A Taxonomy for Approximate Computing Applied to the Sobel Filter

Jørgen Petlund

Thesis submitted for the degree of
Master in Cybernetics and Autonomous Systems
60 credits

Department of Technology Systems
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

A Taxonomy for Approximate Computing Applied to the Sobel Filter

Jørgen Petlund

© 2022 Jørgen Petlund

A Taxonomy for Approximate Computing Applied to the Sobel Filter

http://www.duo.uio.no/

Printed: X-press printing house

Abstract

The resources in a system should be utilized efficiently to obtain relevant information. Approximate computing allows trading a reduction in accuracy for efficiency gains. However, approximate computing is generally concerned with the accuracy and optimization of a single layer, while a system spans several layers. A system approach to approximate computing is needed. A taxonomy is presented that adheres to this principle. The taxonomy classifies approximate computing based on the fundamental components of computing. The techniques are categorized by four classes: precision, memory, iterative and error.

A system for detecting objects using the Sobel filter is used to evaluate the taxonomy. The Sobel filter is made approximate by 49 approaches, using both techniques that operate alone or in unison. The simulations are conducted using high level synthesis tools.

The results show that a local reduction in accuracy does not necessarily propagate through the system, and that less accurate intermediary functionality is advantageous for certain input characteristics. Meaning that significant resource savings can be obtained without reducing the overall system performance.

Contents

1 Introduction **1**

1.1 Aim of thesis . . . 5

1.2 Outline of thesis . . . 6

2 Background **7**

2.1 Information . . . 7

2.2 Sobel filter . . . 9

2.3 Approximate computing . . . 11

2.3.1 Precision scaling . . . 11

2.3.2 CORDIC algorithm . . . 12

2.3.3 Loop perforation . . . 13

3 Methods and implementation **14**

3.1 A novel taxonomy for approximate computing techniques . 14

3.2 Model Description . . . 17

3.2.1 Object detection using contour extraction . . . 17

3.2.2 Sobel filter . . . 23

3.2.3 Hardware acceleration of the Sobel filter . . . 23

3.2.4 Precision . . . 28

3.2.5 Memory . . . 28

3.2.6 Iterative . . . 29

3.2.7 Error . . . 30

3.2.8 Combined approaches . . . 32

4 Experiments and results **34**

4.1 Evaluation metrics and detection cases . . . 34

4.1.1 Case 1 – No detection objects . . . 37

4.1.2 Case 2 – High background noise . . . 37

4.1.3 Case 3 – Low contrast . . . 38

4.1.4 Case 4 – Multiple objects . . . 38

4.1.5 Case 5 – Large object 38
4.1.6 Case 6 – Large object and foreground noise 40
4.2 Reference System 40
4.3 Single technique approaches 42
4.3.1 Precision scaling techniques 42
4.3.2 Iterative techniques 48
4.3.3 Memory techniques 49
4.3.4 Error techniques 50
4.4 Combined techniques 54
4.4.1 Precision scaling and iterative techniques 54
4.4.2 Performance characteristics of combined techniques . 56
4.4.3 Unaccounted resource savings 60
4.4.4 Observations on generality 60
4.5 Correlation 61
4.6 Comparison 64

5 Discussion **68**

6 Conclusion **73**
6.1 Further work 73

Bibliography **75**

Preface

I would like to thank my supervisors Dr. Jonas Moen from the Dep. of Technology Systems and Dr. Alexander Wold from the Dep. of Informatics at UiO. The input they provided and guidance given through writing this thesis has proven invaluable.

The process has been a rigorous task of asking the right questions and interpreting the results. In the end I stand with as many questions as answers.

Chapter 1

Introduction

There is, in any computational system, a finite amount of resources available. These resources should be utilized efficiently to obtain relevant information from data at lowest cost. In order to acquire relevant information, it may not be beneficial to process raw data at full capacity. Techniques to extract information at a lower cost are needed. Hence, the introduction of Approximate Computing (AC).

AC is generally defined as a paradigm that trade efficiency gains for a reduction in output accuracy[1]. Where efficiency primarily refer to either reduced resource usage, increased computation speed or lower energy consumption. Output accuracy can refer to a plethora of metrics, such as: PSNR (peak signal-to-noise ratio), pixel correctness, relative difference, clustering accuracy and mean centriod distance, correct/incorrect decision ratio, and others [2]. The relevant information is the set of possible messages the system is designed to output [3]. Information can not be relevant without context, and it is the context that determines the objective of a system, i.e. the input and output relation. For example when the context is to determine if a picture depicts a person or not, then the relevant information the system communicates out is a simple yes/no, the messages can be represented by a single bit. If it is also desired to know the number of people in the picture then the relevant information requires more bits to represent all possible messages. The accuracy is concerned with whether or not the messages the system outputs are correct for the input. If the system always communicates 'yes' when there is a person depicted in the input image, and correspondingly 'no' when there is not, then the output is fully accurate. If the system only occasionally communicates messages that are correct for the input, then the system is less accurate. Output accuracy is thus, an estimate to quantify the correctness of the messages a system

outputs.

The functionality within a system can be divided into sub-systems. And, the sub-systems can be divided into sub-sub-systems and so on. On its own each sub-system is given some input and produces some output with a given accuracy. If these sub-system are set to operate with lower accuracy, meaning they produce less accurate intermediary results, and by doing so require less resources – while the system communicates the same relevant information – then, the system at large has not become approximate, even though the sub-systems have. Until the accuracy of the overall system is affected and loss of information occurs, the intermediary accuracy is partly redundant. And, if there is redundancy then the resources are not used efficiently. Consequently, it is not the objective to reduce the accuracy, rather it is the opposite. To retain as much accuracy as possible while increasing the efficiency. The catch being that after an optimal encoding is reached the only way to further increase the efficiency is to reduce the accuracy. Thus, a succinct description of AC is:

> *Techniques that increase the efficiency of a computation at the cost of reduced accuracy.*

It is the crucial initial step in every AC technique to uncover approximable variables and operations [2]. Introducing AC to a system is a matter of examining the sub-functionality and in turn incorporate suitable AC techniques. However, AC has not been explored to a great extent in the context of large systems. The focus within AC is on the contained approximate functionality, seen out of context from a larger system. In [4] Agrawal, Choi, Gopalakrishnan *et al.* state that "AC is primarily concerned with optimizing a single layer in the stack" [p. 1]. They propose that combining several techniques spawning multiple levels makes AC viable for a broader range of systems then what it is currently used for.

In [1] a classification based on domain is presented. The categories being: approximate instruction processing, approximate communication, approximate cloud computing and approximate hardware systems. The intent of the classification is to provide an "easy reference of current research trends"[1, p. 622]. Three classifications are presented in [2]. The first is based on implementation approach, the second categorizes by research field, and the third is based on the type of processing unit/component and memory technology used, and optimization objective. Based on the abstraction layer, AC can also be categorized into architecture, software or hardware techniques [5][6][7]. In [8] Moreau, Miguel, Wyse *et*

al. also propose a taxonomy of general purpose AC techniques. The axis they use for classification being visibility, determinism, and coarseness. The terms meaning, respectively: correctability of the approximation effects, reproducibility of the approximate results, and control over the efficiency–accuracy tradeoffs. However, it is not given that the effect of the intermediary accuracy on the overall system performance is known beforehand. As the system functionality unfolds incrementally into new subsystems it is desired to detect the different types of approximable regions and to distinguish what techniques are applicable. Clearly, none of the classifications today adhere to this specific need.

With the focus shifted away from the small contained components and onto integrating AC for larger, more complex systems, a new classification is needed. That instead of looking at approximate computing techniques first, then determining their capabilities, and lastly, propose suitable applications for the technique, does the opposite. Start with the overall system, determine the internal capabilities, and then chose the applicable AC techniques. A classification that aids in uncovering approximable variables and operations, and connects these approximable regions with suitable techniques.

The Sobel filter, often referred to as the Sobel operator, is a convolution kernel used in image processing for edge detection. Within the field of AC it is used primarily as a benchmark for either approximate multipliers, as in the WOAxC framework presented in [9] by Ma, Thapa, Wang *et al.*, or for approximate adders [10] [11]. In [12] Ndour, Jost, Molnos *et al.* benchmark using the Sobel filter for a variable bit-width approximation technique based on load size configuration. The Sobel filter is also used to benchmark in software based AC approaches, such as for the loop perforation (skipping iterations in a loop) and shift techniques presented in [13] by Aponte-Moreno, Pedraza and Restrepo-Calle. The Sobel filter is a suitable candidate to test such a classification on, due to its extensive usage within AC. Additionally, both low level techniques (approximate multipliers/adders) and high level techniques (loop perforation) have been used on the filter. It already exhibits the behavior needed to implement AC techniques spanning multiple levels, though the techniques have not been tested in unison.

An architecture well suited for AC is an FPGA. It allows access to the hardware, which means it is possible to fully define the precision at each

computational step and decide the arithmetic used. Furthermore, FPGAs are well suited for image processing tasks, which is used quite extensively to benchmark AC [1][2][9].

As previously mentioned, AC is applied at the hardware level or the software level, or architecture level, and include techniques that use approximate adders, multiplexers or logic circuits, imprecise/faulty hardware, voltage scaling, precision scaling, memoization, task skipping, approximate compilers and several more [1], [2].

AC techniques at the hardware level are usually made for ASIC-based systems[14]. This includes, among others, the use of approximate adders [15]–[17], multipliers [18], [19] or voltage scaling [20]. However, there is a shift towards more FPGA oriented designs. In [14] Prabakaran, Rehman, Hanif *et al.* propose a design methodology suited for building approximate adders for FPGAs. There is also FPGA specific multipliers [21], [22], and frameworks [23], [24]. Another common implementation on FPGAs are trigonometric approximation functions [25]–[27].

Reducing the precision is a common way to design AC techniques [28]. In [29] Roldao-Lopes, Shahzad, Constantinides *et al.* present a mantissa bit-width scaling technique. It changes the working precision of the mantissa and the number of conjugate gradient iterations to achieve performance gains and is implemented on FPGA. Whereas, Lee, Gaffar, Cheung *et al.* [30] use a fixed point precision optimization technique. The method is based on a static scaling approach for fixed-point arithmetic and an FPGA is used to carry out and test the design.

Both Lee and Gerstlauer [28], and Nguyen, Menard and Sentieys [31] present methods for dynamic precision scaling. In [28] Lee and Gerstlauer use the statistical analysis of noise to create a method for dynamic precision scaling. It reduces design time since it calculates the optimal word length at runtime, instead of relying on simulations. In [31] Nguyen, Menard and Sentieys use a fixed-point specification that dynamically changes based on the *signal to noise ratio* (SNR).

In [32] Tian, Zhang, Wang *et al.* propose a technique for dynamic precision scaling using memory access. Precision scaling usually target on-chip computations, however, it could just as well be done for the off-chip memory. A memory access controller is presented for dynamic precision scaling.

1.1 Aim of thesis

The aim of this thesis is to propose a taxonomy that cater to the idea of integrating AC into larger systems. With the presupposed notion that any large system is to be divided into smaller components applicable for AC. Since the classification aims to put the system first, and AC techniques second, it must hold true for any computation system to be useful.

