29]:

$$\Theta = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \theta_1(1) & \theta_2(1) & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \theta_1(k-1) & \theta_2(k-1) & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & \theta_1(1) & \theta_2(1) & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \theta_1(k-1) & \theta_2(k-1) & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \theta_1(1) & \theta_2(1) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \theta_1(k-1) & \theta_2(k-1) \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

θ_1 and θ_2 are linear interpolation functions given by $\theta_1(i) = \frac{k-i}{k}$ and $\theta_2(i) = \frac{i}{k}$, for $0 <$

$i < k$, where k is the stiffness parameter.

5.1.2. Computing Displacements

The combined coefficient matrix is described as $J = \text{diag}(I_x)\Theta$.

This can be simplified as

$$J^T J \alpha_L = J^T (-I_y)$$

The resulting matrix J has $2M-L$ nonzero entries, where M is the number of pixels in a row in the image. The matrix $J^T J$ is a tri-diagonal matrix and solving for α_L requires computing the inverse of this tri-diagonal matrix. Computation of $J^T J$ takes $(2(2M-L)-L)$ multiplications and computing $J^T (-I_y)$ takes $(2M-L)$ multiplications. Therefore, the order

of complexity for computing the displacements for one row of pixels is $O(M)$, where M is the number of pixels in the row. The displacements are calculated along the row as well as along the column in the reference algorithm and therefore the total complexity is $(O(MN))$, where N is the number of pixels in a column.

5.1.3. Constructing Intermediate Images

The displacements can be used to describe ‘matched’ locations. For example, for a pixel location $[m, n]$, the matched locations are $(m \pm \alpha(m, n), n \pm 1)$ and $(m \pm 1, n \pm \beta(m, n))$ as shown in Figure 19. The intermediate pixel values are obtained by interpolating along these vectors. The displacements are usually non-integer values and result in interpolated values that do not fall on the grid. The off-grid displacements are computed from the neighboring, original data using 1D interpolation. New data is then computed along the displacements using linear interpolation. By repeating this procedure along the columns, four intermediate images are obtained. The four images are then combined using a weighting system.

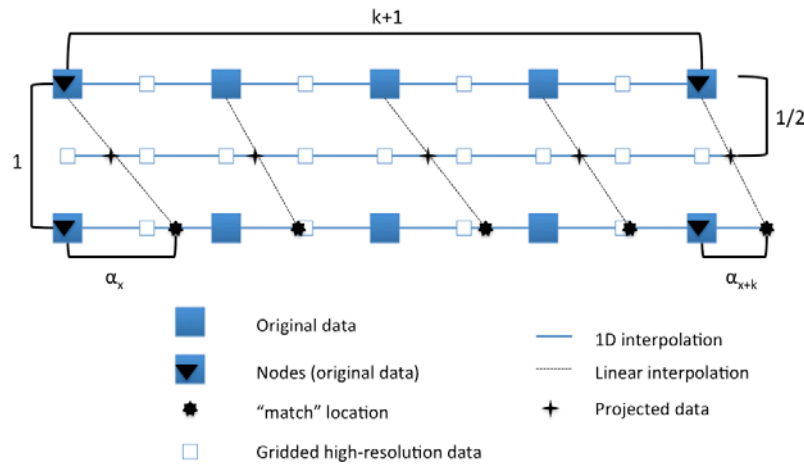


Figure 19. Interpolation of input pixel values along the displacements [29]