An important aspect of this thesis is to evaluate the classification. And, to this extent explore how the classes of the taxonomy interact, i.e. to explore if they hold distinct properties and how the properties relate. To test the classification the Sobel filter is chosen as the benchmark system. For this purpose a context is presented, with the objective to detect aerial vehicles (AV). The system is designed with two principal sub-systems, the Sobel filter and the detection algorithm. The edges produced by the Sobel filter is used as the input to the detection algorithm. The functionality within the Sobel filter will be made available for AC. And, the intermediary output accuracy of the filter will be compared to the overall system accuracy.

To allow for a wide range of applicable AC techniques the filter is hardware accelerated with an FPGA. The hardware acceleration is performed using modern high level synthesis (HLS) tools, and the system is simulated for six detection cases with varying environments.

Research Question I *When does a computation system become approximate?*

There is no guarantee that the intermediary accuracy reduction introduced by the AC techniques propagate through the system and impact the overall system performance. Also, for an image processing task there is a possibility that certain approximations aid the system in reducing noise or favorably differentiate pixel intensity.

Research Question II *Does an approximate computing taxonomy exist that adheres to a system approach?*

The desired taxonomy is one that: makes it easier uncover approximable regions, and that aids in selecting suitable AC techniques. For a system approach it is the objective of the system and the overall performance that holds importance. The functionality of the internal components are secondary.

1.2 Outline of thesis

In the next chapter the relevant theoretical background about information theory, approximate computing and the Sobel filter is presented. Chapter 3 presents the taxonomy, and goes through the simulator setup and explains the implementation details. In chapter 4, the experiments and results are presented along with an analysis of the results. A discussion about the results obtained from the experiments is conducted in chapter 5. The validity of the simulator and the research questions are discussed as well. A conclusion is presented in chapter 6 along with suggestions for further work.

Chapter 2

Background

This chapter explains the theory and concepts used in this thesis. A section on information is presented first. Then, the theory behind the Sobel filter is presented. Lastly, a general outline of AC and relevant techniques are described.

2.1 Information

Information is a term that bear many meanings. However, when it comes to computing it is the definition presented in what is called Information Theory that holds relevance. To be more precise, it is the model presented by C. E. Shannon in *A mathematical theory of communication*[3]. Wherein he states:

> The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point. Frequently the messages have *meaning*; that is they refer to or are correlated according to some system with certain physical or conceptual entities. These semantic aspects of communication are irrelevant to the engineering problem. The significant aspect is that the actual message is one *selected from a set* of possible messages. The system must be designed to operate for each possible selection, not just the one which will actually be chosen since this is unknown at the time of design. [3, p. 379]

Information is thus defined by probability, or as Weaver explains it as ”a measure of one’s freedom of choice when one selects a message” [33, p. 4]. The quantity of information produced can be expressed by the uncertainty,

or entropy, of the outcome. This representation of information is devoid of meaning. A message that contains a lot of meaning or one that is a random sequence of gibberish may be equal in terms of the information they hold. It is the characteristics of the information source that dictate the amount of information. The formula that Shannon reached (See Equation 2.1) was one that "summed the probabilities with a logarithmic weighting" [34, p. 1]. Where H is a measure of the entropy and p_i is the probability of each outcome.

$$H = -\sum p_i \log p_i \tag{2.1}$$

With a logarithmic base 2 this quantity can be represented by binary digits, also known as bits. A binary digit has two states and can be thought of as representing a yes or no question. With the probability of each possible message in mind, how can one construct a scheme such that, on average, the number of yes/no questions asked is minimized? This is the essential question when it comes to coding the messages.

As it turns out, the optimal coding is reached when the average bits needed per symbol is equal to that of the entropy H. The goal being to obtain the least bits per symbol. Consider the example provided by Shannon where there is four symbols A, B, C, D with probabilities $\frac{1}{2}$,$\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{8}$, each successive choice being independent. The entropy is $\frac{7}{4}$ bits per symbol given by:

$$H = -(\frac{1}{2}\log\frac{1}{2} + \frac{1}{4}\log\frac{1}{4} + 2 \times \frac{1}{8}\log\frac{1}{8}) = \frac{7}{4} \tag{2.2}$$

Intuitively one may see that these four symbols conform well to a fixed two bit representation. Presumably with A, B, C, D represented by the binary numbers 00, 01, 10, 11, respectively. However, this encoding does not reach the limiting value of $\frac{7}{4}$ bits per symbol on average. The less intuitive yet better approach is to use the following code:

A	0
B	10
C	110
D	111

For a sufficiently long sequence N the average number of bits used to encode each symbol will be:

$$N[1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 3(\frac{1}{8} + \frac{1}{8})] = \frac{7}{4}N \tag{2.3}$$

This is same as the entropy of the source. Therefore an optimal encoding. More often than not an ideal situation like this is impossible to attain directly. Nevertheless, any source can be coded ideally. The way to do

this is by using *block coding*. The idea is to consider groups of symbols and assign codewords to these groups [35]. The probabilities associated with different groups as well as the increased flexibility in codeword lengths makes it possible to create an ideal coding scheme, given sufficiently large blocks. There are, unfortunately, some practical limitations when it comes to using large block sizes. One must consider the complexity of operations needed by the encoder and the decoder.

In the model presented by Shannon information is always context based and located within the realm of communication. Take a sequence of symbols out of the system and it conveys no information. It is just a placeholder. Still, what if the ensemble of possible messages is unknown? What is say, the "information in a book" [36, p. 10], and should it be viewed as "an element in the set of all possible books"? Clearly, a measure of information that is independent of the context is needed as well. This theory of information is called Kolmogorov complexity and describes the information in individual objects by themselves. The idea being that the entropy is the minimum number of bits needed to reconstruct a particular object [37]. These two notions of information are not contradictory, in fact they supplement each other.

2.2 Sobel filter

The Sobel operator is a well defined edge detection algorithm [9]. First concieved by Irwin Sobel in 1968, at the time under the name *An Isotropic 3x3 Image Gradient Operator* [38]. The purpose of the operator was to "define the magnitude of the directional derivative estimate" [38, p. 1] for a point in a cartesian grid. For a 3x3 image the central gradient is the sum of the 8 directional derivative vectors. Where the magnitude of the directional derivate vector for a given neighbour is defined in equation 2.4.

$$|g| = < density\ difference > / < distance\ to\ neighbor > \tag{2.4}$$

However, the neighbours group into antipodal pairs, as seen from equation 2.5.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} a & & \\ & & \\ & & i \end{bmatrix} \begin{bmatrix} & & c \\ & & \\ g & & \end{bmatrix} \begin{bmatrix} & b & \\ & & \\ & h & \end{bmatrix} \begin{bmatrix} & & \\ d & & f \\ & & \end{bmatrix} \tag{2.5}$$

And, the vector summing within each pair causes all the center values to

cancel eachother out. The resulting vector sum is seen from equation 2.6.

$$
\begin{aligned}
G = \quad & (c-g)/4 \cdot [1,1] \\
& +(a-i)/4 \cdot [-1,1] \\
& +(b-h)/2 \cdot [0,1] \\
& +(f-d)/2 \cdot [1,0]
\end{aligned}
\tag{2.6}
$$

In hardware a divide by 4 is the same as a double right shift. A shift is done in the routing and . Unfortounatly, this procedure loses low order bits. Instead of dividing, it can be convenient to scale the vector by 4, which translates to a double left shift, with no bits lost in the process. The function can then be expressed as the weighting functions for the x and y components found in equation 2.7.

$$
Dx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad Dy = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.7}
$$

In image processing these two 3x3 kernels are convolved with an image to compute the derivatives for the vertical and horizontal changes[39]. The gradient magnitude is then found using equation 2.8.

$$
G = \sqrt{D_x^2 + D_y^2} \tag{2.8}
$$

It is also possible to compute the gradient's direction using the computationally expensive atan2 function using equation 2.9.

$$
\Theta = atan2(G_y, G_x) \tag{2.9}
$$

An example of the Sobel filter is shown in Figure 2.1. Before the derivatives are computed the image is converted to grayscale. Clear lines are found around the outline of the person and at the brink of the horizon, and indicate a sudden change in intensity.

Figure 2.1: The image to the right show the gradient magnitude after the Sobel filter has been applied. The image to the left show the original image before the transformation. (Nasjonalmuseet [40])

2.3 Approximate computing

AC is an umbrella for techniques that deal with inexact computing. Conventionally the field deals with approximate hardware, and specifically circuits[14]. However, the focus has shifted on to FPGAs as well [41]. Approximate software has also become a versatile field, and include approximate algorithms, frameworks and compilers[1].

The need for AC stem from several factors. One is that AC is inherent for any system that samples from a continuous domain into a discrete domain [2]. Another key aspect is that many applications are error resilient [1]. Working with full precision is redundant. Optimization of a system may also outweigh the reduction in accuracy. To sum it up, Mittal states it eloquently:

> For many complex problems, an exact solution may not be known, while an inexact solution may be efficient and sufficient. [2, p. 4]

2.3.1 Precision scaling

A common way to make a system approximate is to use precision scaling, thereby truncating the least significant bits (LSB) [29]. The LSB have a smaller significance than the higher order bits. Removing these low order bits may only have a miner impact on the obtained accuracy [2].

In [30] Lee, Gaffar, Cheung *et al.* use a fixed point precision optimization technique that achieve a reduction in the area and the latency by up to 26% and 12%, respectively. The method is a static precision scaling approach for fixed-point arithmetic called *MiniBit*. The fractional components of

the fixed point number are truncated using static analysis based on affine arithmetic. An FPGA is used to carry out and test the design.

2.3.2 CORDIC algorithm

The **CO**ordinate **R**otation **DI**gital **C**ompute (CORDIC) algorithm is an iterative approach that typically increase precision by one bit per iteration. The original implementation was presented by Volder in 1959 [42]. It is an "iterative method of performing vector rotations by arbitrary angles using only shifts and adds" [25]. Since all trigonometric functions can be derived from functions using vector rotation, it proves very useful. The CORDIC algorithm is used to compute the gradients direction for the Sobel filter given in equation 2.9.

Starting from the general rotation transform the CORDIC algorithm can be derived:

$$x' = x\cos\theta - y\sin\theta \tag{2.10}$$

$$y' = y\cos\theta + x\sin\theta \tag{2.11}$$

A rewrite of the equations can be done using the trigonometric property of $\sin\theta / \cos\theta = \tan\theta$:

$$x' = \cos\theta[x - y\tan\theta] \tag{2.12}$$

$$y' = \cos\theta[y + x\tan\theta] \tag{2.13}$$

Next the rotation angles are restricted so that $\tan\theta = \pm 2^{-i}$, and only iterative rotations are allowed. Multiplication is a costly operation. However, multiplication by a 2^{-i} can be implemented by a simple shift.

$$x_{i+1} = \cos(\arctan \pm 2^{-i})[x_i - y_i\, d_i\, 2^{-i}] \tag{2.14}$$

$$y_{i+1} = \cos(\arctan \pm 2^{-i})[y_i + x_i\, d_i\, 2^{-i}] \tag{2.15}$$

The rotate direction $d_i = \pm 1$ decides the rotation direction based on the value of accumulated angles z_i, such that $d_i = -1$ if $z_i < 0$, else $+1$. For the term $\cos(\arctan \pm 2^{-i})$ note that the cosine is symmetric $\cos(\arctan +2^{-i}) = \cos(\arctan -2^{-i})$ and becomes:

$$K_i = \cos(\arctan 2^{-i}) = 1/\sqrt{1 + 2^{-2i}} \tag{2.16}$$

K_i from equation 2.16 is the magnitude or scale constant can be removed from the iterative equations and computed offline.

$$x_{i+1} = x_i - y_i\, d_i\, 2^{-i} \tag{2.17}$$

$$y_{i+1} = y_i + x_i\, d_i\, 2^{-i} \tag{2.18}$$

$$z_{i+1} = z_i - d_i \text{ arctan}(2^{-1}) \tag{2.19}$$

The arc tangent values are pre-computed. In order to compensate the gain given by K_i, the result has to scale with the reciprocal value of the gain.

$$\mathbf{A}_n = \prod_n K_i^{-1} = \prod_n \sqrt{1 + 2^{-2i}} \tag{2.20}$$

In [25] Andraka presents a survey of CORDIC algorithms. The focus being on FPGA implementation. The following computation modes are covered: sine and cosine, Polar to Rectangular transformation, arc tangent, vector magnitude, cartesian to polar transformation, inverse CORDIC functions, arcsine and arccosine, extension to linear functions, extension to hyperbolic functions. In [26] they present an implementation of a modified CORDIC algorithm. The pre-calculated arctangent angles are instead replaced with an approximation based on Taylor series expansion. Wave generation is used to test the design on a Spartan XC3S500E Xilinx FPGA device. The results show that the design saves ROM space and power consumption at the cost of reduced accuracy.

2.3.3 Loop perforation

Loop perforation is a high level AC technique that skips iterations of a loop. In [43] Sidiroglou-Douskos, Misailovic, Hoffmann *et al.* presents a loop perforation technique where loops are transformed to execute a subset of their iterations. By finding the critical parts of a loop the remaining parts can be tuned such that it creates an approximate results.

Chapter 3

Methods and implementation

This chapter describes the proposed novel taxonomy and the simulation model used to test it. The implementation details and the techniques used are presented along with key design choices.

3.1 A novel taxonomy for approximate computing techniques

AC techniques can generally be categorized as operating at the software-level or at the hardware-level [44]. It can also be convenient to categorize the techniques based on application area, or by what type of device or component the technique is designed for [1][2]. However, neither of these approaches account for the intrinsic nature of approximate computing. Is there some fundamental way that approximate computing works? Well, the accuracy of the outcome is what determines the approximation level. Where accuracy refers to how close the approximated value is to the correct value. When it comes to computing there are two ways the accuracy can be reduced: (1) by changing the data or data structure of the inputs, or (2) by modifying the algorithm employed. In other words; Approximation techniques can operate directly on the data or on the methods that transform the data for a given computation.

Changing the data structure to allow for AC is, primarily, that of changing the precision of the data. Precision is the measure of detail, i.e. the number of digits or bits used to represent a value. Precision can also be dealt with indirectly. That is, for an application that uses lookup in memory instead of computing at run time, the precision must

be determined in the pre-processing. While still being dependent on the precision, rearrangement of the data can also affect the accuracy and efficiency of a system. The discretization level of the lookup table (LUT), i.e. the sample range, infers a precision constraint on the indexing operation. A LUT with 10 samples can not be accessed at index 20. The input value must therefore be scaled, or a hash function can be used to compute the slot. The data structure approach can be divided into two subcategories: (1) techniques that directly reduce the precision of the data, and (2) memory based approximation techniques that depend on both precision and data arrangement.

An algorithm is "a set of guidelines that describe how to perform a task" [45, p. 1]. In essence, it describes the transformation of the data. When it comes to approximate computing there are two ways to reduce the accuracy at the algorithm level: Using algorithms that iteratively increase the accuracy of the outcome, or by means of incorporating errors into the instruction set.

It must be noted that the accuracy is related to the correct value and that this correct value must be defined somewhere. Thus, approximate computing techniques are not approximate by themselves. Without relation to a measure of accuracy they are only computations, determined by some input and a method of transformation.

To summarize, the classification of approximate computing techniques boils down to the question: Where does the alteration occur that makes it approximate? The answer being either in the data structure, as a change in precision or the memory, or at the algorithm level, as incorporation of error or use of iterative techniques. Based on this a classification for approximate computing techniques is presented in Table 3.1. The techniques need not be exclusive to one domain, yet the idea is that components can be broken down into the different subcategories.

Table 3.1: Classification of approximate computing techniques.

		Description	Example of techniques
Data structure	Precision	Techniques that directly reduce the precision of the data that is to processed.	Mantissa bit-width scaling: [29]*, Fixed point precision optimization: [30], Dynamic precision scaling: [28] [31], Memory access dynamic precision: [32]**
	Memory	Concerned with the precision and discretization level of pre-computed values that are used in the computations and the data organization to create approximate results.	LUT allocation based on kernel input-output relationship: [46] , Data block organization: [32]**, Memory processing in DRAM: [47] [48], Load value approximation: [49], LUT optimization of KCM: [50]
Algorithm	Iterative	Algorithms that iteratively increase the accuracy of the outcome. Can be tuned for different levels of approximation.	CORDIC/Trigonometric functions: [42] [26] [27], Conjugate Gradient parallel solver: [29]*, Minimal residual algorithm for FPGA: [51], Dynamic optimization of iterative methods: ApproxIt Framework [52] and MIPAC Framework [53]
	Error	Techniques that incorporate error in the instruction set of an algorithm or change the model used. The result may have lower accuracy.	Loop perforation: [43], Approximate Full Adder Cells: [15] [16], Reconfigurable Adder: [17] , Inaccurate 2x2 multiplier: [18], FPGA specific approximate adders: [41] [14], FPGA specific multipliers: [21] [54] , FPGA specific constant coefficient multipliers: [19] [22], ACCEPT compiler framework: [23], Voltage overscaling: [20], Neural accelerators: SNNAP Framework [24]

**The technique presented by Roldao-Lopes, Shahzad, Constantinides* et al. *use both Mantissa bit-width scaling (Precision) and a Conjugate Gradient parallel solver (Iterative).*

***Tian, Zhang, Wang* et al. *present a technique that combine memory access for dynamic precision (Precision) and data block organization (Memory).*

3.2 Model Description

The purpose of this model is to present a simulator where a system approach to approximate computing is applied. With the overall intent that this will provide a sufficient test environment for the taxonomy, and that the comparison of AC techniques applied separately or in combination will give new insight into the field when the results are examined higher in the system hierarchy.

The system chosen is one where the objective is to detect certain AVs from an image feed, in particular the detection of planes and helicopters. To do this, contours of possible detection objects are extracted and ranked according to a set of rudimentary shape criteria. In order to extract the contours the detection algorithm needs to find the edges in the image. The Sobel operator is used for this purpose, and provides the detection algorithm with an approximation of the gradient magnitudes. To properly extract a closed contour the detection algorithm also uses the gradients direction to walk around the outline of an object.

The system can be split into two distinct sub-systems, the detection algorithm and the Sobel filter. It is only the Sobel filter that will be made available for AC techniques. The internal operations of the detection algorithm are considered fixed, thus the only way to change the output is to provide a different input. An input, which in turn is determined by the Sobel filter.

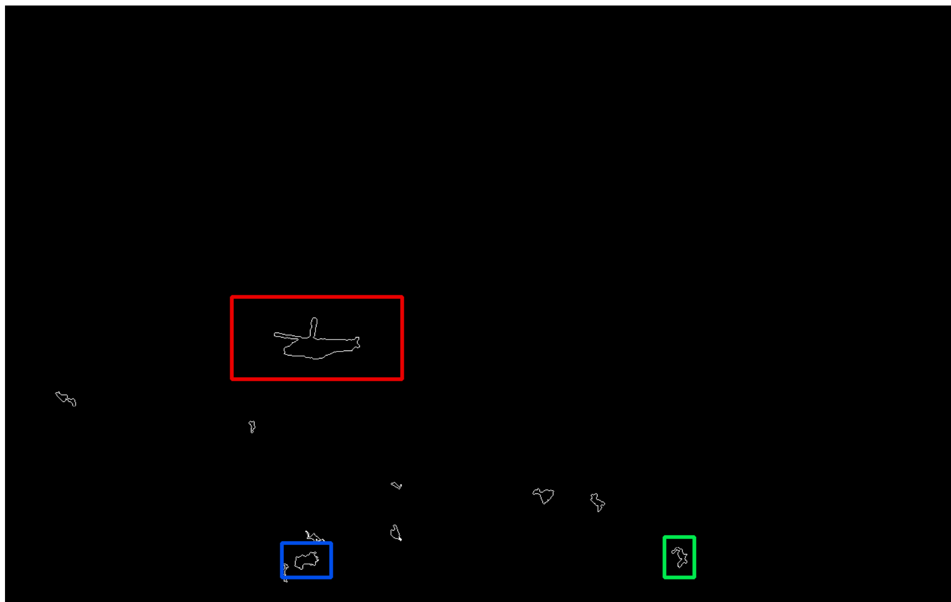
This system is placed in a testbench that provides the image feed for six test cases and validate the results. A correct detection is considered valid only when the highest ranked closed contour matches the global output provided by the testbench.

3.2.1 Object detection using contour extraction

The input the detection algorithm take is one direction image and one magnitude image, both images being 16 bit single channel images. The representation of angles in the direction image must be in the range from -180° to 180°, and the magnitude intensity must be in the range from 0 to 65535. The transformation outputs a contour image along with the position and size of the contours, which are ranked by the best detection fit. Figure 3.1 show an example of a correct detection of an AV together with the second and third highest ranked contours.

(a) Original image.

(b) Gradient magnitude.



(c) Contour image with the three highest ranked contours marked.

Figure 3.1: A helicopter is detected by the the system. The contour is extracted and the position marked by a red rectangle. The noise from the trees produce detections and two contours of lesser rank are marked by the blue and the green rectangles.

The structure of the object detection algorithm is loosely defined by three main steps:

1. Apply a threshold to the gradient image that only saves the high intensity edges, and use a kernel to exclude noise.
2. Then, for each possible contour do an edge walk on the threshold image using the gradient direction.
3. Determine if the closed contour fits the shape criteria, and if it does, rank it accordingly.

Gradient threshold

The first step towards extracting the contours is determining which edges to keep. Thus, a threshold is put on the intensity of each pixel in the gradient magnitude image. The threshold value is chosen as the squared average of the mean of the image, and is found using equation 3.1. This is an arbitrary value, that combined over the six cases give a threshold value of approximately 960.

$$\text{Squared average mean} = \left(\sum_{1}^{c} \sum_{1}^{f} \sum_{0}^{m-1} \sum_{0}^{n-1} g(i,j) \frac{1}{mncf} \right)^2 \tag{3.1}$$

g represents the matrix data of the image frame.
f represents the frame count in each case.
c represents the number of cases.
m represents the numbers of pixel rows and **i** represents the row index.
n represents the number of pixel columns and **j** represents the column index.

With the obtained edge image the algorithm finds the next possible seed point to start a contour walk. The seeds are only valid if the pixel is part of a 3 by 3 kernel where all individual pixels are above the threshold. This is done to reduce noise by excluding pixels that are not part of a refined edge.

Contour walk

When a valid seed point is found the contour walk starts. The walking process steps one pixel to the 'right' and one pixel to the 'left' for each iteration. The idea is that when they meet again after having walked the outline of an object a closed contour has been found.

The gradients direction is used for choosing the next pixel for the right and left walk. For a single step the left and right walk can only go to one of the eight neighbouring pixels. Since the gradients direction is a good approximation of the normal to the edge it can be used to force the left and right walk to always step in the correct direction. When choosing the next pixel for the left walk the neighbouring pixels are checked in a counter-clockwise manner until a valid pixel is found. It first checks if the pixel to the left of the normal is valid, and like a clock with eight ticks goes around until it finds a valid pixel. The process is the same for the right walk, with the distinction that it first checks to the right of the normal and goes in a clockwise manner.