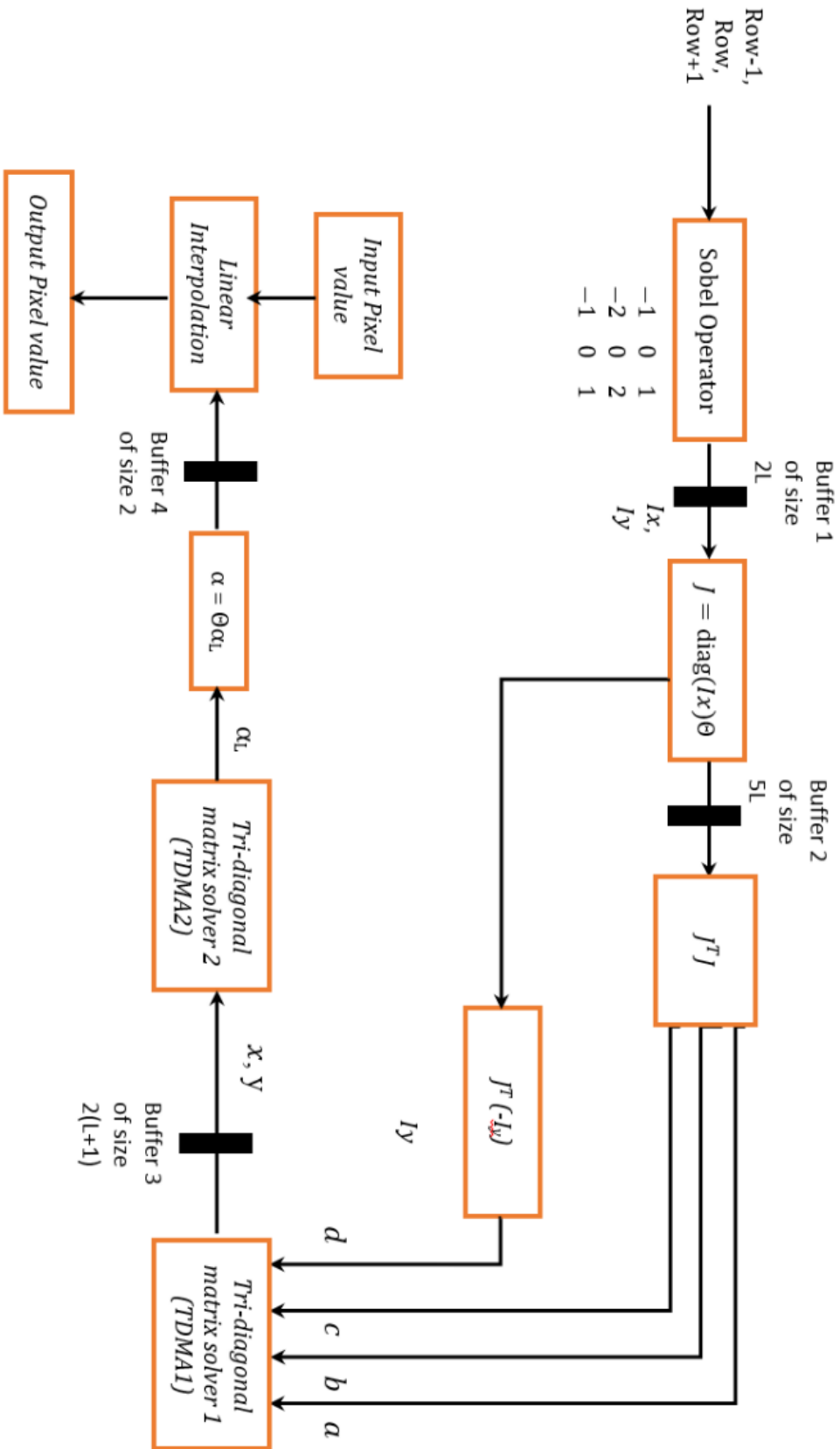
5.2. Our Implementation

Our implementation of the SAGA algorithm is based on the following simplifications:

1. Keep the enlargement factor at 2, which means the algorithm would generate an output image that is twice the size of the input image.
2. Keep the stiffness parameter 'k' constant at a value of 8.
3. Compute the displacements along the rows only and apply uniform weight for both images. This makes the complexity of the algorithm $O(N)$ and reduces the execution time of the algorithm.
4. All 1D interpolations are computed using linear interpolation.

Furthermore we use images of size 64x64 to make the simulation time fast.