Listing 3.1: Contour walk – Left step

```
for each i in image rows
    for each j in image columns
        if ( all neighbors to pixel(i,j) > threshold )
            while n < max step count
                Direction = Angle image(i,j)
                Starting neighbor = Angle-to-position(Direction)
                for each neighbor to pixel(i,j)
                    K = Starting neighbor
                    if K > threshold
                        Next pixel = K position
                        break
                    K = K - 1
                    if K < 0
                        K = 8
                (i,j)  ← Next pixel (i,j)

 Angle-to-position function : input Direction
    if Direction == 0
        return  relative position(bottom,left)
    if Direction > 0 AND Direction < 45
        return relative position(middle,left)
    if Direction < 0 AND Direction > -45
        return relative position(bottom,left)
    ***Do for all 16 possible positions***
```

When a pixel is determined to be on the left or right side of the normal the center of the pixel used as the reference point. Thus there are eight options when the direction points exactly at a pixel center and eight options when the direction points between the center of two pixels. This means that the walk has 16 distinct states for choosing the starting pixel at each step. The pseudocode for the left contour walk is given in Listing 3.1.

Only closed contours are found using this technique, lines and other open contours are discarded since the gradients direction prohibit back-tracking on the same edge. However, this means that sharp corners open for infinite looping between the same two pixels. To account for this a loop breaking mechanism is employed. The walk stores the position history of the previous pixel and refers to this when a new pixel is chosen. If the previous pixel position matches the new position a flag is raised. At the next step the walk skips the two first positions. The reason a skip factor of two is used instead of one, is because the next position could point right back at

the loop sequence. With a skip factor of two this scenario becomes highly unlikely. The next pixel chosen could be one layer below the outline edge. This proves unproblematic as the walk will correct itself on the next step. It always pushes towards the outline. The mechanism does not account for loops with three or more steps involved.

The walk concludes when the left and right process step onto the same pixel, or if they step onto two pixels that coincide after the initial step. The other way the walk concludes is if either the left or right walk reach the starting seed after ten steps. These two conditions for termination give closed contours, and the shape is passed to the next phase, the criteria checking and ranking. The walk can also terminate without having found a closed contour when the step count exceeds the stepping limit, which is set to two times the image width.

Contour shape criteria

After a contour has been extracted by the edge walking process it is submitted to three shape criteria to determine if it should be discarded or put in the ranking list.

The first criterion is a squareness test. The contour walk keeps a history of the x- and y-axis minimum and maximum positions. In turn, this gives the width and height of the shape. The criterion is listed in equation 3.2. If the length of the width and height is within 30% of each other the shape is considered too square, and is discarded. The shape of the AVs that the system tries to detect has certain properties. One being that the ratio between the two visible principal axes are above 30% when the orientation is close to the image. This means that at orientations close to 45° in either quadrant a viable contour can be discarded. The benefit of the criterion is that a substantial amount of image artifacts are within this range, and are removed from the ranking. Figure 3.2 show image artifacts removed by the criterion. Without the criterion these artifacts would have been put into the ranking list.

$$Pass\ \neg\ \frac{w}{h} < 1.3\ \wedge\ \frac{h}{w} < 1.3 \tag{3.2}$$

w represents the contour width.
h represents the contour height.

The second criterion is a test for removing lines. The criterion is listed in equation 3.3. If the ratio between the width and height is above 5 or below

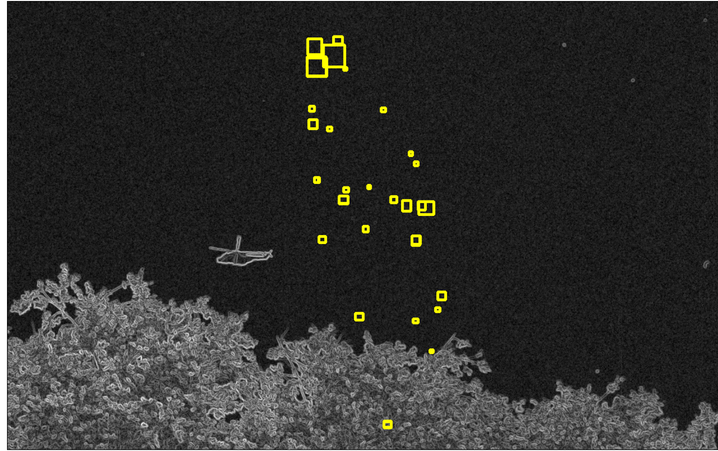


Figure 3.2: Image artifacts removed by the squareness test.

1/5, the shape is considered too long and is discarded. This test faces the opposite problem of the squareness test. Lines and artifacts that should be discarded can be kept if they are oriented favorably.

$$Pass \neg \frac{w}{h} > 5 \lor \frac{w}{h} < \frac{1}{5} \tag{3.3}$$

w represents the contour width.
h represents the contour height.

The third criterion is a size check of the circumference. The walk stores the step count. With the assumption that each step increases the length of the contour with two pixel distances, the step count can be used as a measure of the circumference. The criterion is listed in equation 3.4. The circumference must be above 41 pixel neighbor distances, i.e. that the step count has over 20 steps.

$$Pass \Leftarrow n > 20 \tag{3.4}$$

n represents the step count.

When the shape passes the criteria it is ranked according to the size of the area it occupies. The shape with the largest area, given by the width times the height, is deemed the best fit. The list incrementally keeps track of the size hierarchy and the center position of the contour area. The reason area is used instead of circumference for the ranking, is because of the assumption that the step count always steps two pixel distances. This need not be the case. When a looping sequence occurs where one of the walk directions

get stuck, the other walk process might still reach it or the starting point. Thereby producing a false circumference that is larger then the true outline. It is also a problem that noise and image artifacts create fractured outlines that produce a high step count while still being contained within a small area. Using the area for the ranking absolves these issues to a high degree.

3.2.2 Sobel filter

The task for the Sobel filter is to produce the gradient and direction images that the detection algorithm take as input. The Sobel filter takes a single channel image and computes the image gradients x and y derivatives using two three by three kernels. Then the derivatives are used to compute the gradients magnitude and direction. The Sobel filter module can be divided into three main steps:

1. Create a window of three by three neighbouring pixels.
2. Use the window and perform the convolution for the x and y derivatives.

$$Dx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad Dy = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

3. Then compute the magnitude $G = \sqrt{D_x^2 + D_y^2}$ and the direction $\theta = atan2(D_y, D_x)$.

To allow for a wide range of AC techniques to be implemented the filter is hardware accelerated for a system-on-chip with programmable logic. The exact model being a Zynq-7000 evaluation board with part number *xc7z045* and package option *ffg900-2*. The acceleration is performed using HLS with Vitis HLS. The filter behavior is implemented in C and synthesized to a hardware circuit. Identical timing constraints are applied on the design variants. Using HLS greatly reduces the development time compared to development at the register transfer level (RTL).

3.2.3 Hardware acceleration of the Sobel filter

When an image is sent to an FPGA it will typically be transferred in a raster-scan manner that streams data pixel by pixel [55]. This streaming process is shown in Figure 3.3.

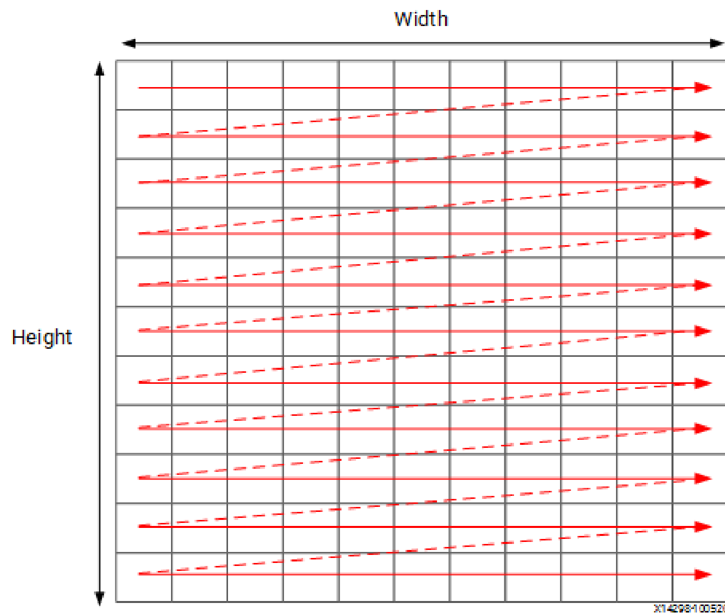


Figure 3.3: Raster scan streaming of pixel data. The process goes pixel by pixel across each line before stepping down to read the pixels at the next line. (Xilinx [55])

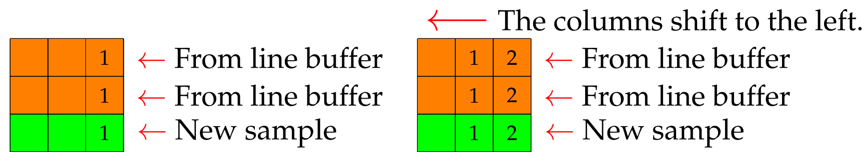
Implementing the Sobel filter in a hardware efficient way is primarily done by maximizing the flow of data and avoiding re-reads. In short the FPGA prefers data samples which continuously flows through the implementation. An image consist of several parameters which determines the memory space it occupies. For any image there is width and height, the number channels and the pixel depth. The required memory can be found:

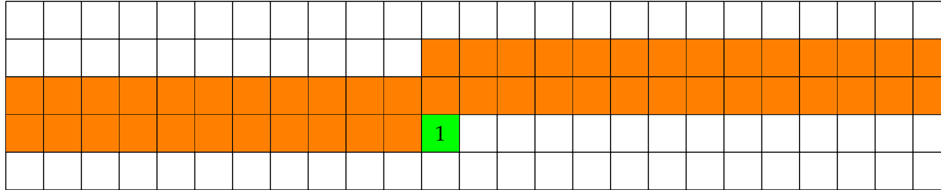
$$Memory = Width \times Height \times Channels \times Pixel\ Depth \tag{3.5}$$

If the intermediary images for the x and y derivatives, as well as the gradient magnitude and direction are to be stored on the device the filter might require an inordinate amount of resources. This not needed. The limiting factor for reducing the intermediary storage is primarily determined by the kernel height and the image width.

When an image is streamed through the Sobel filter module a window of three by three neighbouring pixels is used to compute the derivatives. Since the pixels are read in a streaming manner a line buffer is implemented. It stores the two previous lines. When a new sample is read another sample is pushed out and the current column is shifted one line down.

The window also updates as new samples are read in. A key point is that the window reads the new sample value and a new column from the





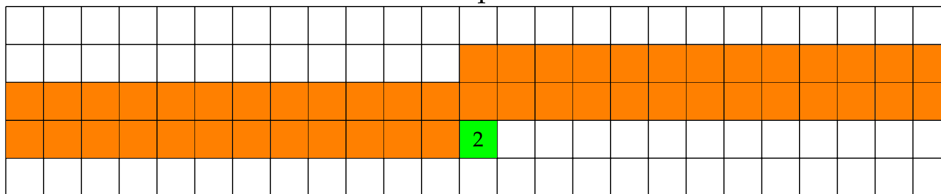


Figure 3.4: The window reading in a new sample and a column from the line buffer. When the next sample is read in the columns shift to the left. When the line buffer reads in a new sample the columns are shifted down.

line buffer before the line buffer is updated. The process of updating the window and line buffer is shown in Figure 3.4. This process creates a delay from the first read in to a window can be streamed out. Which is due to the fact that a full window is not obtained until a sample has been read into the bottom right position. The number of samples needed is the width of the image plus the number of kernel columns to the right of the center pixel.