Figure 20. Proposed hardware architecture of SAGA



The hardware architecture of SAGA is shown in Figure 20. It is translated into a streaming architecture, where the output of each block is fed to the input of the next block. The flow shown in Figure 20 is for a single row and has to be repeated for every row and column in the image. Next, we describe each hardware block used in the algorithm. All hardware blocks are synthesized using Design Compiler from Synopsys. The area, latency and power estimations are obtained using the Design Compiler for the SAED 90nm Generic Library (optimized for power). The performance of the design is evaluated by comparing the PSNR performance with respect to the original image.

5.2.1. Sobel Operator

The Sobel operator is a 3x3 kernel used for edge detection. The partial derivatives I_x and I_y are obtained by applying the Sobel operator along rows as well as columns. These are implemented using shifts and additions as shown in Figure 21. The input pixel values are scaled down by a factor of 8 to accommodate the high precision requirements of the following stages. We evaluated the performance of the system by running this block for 10, 12 and 16 bits internal precision and achieved PSNR performance of 16.724 dB, 28.85 dB and 28.85 dB, respectively. This analysis was used to fix the internal precision for this module at 12 bits.

We also use Approximation 5 FAs to build the 12bit parallel adder used in this module; the 2 LSBs are added using approximate FA and 10 MSBs are added using accurate FAs. The PSNR performance of the block using approximate parallel adders is 28.041 dB. Since the drop in PSNR performance is negligible, we use the configuration (10+2) to implement the parallel adder, in this block. The hardware block diagram of this module is shown in Figure 21. The synthesis results are presented in Table XII, at the end of this section.

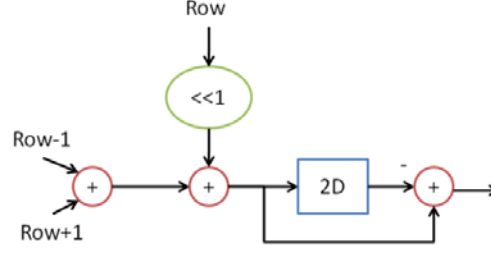


Figure 21. Block diagram for hardware implementation of Sobel operator

5.2.2. Computing the J matrix

The matrix J , also defined as the coefficient matrix, is obtained by interpolating the $\text{diag}(I_x)$ using the interpolation matrix Θ . J is a band diagonal matrix of size $M \times (L+1)$. Since several entries in $\text{diag}(I_x)$ and Θ are 0, it takes only $2M-L$ entries to store the entire J matrix. The choices available for internal precision are 10, 12 and 16 bits. The PSNR obtained for the various configurations are 23.61, 29.79 and 29.79 dB, respectively. We therefore choose 12 bits as our internal precision. This module does not use any parallel adders and therefore, approximate parallel adders are not used in this module.

The stiffness parameter, k , is reported to be 6 to 8 in [29]. Since the interpolation function θ , requires a division by k , we keep k to be a power of 2, namely 8. This enables the interpolation function to be implemented by using a multiplication and right shift operation (to implement division by 8).

Since several entries of matrices $\text{diag}(I_x)$ and Θ are zero, we do not require a complete matrix multiplication. Figure 22 illustrates the computation of matrix J by multiplying only the non-zero entries of $\text{diag}(I_x)$ and Θ . The module takes in k I_x values as input and computes the $2k-1$ interpolated J matrix values. In our implementation, we store these values in a buffer of size $2M-L$.

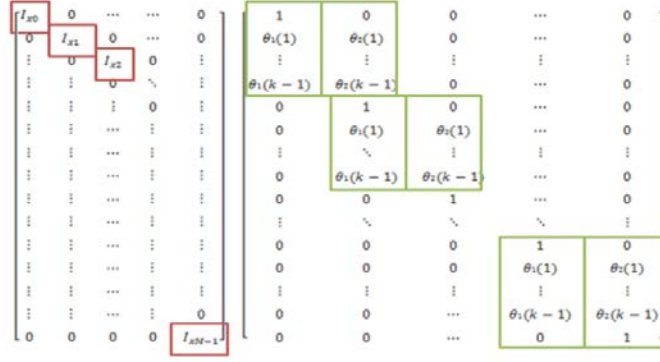


Figure 22. Computation of matrix J from $\text{diag}(I_x)$ and Θ

5.2.3. Computing $J^T J$ and $J^T(-I_y)$

Computations in the $J^T J$ and $J^T(-I_y)$ are implemented using an accurate 8×8 multiplier. The internal precision of the $J^T J$ module is 16 bits and that of $J^T(-I_y)$ module is 20 bits. We arrive at this precision, by comparing the PSNR performance for different values of internal precision for $J^T J$ and $J^T(-I_y)$ modules. Our choice of precision gives a PSNR of 27.97 dB, which is acceptable. By using the approximate parallel adder, the PSNR of the block drops to 23.79 dB, which is very low. We therefore do not use approximate FAs in this module.

$J^T J$ is a tri-diagonal matrix; therefore only $3(L+1)$ elements are required to define this matrix. The three entries of this matrix are represented using a, b and c in Figure 20. The $J^T(-I_y)$ matrix can be defined by using $L+1$ entries. This is the fourth input, d, to the tri-diagonal matrix solver, TDMA1, in the block diagram, shown in Figure 20. Since computation of d is more sensitive, it is kept at a higher precision of 20 bits compared to inputs a, b and c.

5.2.4. Calculating nodal displacements α_L

As seen from equation (4), solving for α_L requires computing the inverse of the tri-diagonal matrix $J^T J$. We use Thomas Tri-diagonal Matrix Inversion algorithm [30] to compute the inverse. The Thomas algorithm is based on LU decomposition where the tri-diagonal matrix is

decomposed into lower and upper triangular matrices. The algorithm is executed in two steps, the first step iterates through the entire matrix downwards (TDMA1) and the next step iterates through the matrix upwards (TDMA2). This means all computations in the first step must be completed before the second step is executed. This is accomplished by introducing a buffer of size $2(L+1)$, between the two stages TDMA1 and TDMA 2 as shown in Figure 20.

The internal precision of both TDMA1 and TDMA2 is 32 bits. We implement the parallel adder by adding the 2 LSBs using approximate FAs and adding the MSBs using accurate FAs. Computation of α_L is integral to the algorithm as α is obtained by interpolating these values. By keeping the accuracy of α_L high at 32 bits, we ensure that the accuracy of the system is maintained. The hardware architecture for TDMA1 and TDMA2 are shown in Figure 23.

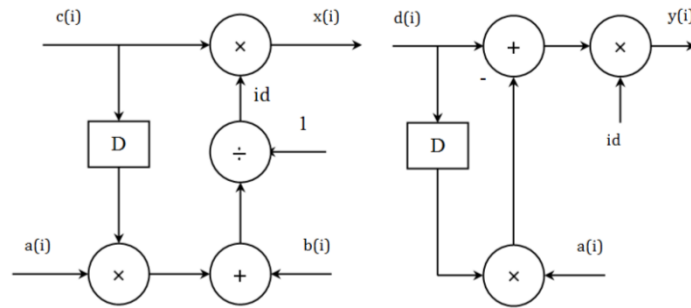


Figure 23. Hardware architecture for TDMA1 and TDMA2

TDMA1 requires a divider circuit to compute the id as shown in Figure 23. We implement the divider circuit using the Newton-Raphson iterative method. To compute N/D , this algorithm computes the reciprocal $1/D$ iteratively and multiplies it by N . Both N and D are 32 bits wide. The Newton-Raphson method requires D to be scaled to a value between 1 and 2. This drops the precision of scaled D to 16 bits. The algorithm begins with an initial approximation of D^{-1} . It then improves the estimate of D^{-1} , X_i , iteratively as

$$X_{i+1} = X_i \times (2 - D \times X_i).$$

The divider circuit is implemented in hardware based on the implementation in [31]. By using multiplexers and load signals, the above logic can be implemented using only one 16×16 multiplier. We use the approximate multiplier described in Chapter 3 to implement this multiplier. The 16 bits least significant partial product is computed using the most aggressive approximation; the middle partial product is using medium approximation and the most significant partial product using accurate multipliers.