For this implementation the edge pixels are clamped to zero. By doing so no overhead is created for filling in values outside the bounds of the image.

An advantage that comes with hardware acceleration is that the data width can be controlled at every step. The input data width is set to eight bits and represent a pixel intensity in the range from 0 to 255. When the derivatives are computed the precision must be increased to retain the full range of possible results. In a software approach the results would normally be put into a 16 bit image. However, only an increase by three bits is needed, since the derivative values are in the range from -1020 to 1020. Both the gradients magnitude and direction can be represented accurately at a bit width of 11, given that they are represented as whole numbers. The magnitude values are in the range from 0 to 1442 which can be represented by 11 bits as unsigned integers. The angle representation of the gradients

direction is in the range from -180 to 180. In this implementation the fractional components of the magnitude and angle representation are cut off. This is done because the detection algorithm operates on integers, and because a precision scheme without fractions is considered sufficiently accurate.

There are four data transformations occurring in the Sobel filter module. The convolution for the x derivative, the convolution for the y derivative, the computation of the gradient magnitude and the computation of the gradient direction. The convolution operations run in parallel, and consist of three steps plus an initialization phase that also keeps track of the pixel count. The steps are as follows: (1) the window is read in, (2) the convolution is performed concurrently for each element and summed, and finally, (3) the derivative is streamed out.

The computation of the gradient magnitude and direction is also run in parallel. The magnitude is found using equation 2.8. The multiplication and summing of the derivatives are unproblematic. However, finding the square root requires using an internal HLS function, the fixed point square root function. Due to limitations in the simulation software the input for the fixed point square root function is cast to floats. Meaning they now have a width of 32 bits. This adds overhead to the magnitude computation.

For the gradient direction the angles are computed using the 2-argument arctangent, know as the atan2 function. The CORDIC algorithm is employed to perform this computation, and the precision is set to nine rotations. Which gives a margin of error of approximately 0.05 degrees. The implementation is based on the approach proposed by Xilinx [56]. The angle representation is chosen such that at each iteration the tan(angle) is equal to n right shits, where n stands for the iteration count. The CORDIC algorithm uses a LUT for the pre-computed angles that correspond to these shift values.

The filter has added read in and read out functionality that use AXI memory mapped interfaces. This handles the initialization and termination of each frame, and the pixel sampling count from each frame. However, the actual memory is not located in the hardware, but handled by the testbench. The system schematic with the data flow through the internal modules of the Sobel filter is shown in Figure 3.5. As can be seen from the figure, pre-processing and post-processing of the image data is also performed by the testbench. These two operations are added to scale the data if needed.

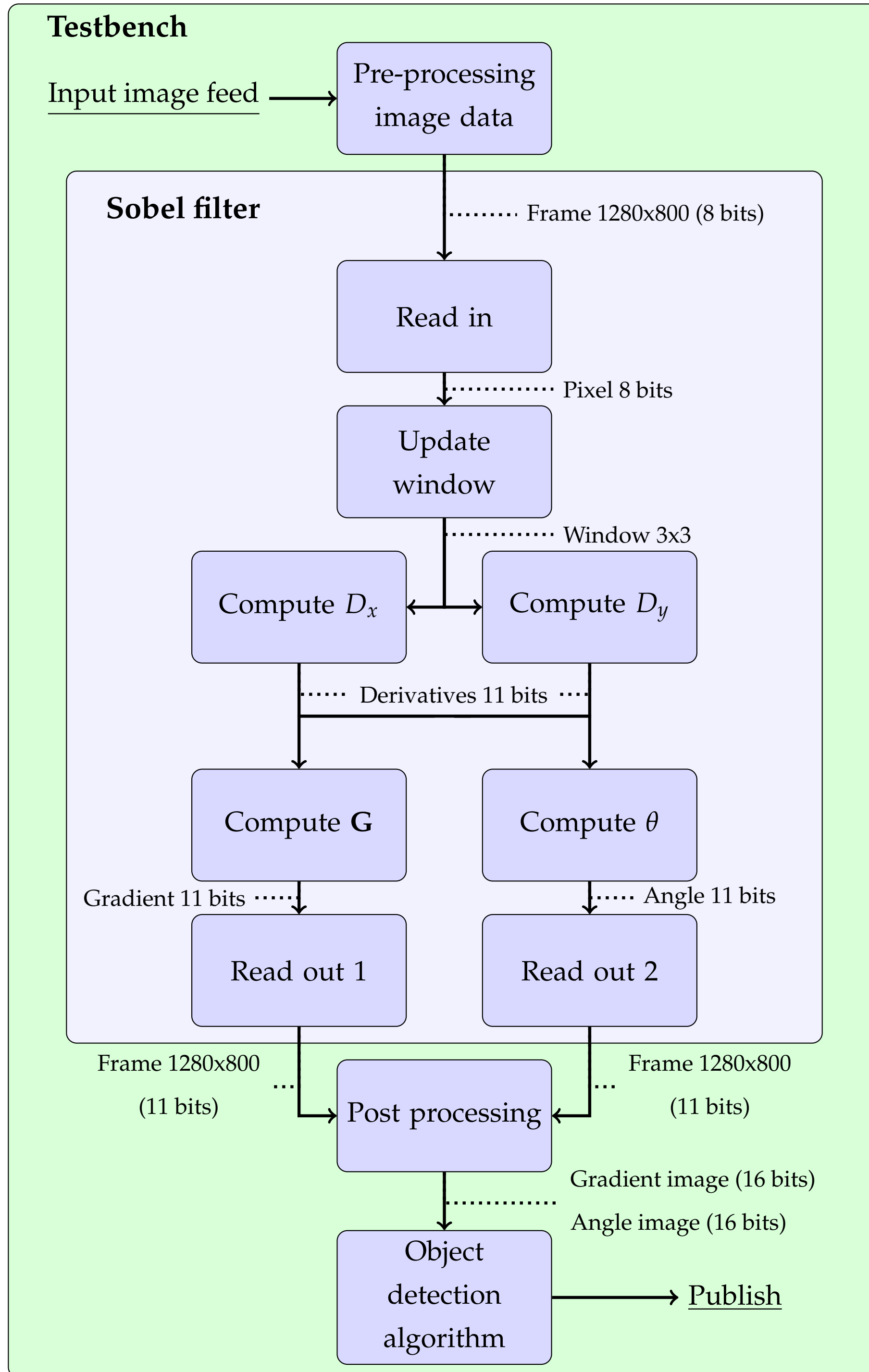


Figure 3.5: The testbench and Sobel filter program flow. The data representation and precision between functions is indicated by the dotted lines.

The following sections describe the techniques proposed for the system for the different classes. Starting with techniques that are categorized within *precision*, then *memory*, next *iterative*, and finally *error* inducing.

3.2.4 Precision

Two precision scaling techniques are implemented on the Sobel filter, one for the input and on for the intermediary derivative results. The precision scaling impose the precision scheme throughout the Sobel filter. Meaning that changing the input precision dictate the precision of the output. The post-processing must therefore be scaled accordingly. Thereby insuring that the data representation is correct before the data is passed on to the detection algorithm.

Precision scaling of the pixel input

Precision scaling of the input pixel depth is carried out by truncation of the LSB. This is a scalable technique and several approaches are tested for the technique. The precision scaling is performed with a reduction in precision from seven bits to a single bit. Each reduction in precision set the required bit width for the subsequent intermediary outputs and the final output. Due to the nature of the data transformations the precision of these outputs are always three bits wider than the input.

Precision scaling of the intermediary derivative output

Precision scaling of the intermediary derivative output is also carried out by means of truncation of the LSB. This technique is scalable as well and approaches using from ten to three bits are tested on the filter.

3.2.5 Memory

This Sobel filter implementation does not use a LUT for any of the primary sub-modules. It is only within the CORDIC algorithm that pre-calculated values are used for the incremental addition or subtraction of angles. Even though it would be possible to approximate this LUT, the CORDIC algorithm is in itself an AC technique. Averting from nesting techniques within each other, there is another function which could benefit from computing with memory. That is the computation of the gradient magnitude. The magnitude is computed by first squaring the x and y derivatives and then adding them together. Then, the square root of this value is computed. Instead a LUT approach is implemented, that takes two slot values and returns the magnitude directly.

Lookup table for gradient magnitude

Since the range of values are from -1020 to 1020 the absolute value for each input is found before indexing the LUT. The derivatives use two's compliment, thus only the most significant bit (MSB) needs to be checked. If the number is negative the corresponding absolute value is found by a logic NOT and a left shift.

Implementing a LUT that has 1021 by 1021 indexing slots and returns a value with a width of 11 bits requires a memory of 11444400 bits. This is a large LUT. In comparison the line buffer use 20480 bits. In fact, the HLS tool is not able to properly synthesize a table of that size. Instead of using 1020 by 1020 indices the LUT is reduced to a discretization of 128 by 128, occupying a memory of 180224 bits. The derivatives must now be scaled before indexing the LUT. Conveniently the discretization of the LUT correspond to a seven bit representation. The derivative values are therefore shifted three times to the right before the indexing operation.

Modified lookup table for gradient magnitude

If a LUT is made that has 1021 by 1021 slots yet a return value with a bit width of one, then the memory requirement is reduced to 1042441 bits. The gradient magnitude is only used by a threshold function. The single bit return value is therefore made to represent magnitudes above or below the threshold. This is a set value, and corresponds to 960 in the 16 bit representation, or for the 11 bit representation:

$$t = 960 \times \frac{2^{11} - 1}{2^{16} - 1} = 29.9858 \approx 30$$

The LUT can be reduced further. The magnitude will always be above the threshold value if either of the derivative values are 30 or above. Instead of using a 1021 by 1021 LUT it is reduced to a 30 by 30 LUT. This adds two compares to the design, which are needed to determine whether to index the LUT or to set the value directly. The pseudocode for the technique is given in Listing 3.2 The approach uses a LUT which occupies a memory of 900 bits, and is only indexed if both the x and y derivatives are below 30.

3.2.6 Iterative

The only measure needed to integrate an iterative AC technique is to change the CORDIC algorithm.

Listing 3.2: LUT 30 by 30 pseudocode

```
if (x[MSB]==0)
    x = NOT(x<<1) //Shift and NOT to change signedness
if (y[MSB]==0)
    y = NOT(y<<1) //Shift and NOT to change signedness
G = if (x >= 30 OR y >= 30) then 1 else LUT(x[0:6],y[0:6])
```

Reduced CORDIC rotation steps

Making the CORDIC algorithm less precise is done by reducing the number of rotation steps. To further reduce the resource usage, the internal LUT for the angles is scaled down to match the current iteration count as well. This is a scalable technique and is tested for approximate approaches ranging from five to one rotation.