D^{-1} corresponds to id in the architectural block diagram presented earlier. After the computation of id , $x(i)$, $y(i)$ and α_L are computed using accurate multipliers. α_L is obtained by solving $\alpha_L(i) = y(i) - x(i) \alpha_L(i+1)$. The first $\alpha_L(i) = y(i)$.

5.2.5. Calculating displacements α

The set of displacements α is calculated by interpolating α_L along the interpolation matrix Θ . We implement this using an accurate multiplier. The precision of this module is 32 bits. All parallel adders used in this module are implemented by adding the 2 LSBs using approximate FAs and adding the MSBs using accurate FAs.

5.2.6. Linear Interpolation to compute output pixel values

The output pixels are obtained by interpolating the original data using 1D interpolation and along the displacements, as shown in Figure 20. The linear interpolation module also uses the same divider module described above.

5.3. Timing Analysis of our hardware architecture

In this section, we describe the timing flow for our hardware implementation. For this analysis, we make certain assumptions regarding the calculation of number of cycles. The number of

cycles for an L bit parallel adder is assumed to take L cycles, an $L \times L$ multiplication takes $4L$ cycles, and all hardwired shifts, comparison and assignment operators take 1 cycle. Table XII provides the computation of the number of clock cycles required for each module. This is calculated from the various operations used to implement each module. Figure 24 illustrates the timing diagram for the pipelined implementation of the algorithm.

TABLE XII. TIMING ANALYSIS OF OUR HARDWARE ARCHITECTURE

Module	# of computations required	# of clock cycles required
Sobel Operator	$k \times (1 \text{ shift} + 5 \text{ 12 bit additions})$	488
J	$k \times 8 \times 8$ multiplications + 16 hardwired shifts	272
$J^T J$	$2k \times 8 \times 8$ multiplications + 15 16bit additions	752
$J^T(-I_y)$	$2k \times 8 \times 8$ multiplications + 15 16 bit additions	752
TDMA1	Computation of denom, inverse and normalization	2072
TDMA2	1 32×32 multiplication, 1 32 bit addition and 1 assignment	161
$\alpha = a_L \times \Theta$	8 32×32 multiplications, 13 shifts and 7 32 bit additions	1261
Interpolate along α	Interpolation using inverse and multiplication	1926
1D interpolation	2 16×16 multiplications, 1 16 bit addition and 1 shift	113

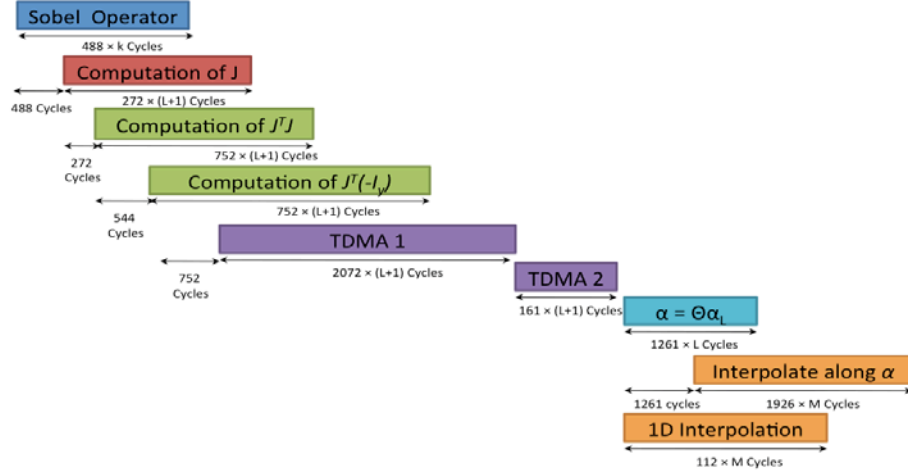


Figure 24. Timing Analysis of our hardware architecture (image not drawn to scale)

We can see that most of the computations are pipelined with the exception of TDMA2, which cannot be started before TDMA1 is completed.

5.4. Hardware Synthesis and Performance Results

We present the synthesis results – area, latency and power, obtained for the various hardware modules in Table XII.