3.2.7 Error

Three error inducing techniques are introduced to the system. All three techniques operate on different sections of the system. Loop perforation is used on the overall streaming loop and reduce the number of pixels that are sampled from the image. A kernel reduction approach is implemented that change the calculation of the x and y derivative. Resulting in a smaller line buffer and removing the read in delay for the window updating module. An approach that completely replace the CORDIC algorithm and the way the angles are computed is presented as well.

Loop Perforation

The read in and read out functionality allow for different sampling rates, than simply reading in the full frame. Instead of going through all the pixels, a loop perforation technique is implemented that skips every second column of the image. Thereby reducing the sampling count by 50%. This technique uses the read in function to skip the samples. The read out function fills inn the missing values. This is done by extending the results of one column to two columns for the final output. This techniques has two approaches. The first is as described above, with no further functionality added. The second approach is a scaled loop perforation technique.

Using the loop perforation technique changes the distance from the center to its neighbors. The new weighing distances for the Sobel filter

is given in equation 3.6.

$$Original = \begin{bmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & c & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{bmatrix} \qquad New = \begin{bmatrix} \sqrt{3} & 1 & \sqrt{3} \\ 2 & c & 2 \\ \sqrt{3} & 1 & \sqrt{3} \end{bmatrix} \tag{3.6}$$

Using these new distances create complexity for the convolution operation. However, this is approximate computing, and the implementation need not be exact. The y derivative see less change than the x derivative, and is set to operate as before. Whereas the weighing distance for the x derivative is approximately twice as long. For the scaled approach the y derivative is kept as is, while the x derivative value is halved using a right shift. The new convolution kernels for the scaled approach are found in equation 3.7.

$$Dx = \begin{bmatrix} -0.5 & 0 & 0.5 \\ -1 & 0 & 1 \\ -0.5 & 0 & 0.5 \end{bmatrix} \qquad Dy = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.7}$$

Kernel reduction

The kernel reduction method does not sample for the last row of the Sobel filter kernel. The second row is instead expanded to represent the third row as seen in equation 3.8.

$$Original = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad New = \begin{bmatrix} a & b & c \\ d & e & f \\ d & e & f \end{bmatrix} \tag{3.8}$$

This method also changes the distances from the center to the neighbouring positions. The method is therefore split into two approaches, one that operates as is, and another that scales the output. A way to look at this reduction is that the y derivative now finds the gradient between the b and original e position for the given kernel. To avoid overhead the sum of the new weighing distances is set to be approximately half of the original in the y-direction. This gives a scaled approach where the y derivative is doubled. Which is done using a left shift for the intermediary output. And, where no scaling is performed on the x derivative. The new convolution kernels for the scaled approach are found in equation 3.9.

$$Dx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad Dy = \begin{bmatrix} 2 & 4 & 2 \\ 0 & 0 & 0 \\ -2 & -4 & -2 \end{bmatrix} \tag{3.9}$$

Reduction of angle representation to 16 states

The detection algorithm has 16 states for choosing the starting pixel at each step. The normal of the edge can point exactly at a pixel center or between two. This means that the representation of angles found when computing the gradient direction can be set to directly produce these 16 states. This is done using a set of compares. The pseudocode for the technique is given in Listing 3.3. First a zero check is performed on both derivatives. If either is zero the value is set to the corresponding angle of either -90, 0, 90 or 180 degrees. If not, then the quadrant is found by comparing the MSB of the derivatives. The absolute value of any negative value is found using a NOT and a left shift. For the four quadrants the angle is found by a compare to be: between 0 and 45 degrees, exactly 45 degrees, or between 45 and 90 degrees. This is a minimal approach to finding the angles that by nesting three compares finds the 16 states used by the detection algorithm.

Listing 3.3: Angle representation to 16 states

```
if (x!=0 AND y!=0 )
    if (x[MSB]==0 AND y[MSB]==0)
        if (x > y)       Angle = 22
        else if(x < y)   Angle = 67
        else             Angle = 45
    else if (x[MSB]==0 AND  y[MSB]==1)
        y = NOT(y<<1) //Shift and NOT to change signedness
        if (x > y)       Angle = -22
        else if(x < y)   Angle = -67
        else             Angle = -45
    else if (x[MSB]==1 AND y[MSB]==0)
        x = NOT(x<<1) //Shift and NOT to change signedness
        if (x < y)       Angle = 112
        else if(x > y)   Angle = 157
        else             Angle = 135
    else if (x[MSB]==1 AND y[MSB]==1)
        if (x > y)       Angle = -122
        else if(x < y)   Angle = -157
        else             Angle = -135
else if (x==0 AND y== 0)    Angle = 0
else if (x == 0)         Angle = y[MSB] == 0 ? 90 : -90
else if (y == 0)         Angle = x[MSB] == 0 ? 0 : 180
```

3.2.8 Combined approaches

Certain techniques can be implemented simultaneously without any further adaption of the system. Others, need added functionality to interact

properly. Since the precision reduction techniques regulate the precision scheme of the system, they are not affected by other techniques. The CORDIC algorithm, loop perforation, kernel reduction and angle reduction techniques are unaffected by the precision and representation of data outside their functionality. The LUT does, however, rely on the precision and representation to function correctly and must be scaled, or changed appropriately, to fit the current data flow.

Whenever the precision changes so does the LUT. If the precision is reduced then the LUT must scale the return values to accommodate this. For the modified LUT the pre-computed values must be computed again and the compares must be set for the precision. If the precision is reduced by two bits then the threshold value becomes $30/4 = 7.5$. The return values of the LUT are one if the magnitude of the indexing values are above 7.5. An example of the altered LUT pseudocode for a two bit reduction is shown in Listing 3.4.

Listing 3.4: LUT 8 by 8 indexing slots

```
if (x[MSB]==0)
    x = NOT(x<<1) //Shift and NOT to change signedness
if (y[MSB]==0)
    y = NOT(y<<1) //Shift and NOT to change signedness
G = if (x >= 8 OR y >= 8) then 1 else LUT(x[0:4],y[0:4])

LUT[8][8] = {
{0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 1, 1, 1},
{0, 0, 0, 1, 1, 1, 1, 1}}
```

Chapter 4

Experiments and results

This chapter presents the simulation experiments, the results and analysis. First the different detection cases used in the simulation experiments are presented. Key properties and to what extent they represent edge cases for the system is given. Then the simulation results for the unaltered system is provided. Next the results and analysis for the single method implementations is presented. Following this the results and analysis of the combined techniques is given. Lastly, a comparison of both combined and single method approaches is presented, along with an analysis of the correlation between the intermediary accuracy reduction compared to the overall system performance.

The system without any integrated AC techniques will from here on out be referred to as the *Reference system*. The resource usage and performance metrics from this system will be used as the benchmark for comparing the AC techniques.

4.1 Evaluation metrics and detection cases

This section describes the detection cases and the metrics used to evaluate the results.

The acceleration is performed using HLS with Vitis HLS. The filter behavior is implemented in C and synthesized to a hardware circuit. Identical timing constraints are applied on the design variants.

There was, however, some limitations on performing the RTL/C co-simulation. The time it takes to process a single frame for the co-simulation is in the range from 500-560 seconds. With multi-threading using ten

cores on an i7-9750H CPU 2.60GHz × 12. That means running all 49 approaches for six detection cases with 500 frames would take a minimum of 20416 hours, or around two years and four months to complete. Instead, only a sample from each detection case is used to validate that the RTL implementation performs as expected. The overall detection performance is found using C simulation. An important consideration that was made when choosing this, is that all AC techniques implemented are deterministic. Meaning that the output can always be traced back to the input, and that no techniques rely on timing violations or other stochastic variables.

The image feed provided read in the first 500 frames for all the cases. This is done to make the weighing of each case equal. The cases samples every 20th frame. Meaning each case consist of 25 image samples. The videos are filmed using a Phantom Miro 320 High speed camera with a 400-600 mm zoom lens. The videos are captured outside in daylight with 20 km meteorological visibility. Which is the sensor limit.

The accuracy of the system is determined by the rate of correct detections, and is referred to as the system performance. The system performance is measured by the testbench. A correct detection is made only if the center of the shape is within a region specified by the testbench. This region is verified as containing the desired AV by visual inspection. The overall system performance is the average of the combined system performance for all the cases.

The design is pipelined and uses the dataflow pragma to ensure the highest possible streaming rate. No constraints are put on the techniques and the HLS tool is free to make suitable implementation changes. Which means it can utilize all of the available resource primitives. This makes is harder to compare the resource usage for the implementations. The advantage is that the latency and throughput is accurately portrayed.

The resource usage by the design is given in the amount of Block RAM components (BRAM), digital signal processing blocks (DSP), flip-flops (FF) and LUTs used. The iteration latency for the sub-functions is presented as well. This measure refers to how many clock cycles the function uses to start the streaming loop, read in a sample, perform the transformation and then write a sample out. The throughput (TP) is given in frames per second (FPS) and is found using equation 4.1. For reference a single frame holds a data size of 1.02 MB. The clock period is set to 100 MHz for all approaches.

$$FPS = \left(\frac{\text{System latency}}{\text{Clock period}}\right)^{-1} \tag{4.1}$$

AC is suitable for error resilient applications [1]. The reason applications are error resilient is in part due the difficulty of quantifying the accuracy of the results. It is desired to compare the accuracy of the intermediary results to the system performance. However, a metric must be chosen to do so.

Quantifying the accuracy for image degradation is a common problem in AC [2]. In [12] the Structural Similarity Index is used to quantify the accuracy reduction. In [9] they use peak-signal-to-noise ratio (PSNR) to quantify the accuracy. PSNR is usually expressed in the logarithmic decibel scale. The PSNR is calculated from equation 4.3.

$$MSE = \frac{1}{mn}\sum_{0}^{m-1}\sum_{0}^{n-1}||f(i,j) - g(i,j)||^2 \tag{4.2}$$

$$PSNR = 20log_{10}\left(\frac{Max_f}{\sqrt{MSE}}\right) \tag{4.3}$$

f represents the matrix data of the original image.
g represents the matrix data of the degraded image.
m represents the numbers of pixel rows and **i** represents the row index.
n represents the number of pixel columns and **j** represents the column index.
Max_f is the maximum signal value in the original image.

A more direct approach is to use the error distance. This is done in [11] to compare between approximate adders applied to the Sobel filter. Error distance is chosen as the metric to represent the accuracy of the intermediary results. In image processing terms error distance becomes the average pixel difference (APD). The APD is found by using equation 4.4.

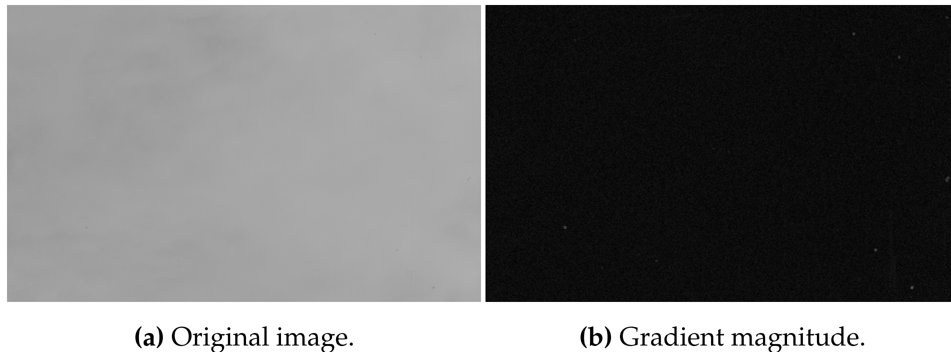
$$APD = \frac{1}{mn}\sum_{0}^{m-1}\sum_{0}^{n-1}||f(i,j) - g(i,j)|| \tag{4.4}$$

f represents the matrix data of the original image.
g represents the matrix data of the degraded image.
m represents the numbers of pixel rows and **i** represents the row index.
n represents the number of pixel columns and **j** represents the column index.