TABLE XIII. SYNTHESIS RESULTS OF ALL THE HARDWARE MODULES

Module	Area – combinational (nm ²)	Latency (ns)	Dynamic Power (mW) for V=1.2V
Sobel Operator	3819.1	2.52	0.18
J	20934.1	4.05	1.29
$J^T J$	89664.3	12.97	7.59
$J^T (-I_y)$	101869.1	10.03	0.96
TDMA1	293305.6	115.28	24.11
TDMA2	55766.0	38.78	6.46
α	34632.8	7.62	3.25
1D Interpolation	33249.5	6.20	3.43
Interpolation along α	147983.3	79.75	11.35

The total time to process one row is $7.76 \mu\text{s}$, when the clock period is 10 ns. The latency to execute the algorithm for the entire image would therefore be $7.76 \mu\text{s}$ multiplied by the number of rows plus the number of columns. For our system, the total latency of the design is $496.48 \mu\text{s}$.

We now present the PSNR results obtained by implementing SAGA algorithm using double precision floating point MATLAB implementation as well as our hardware implementation. We also present the results of our implementation using accurate as well as approximate parallel adders and multipliers. Approximate parallel adders were used in the Sobel operator, TDMA1 and TDMA2 modules; approximate multiplier was used in the divider circuit. The ground truth is the original image of size 127×127 . We can see that our hardware implementation achieves comparable PSNR with respect to the MATLAB implementation for all the images. The use of approximate parallel adders and multipliers leads to a very small drop in PSNR performance for all the images.

TABLE XIV. RESULTS FOR SAGA

Image	PSNR (dB)		
	MATLAB Implementation [29]	Hardware Implementation	
		Using accurate parallel adders	Using approximate parallel adders and multipliers
Baboon	32.21	31.83	31.31
Barbara	32.69	31.91	31.45
Boat	31.57	30.64	30.07
House	30.84	30.36	29.83
Lena	33.02	31.74	31.88
Peppers	32.54	31.52	30.59
Average	32.14	31.33	30.86

CHAPTER 6

CONCLUSION

This thesis studied the use of approximate adders and multipliers in reducing the area and latency of image processing kernels with only a small loss in performance. We also develop a hardware architecture for the state-of-the-art image enlargement algorithm, Segment Adaptive Gradient Angle (SAGA), and implement it using pipelined hardware blocks.

6.1. Summary

We use several approximate versions of the mirror adder presented in [23] to implement approximate parallel adders. A combination of accurate and approximate parallel adders was used to implement 2D Discrete Cosine Transform algorithm and our implementation was compared against several configurations of the reference algorithm [23]. Our data-path was 16 bits wide and the 16 bit approximate parallel adders were implemented using approximate FAs to add 6 LSBs and accurate FAs to add 10 LSBs. We synthesized the competing implementations using Synopsys in 90nm technology and computed the PSNR with respect to the double precision floating point MATLAB implementation. We tested the implementations for test images Barbara, Baboon, Boat, House, Lena and Peppers, of size 512x512. By using a mixture of accurate and approximate adders, we were able to achieve a reduction of ~34% to 38% in area with comparable PSNR performance compared to the implementation presented in [23].

We also compared the performance of the 2D DCT system built with parallel adders against truncation. We used Approximation 5 FA to build the approximate parallel adder since it uses the same number of logic gates as a truncated adder. We show that using approximate parallel adders is far more effective than brute force truncation. We achieve negligible reduction in PSNR performance for an implementation where 4 LSBs were added using Approximation 5

adders. In contrast, 4 bit truncation of the 16 bit parallel adder results in more than 10 dB drop in PSNR performance and is clearly not acceptable.

Next we propose several enhancements to the approximate multiplier proposed in [24]. The reference multiplier is built with 2x2 multiplier building blocks that produce 3 bit outputs (instead of 4 bits). We propose judiciously replacing some of the approximate 2x2 multiplier blocks with accurate 2x2 multipliers. This results in significant improvement in accuracy of the proposed multiplier for the case when both inputs are large.