The testbench keeps track of the difference for the intermediary results between the reference system and AC approaches.

4.1.1 Case 1 – No detection objects

The first case the system is tested for, is one without a viable object to detect. The image feed still contain noise and some image artifacts. An image from the feed is shown in Figure 4.1. The system passes the test if the ranking list is empty.

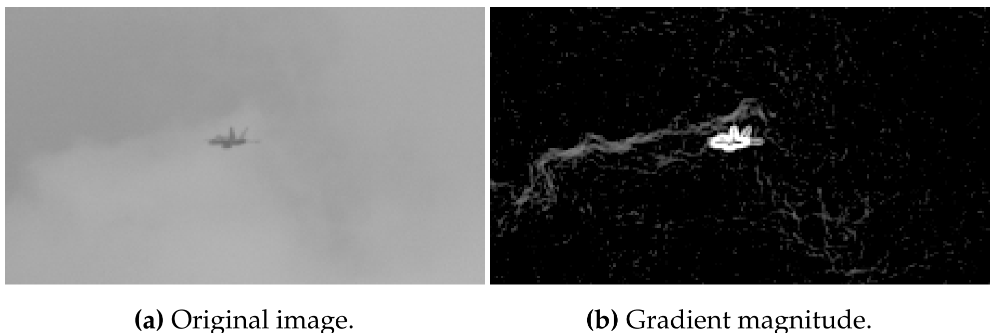


(a) Original image. **(b)** Gradient magnitude.

Figure 4.1: The original image and gradient magnitude image for the first case.

4.1.2 Case 2 – High background noise

The second case has a high degree of background noise. And, the AV that is to be detected, at times overlap with regions where the background noise produce edges. The detection algorithm does not have any procedures for separating shapes. This case propose a challenge for the system. The AV that is to be detected as well as the outline of a cloud can be seen in Figure 4.2.

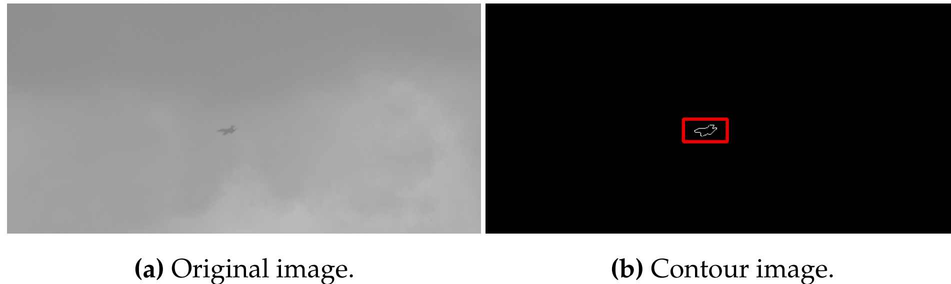


(a) Original image. **(b)** Gradient magnitude.

Figure 4.2: The original image and gradient magnitude image for the second case. The images are cropped to the content.

4.1.3 Case 3 – Low contrast

In the third case there is low contrast between the AV and the background. An unaltered image from the feed and the resulting contour image with detection markers is shown in Figure 4.3



(a) Original image. **(b)** Contour image.

Figure 4.3: The original image and the contour image for the third case. The images are cropped to the area containing the AV. The only shape extracted is that of the plane. This is a correct detection by the system.

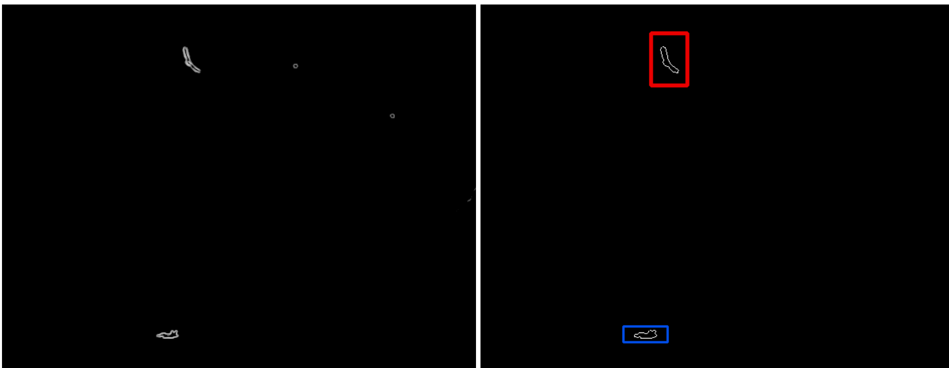
4.1.4 Case 4 – Multiple objects

Detection case four provide an image feed with two objects, one AV and a bird. The detection is considered valid if the highest ranked shape is for the AV. A situation where the bird is given the highest ranking instead if the AV is shown in Figure 4.4.

4.1.5 Case 5 – Large object

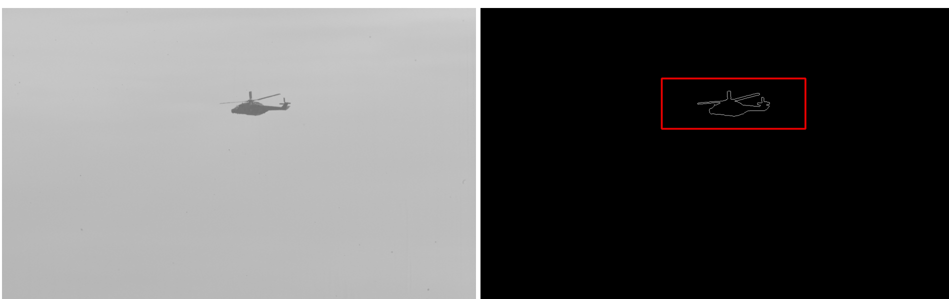
In the fifth case there is only a single large AV. The background noise is minimal and the contrast is high. This is an ideal detection environment. An example from the image feed and resulting contour image is shown in Figure 4.5.

(a) Original image.



(b) Gradient magnitude. **(c)** Contour image.

Figure 4.4: The image feed provided in the fourth case. The resulting gradient magnitude image and the contour image with detection markers is also shown. The highest ranked shape is given to the bird. A correct detection is not made. The gradient magnitude and contour images are cropped to content.



(a) Original image. **(b)** Contour image.

Figure 4.5: The original image and the obtained contour image with detection markers for the fifth case. The system correctly detects the AV.

4.1.6 Case 6 – Large object and foreground noise

Case six present an environment where a high contrast AV is to be detected with a substantial amount of foreground noise. The image feed include trees that at times cover up over a third of the image. In Figure 4.6 two image samples from the feed is shown.

(a) Early. **(b)** Late.

Figure 4.6: Two samples from the sixth detection case. The AV gets closer to the foreground noise over time and the area covered by the foreground noise increases.

4.2 Reference System

The overall detection performance for the reference system is 90.56% correct detections. The case by case detection performance is for cases one to six: 100%, 60%, 96.67%, 96.67%, 100% and 90% detection correctness, respectively. Unsurprisingly, the system performs poorly when the object that is to be detected overlaps with regions with high intensity change in the background noise. This can be seen for the second case when the plane passes over the outline of the clouds. An example of this overlapping is shown in Figure 4.7.

The resource usage for the reference system is presented in Table 4.1. The delay in the Update window function, primarily caused by the buffering, is 1285 cycles. Since 1281 samples must be read in before a full window can be streamed out, the remaining four cycles account for the processing time need to update the window and the line buffer. Thus, the functionality in function use only four cycles. Computing the magnitude produce iteration latency of 15 cycles and computing the direction use 17 cycles. Other than reducing the buffer count, it is these two functions that create the longest delays in the system. Using the most cycles to perform their respective transformations.

Figure 4.7: Gradient magnitude image from the second detection case. The plane passes over a region with high intensity change in the background noise. The system does not report a valid detection.

The function computing the magnitude use two DSP blocks, 600 FF and 1255 LUTs. The Compute θ function use one DSP blocks, 1548 FF and 3486 LUTs. It is these two functions that use the most resources as well, together with the window buffering, which utilizes two BRAM. The read in and read out functions use the least amount of resources, 24/43 FF and 86/77 LUTs, respectively. These functions only provide the pixel loop that iteratively fetch pixels from memory and passes it to the stream. With the difference that the read out function writes the transformed samples back into a memory mapped location. Computing the derivatives use 183/182 FF and 307/298 LUTs for the x and y derivatives, respectively. No DSP blocks are needed and the iteration latency is four cycles.

Table 4.1: Latency and resource estimates for the reference system.

Modules & Functions	TP (FPS)	IL (Cycles)	BRAM (Amount)	DSP (Amount)	FF (Amount)	LUT (Amount)
Top function	97.532		2	3	3683	6443
Read in		2	-	-	24	86
Update window		1285	2	-	169	241
Compute D_x		4	-	-	183	307
Compute D_y		4	-	-	182	298
Compute **G**		15	-	2	600	1255
Compute θ		17	-	1	1548	3486
Read out 1		2	-	-	43	77
Read out 2		2	-	-	43	77

4.3 Single technique approaches

The single technique approaches are examined in detail. Both the changes in the sub-functions and the resulting intermediary images are presented when required.

4.3.1 Precision scaling techniques

Figure 4.8 show the detection performance when using precision scaling for the input pixel depth. The expected outcome would be that the performance deteriorate step by step as the precision is lowered, as seen in [32], [11] and [29]. And, that the magnitude of error increases as the relative size of the truncation removes a larger, and larger portion of the data. At least, that is the expected outcome when the performance is reviewed at the same stack level. This is not the case when the performance is examined higher in the system hierarchy.

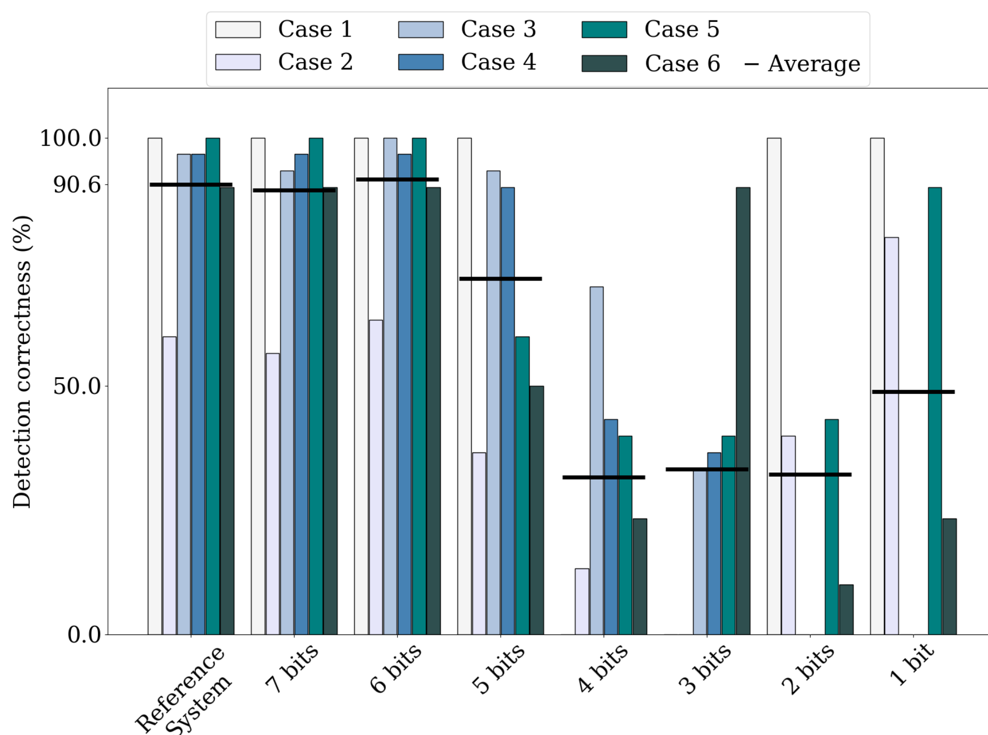


Figure 4.8: Detection correctness when using precision scaling for the input pixel depth. The reference system uses 8 bits for the input pixel depth.