We study the effect on the PSNR performance of image processing kernels such as Gaussian FIR Filter and Fast Fourier Transform when implemented using approximate multipliers. We calculate the PSNR by comparing the results obtained by hardware implementation of the algorithm with MATLAB implementation in double precision floating point. For the Gaussian FIR Filter, we show that use of proposed 8x8 multiplier results in an average drop of 3 dB PSNR performance compared to the accurate baseline case, with an estimated 33% reduction in area, in terms of number of logical gates required for the implementation. For the 2D 32x32 point Fast Fourier Transform, we show that use of proposed 8x8 multiplier results in reduction of 1.3 dB in PSNR performance compared to the accurate case and 4.7 dB improvement in PSNR performance compared to the implementation using [24].

We also describe our hardware implementation of the state-of-the-art image enlargement algorithm Segment Adaptive Gradient Angle (SAGA). We map this algorithm into pipelined hardware blocks and present synthesized results of each block. We perform timing analysis of each hardware block and show that each row can be completed in 496.48 μ s when the system is clocked at 100 MHz. We achieve an average PSNR performance of 31.33 dB using accurate parallel adders and multipliers, compared to the real 127x127 image. The average PSNR

performance of the system using approximate parallel adders and multipliers is 30.86 dB. We can see that the drop in performance is negligible for the approximate case. The performance of both accurate as well as approximate implementation is very close to the PSNR of 32.14 dB that is obtained by the floating point double precision MATLAB implementation of SAGA.

6.2. Future Work

The approximate adders and multipliers reduce area and latency. The reduced latency can be translated into increased throughput or reduced power consumption through voltage scaling. Thus approximate circuits can be used to reduce power with minimal loss in performance.

Our current implementation of the SAGA algorithm includes only the data-path. The next step would be to implement the buffers between the hardware blocks and build the memory interfaces for a more complete design.

SAGA also provides additional opportunities to apply the approximate circuits. While we used approximate circuits to implement some of the hardware blocks, a more thorough investigation on application of even more aggressive approximations and compensating them using correction terms, can be done.

REFERENCES

- [1] C. Chen and et al., "Analysis and architecture design for variable block-size motion estimation for H.264/AVC," IEEE Trans. on Circuit and System-I, vol. 53, no. 2, pp. 578-593, Feb. 2006.
- [2] A. Sinha, A. Wang and A. Chandrasekaran, "Energy scalable system design," IEEE Trans. on VLSI Systems, vol 10, pp. 135-145, April 2002.
- [3] K. J. Lin, S. Natarajan and J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems," Real-Time System Symposium, pp. 210-217, Dec 1987
- [4] S. H. Nawab, A. V. Oppenheim, A. Chandrakasan, J. M Winograd, and J. T. Ludwig, "Approximate signal processing," Journal of VLSI Signal Processing, vol. 15, pp. 177-200, 1997
- [5] Y. Andreopoulos and M. van der Schaar, "Incremental refinement of computation for the discrete wavelet transform," IEEE Trans. on Signal Processing, vol. 56, pp. 140-157, Jan. 2008
- [6] Y. Emre and C. Chakrabarti, "Low energy motion estimation via selective approximation," IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors, pp. 176-183, Sept. 2011.
- [7] J. Y. F. Tong, D. Nagle and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating point arithmetic," IEEE Trans. on VLSI Systems, vol. 8, pp. 273-286 June 2000.
- [8] Z. He, C. Tsui, K. Chan, and M. L. Liou, "Low-power VLSI design for motion estimation using adaptive pixel truncation," IEEE Trans. on Circuits and Systems for Video Technology, vol. 10, no 5, pp. 669-678, August 2000.
- [9] S. H. Kim, S. Mukohopadhyay and M. Wolf, "System level energy optimization for error tolerant image compression," IEEE Embedded System Letters, vol. 2, pp. 81-84, Sept. 2010.
- [10] J. Park, J. H. Choi and K. Roy, "Dynamic bit-width adaptation in DCT: an approach to trade-off image quality and computation energy," IEEE Trans. on VLSI Systems, vol. 18, pp. 787-793, May 2010.
- [11] A. Chandrasekaran and R. Brodersen, "Minimizing power consumption in digital CMOS circuits," Proc. of the IEEE, pp. 498-523, April 1995.
- [12] B. Shim, S. R. Sridhara and N. R. Shanbag, "Reliable low-power digital signal processing via reduced precision redundancy," IEEE Trans. on VLSI Systems, vol. 12, pp. 497-510, May 2004.

- [13] R. Hegde and N. R. Shanbhag, "Soft digital signal processing," *IEEE Transactions on VLSI Systems*, vol. 9, no. 12, pp. 813-823, Dec 2001.
- [14] N. R. Shanbhag, R. A. Abdallah, R. Kumar and D. L. Jones, "Stochastic computation," *Design Automation Conference*, pp. 1589-1592, May 2011.
- [15] J. Sartori and R. Kumar, "Architecting processors to allow voltage/reliability tradeoffs," *International Conference on Compiler Architectures and Synthesis for Embedded Systems*, pp. 115-124, Oct. 2011.
- [16] I. J. Chang, D. Mohapatra and K. Roy, "A voltage-scalable & process variation resilient hybrid SRAM architecture for MPEG-4 video processors," *Design and Automation Conference*, pp. 670-675, 2009.
- [17] M. Cho, J. Schlessman, W. Wolf and S. Mukhopadhyay, "Accuracy-aware SRAM: a reconfigurable low power SRAM architecture for MPEG-4 video processors," *Design and Automation Conference*, pp. 823-828, 2009.
- [18] B. Shim, S. R. Shridhara and N. R. Shanbhag, "Reliable Low-Power Digital Signal Processing via Reduced Precision Redundancy, " *IEEE Transactions On Vlsi Systems*, vol. 12, no. 5, May 2004.
- [19] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," in *Proc. Design, Automat. Test Eur.*, 2008, pp. 1250-1255.
- [20] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *Proc. Design, Automat. Test Eur.*, 2010, pp. 957-960.
- [21] 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 Trans. Circuits Syst. Part I*, vol. 57, no. 4, pp. 850-862, Apr. 2010.
- [22] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67-73, Mar. 2004.
- [23] V. Gupta, D. Mohapatra, A. Raghunathan, K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, January 2013.
- [24] P. Kulkarni, P. Gupta, and M. D. Ercegovac, "Trading accuracy for power in a multiplier architecture," *J. Low Power Electron.*, vol. 7, no. 4, pp. 490-501, 2011.
- [25] Venkataramani, S. ; Sabne, A. ; Kozhikkottu, V. ; Roy, K. ; Raghunathan, A., "SALSA: Systematic Logic Synthesis of Approximate Circuits, *Design Automation Conference (DAC)*," 2012 49th ACM/EDAC/IEEE

- [26] R. Venkatesan, A. Agarwal, K. Roy and A. Raghunathan, "MACACO: Modeling and Analysis of Circuits for Approximate Computing," Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference
- [27] Chan, Wei-Ting J. ; Kahng, Andrew B. ; Kang, Seokhyeong ;Kumar, Rakesh ; Sartori, John, "Statistical Analysis and Modeling for Error Composition in Approximate Computation Circuits,"
- [28] P. Duhamel, "Implementation of Split-Radix FFT Algorithms for Complex, Real and Real-Symmetric Data," IEEE Trans. on Accountics, Speec, and Signal Proc., vol. ASSP-34, no. 2, April 1986.
- [29] C. M. Zwart, and D. H. Frakes, "Segment Adaptive Gradient Angle Interpolation," IEEE Transactions On Image Processing, Vol. 22, No. 8, August 2013
- [30] Tridiagonal Matrix Inversion algorithm: http://www3.ul.ie/wlee/ms6021_thomas.pdf
- [31] Newton Raphson method for Reciprocal Approximation: Digital Computer Arithmetic Data-path Design Using Verilog HD by James E. Stine
- [32] Y. Emre and C. Chakrabarti, "Quality-Aware Techniques for Reducing Power of JPEG Codecs," J Sign Process Syst (2012) 69:227–237
- [33] University of Southern California – Signal and Image Processing Institute Image Database: <http://sipi.usc.edu/database/>