The system performance is generally lower further to the right in the plot. Yet, on a case by case basis the results from the previous higher precision approach does not give a good indication of the current performance. Nor

does the deterioration of the system performance follow any apparent function.

Increased performance is seen for the 6 bit approach. It obtains equal or better performance than the reference system across all cases. The most surprising results is for the single bit approach. Which show a higher detection correctness for the second case than the reference system, and the 6 bit approach.

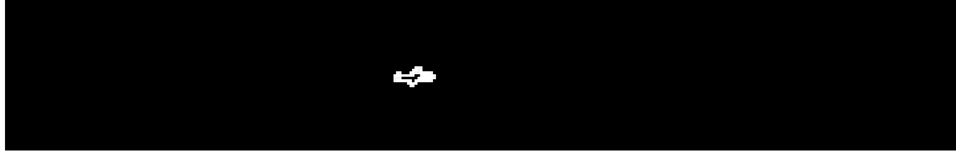
A reason behind the varying performance is that whenever a bit is truncated from the pixel data, the difference in intensity can both increase and decrease. What determines this is the bit representation of the numbers. For an example, two numbers and the difference between them: the first number being 255 with a logarithmic base 2 representation of 11111111_2. Take the second number to be $128 = 10000000_2$. Initially the difference is $127 = 1111111_2$. At each step the LSB is truncated and the result is scaled back to the original 8 bit representation. For the first truncation the difference becomes $126 = 1111110_2$ after scaling. Then the difference becomes $124 = 1111100_2$, then $120 = 1111000_2$, and for the last truncation, leaving only the MSB, the difference becomes 0. The opposite it possible as well. Take the numbers $128 = 10000000_2$ and $127 = 01111111_2$. The difference between them is initially one. As the LSB are truncated the difference increases, until only one bit is left. Scaled back to the 8 bits representation, the difference becomes $00000001_2 << 7 = 10000000_2 = 128$.

Precision scaling before computing the derivatives can therefore have a large impact on the edges produced. If this reduces the noise and highlight the outline of the objects, or if it highlights the noise and hides the object, is not given.

As for the second case with an input bit width of one, the performance indicate that the truncation is favorable. This can be verified by examining the sampling images produced by the testbench. Figure 4.9 show the gradient image for the 1 bit approach when the AV overlaps the edge of the cloud. All the noise is gone and only the outline of the AV remains, making the detection and ranking a trivial matter. This is also what happens for the fifth case, and to an extent for the sixth case using 3 bits. For the 3 bit approach the foreground noise is not gone. Yet a crisp outline of the AV is produced and all background noise has been removed, as can be seen from Figure 4.10.

The resource usage for at the input precision scaling approaches and the

(a) Reference system.



(b) Precision scaling 1 bit.

Figure 4.9: Gradient magnitude image of the second detection case. Precision scaling to 1 bit removes the noise completely.

affected sub-modules is shown in Table 4.2. Here the reduced resource usage follows the reduction in precision. Which is the expected outcome. Certain discrepancies are found, such as the angle computation using more resources for the 2 bit approach than the 3 bit approach. However, the exact selection of resource primitives made by the HLS tool is not examined in detail here. The primary resource saving is that all approaches use one less DSP block for the Compute **G** function. Going from five to four bits also reduces the system latency, as the two convolution operations use one less clock cycle.

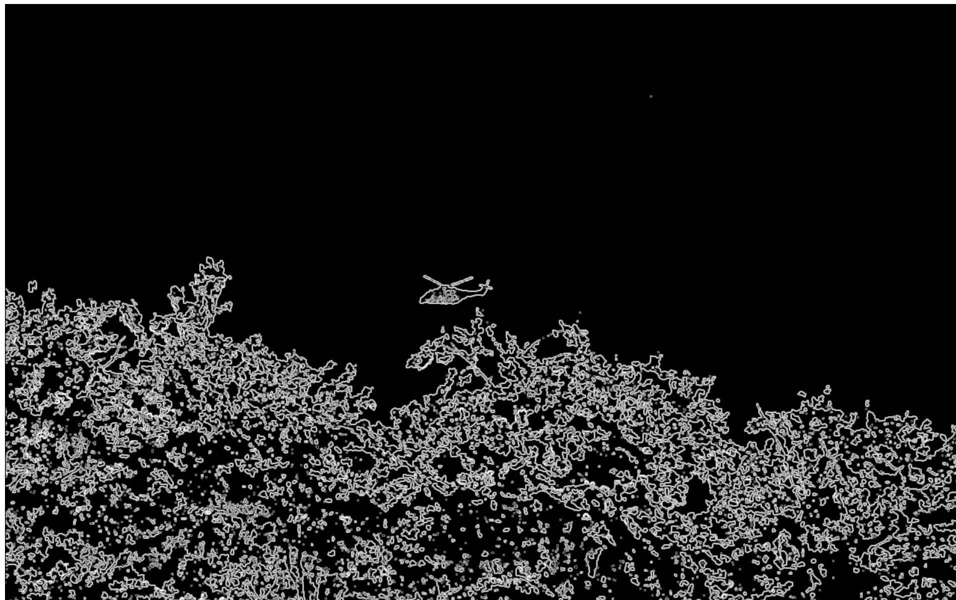


Figure 4.10: Gradient magnitude image for detection case six with a 3 bit input.

Table 4.2: Latency and resource estimates for precision scaling of the input pixel depth.

Modules & Functions	TP (FPS)	IL (Cycles)	BRAM (Amount)	DSP (Amount)	FF (Amount)	LUT (Amount)
Reference system						
8 bits	97.532		2	3	3683	6443
7 bits	97.532		2	2	3657	6485
Update window		1285	2	-	160	241
Compute D_x		4	-	-	176	302
Compute D_y		4	-	-	175	293
Compute **G**		14	-	1	599	1315
Compute θ		17	-	1	1546	3478
6 bits	97.532		2	2	3574	6437
Update window		1285	2	-	151	241
Compute D_x		4	-	-	169	297
Compute D_y		4	-	-	168	288
Compute **G**		14	-	1	589	1299
Compute θ		17	-	1	1496	3456
5 bits	97.532		2	2	3582	6433
Update window		1285	2	-	142	241
Compute D_x		4	-	-	162	292
Compute D_y		4	-	-	161	283
Compute **G**		14	-	1	579	1287
Compute θ		17	-	1	1537	3474
4 bits	97.532		2	2	3336	6314
Update window		1285	2	-	133	241
Compute D_x		3	-	-	77	266
Compute D_y		3	-	-	76	257
Compute **G**		14	-	1	569	1273
Compute θ		17	-	1	1480	3421
3 bits	97.532		2	2	3295	6286
Update window		1285	2	-	124	241
Compute D_x		3	-	-	73	263
Compute D_y		3	-	-	72	254
Compute **G**		14	-	1	559	1266
Compute θ		17	-	1	1466	3406
2 bits	97.532		2	2	3321	6283
Update window		1285	2	-	115	241
Compute D_x		3	-	-	69	260
Compute D_y		3	-	-	68	251
Compute **G**		14	-	1	549	1256
Compute θ		17	-	1	1519	3419
1 bit	97.532		2	2	3286	6242
Update window		1285	2	-	106	241
Compute D_x		3	-	-	64	252
Compute D_y		3	-	-	63	240
Compute **G**		14	-	1	539	1249
Compute θ		17	-	1	1513	3404

Figure 4.11 show the system performance for the approaches that use precision scaling for the intermediary derivative output. Compared to precision scaling of the input, this method show consistency as the precision is lowered. Going from the reference system, which uses 11 bits, the performance gradually increases until it reaches 9 bits. At 9 bits the overall system performance is at 92.78%, being 2.22% higher than the reference system. After this the performance start decreasing, and the performance falls drastically when going from 7 to 6 bits.

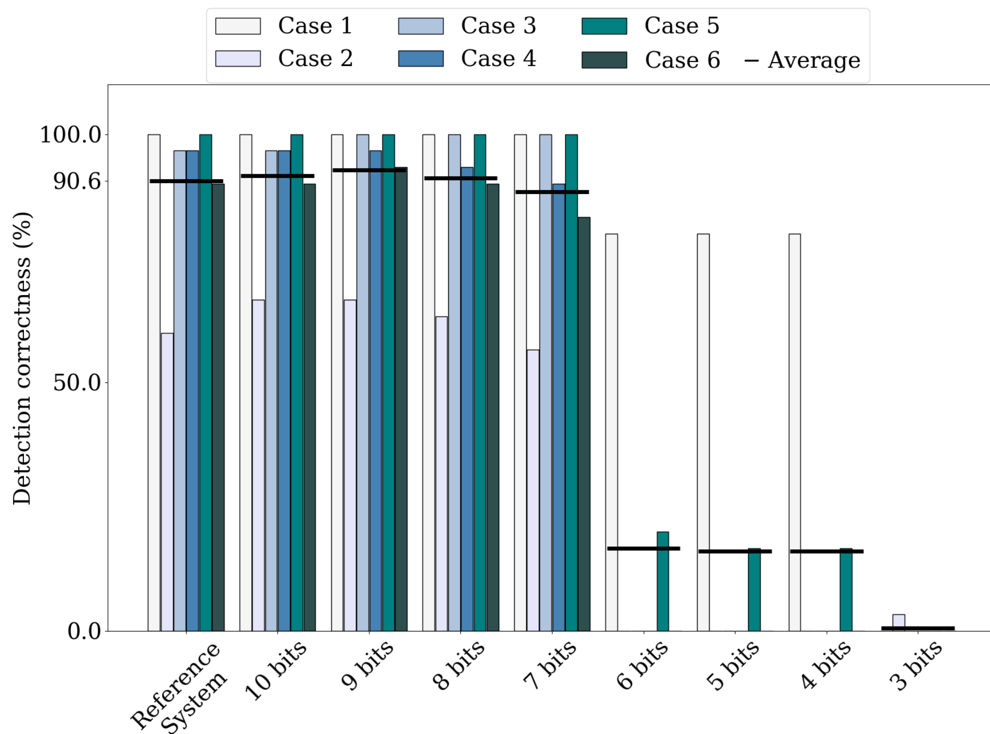


Figure 4.11: Precision scaling of the intermediary derivative output. The reference system uses 11 bits to represent the derivatives.

The 8 bit approach achieves greater overall system performance, 91.11% detections compared to the reference system at 90.56%. However, as opposed to the two previous truncations, the the case by case performance is not greater than or equal to the reference system. For the fourth case the performance drops to 90% correct detections, whereas the reference system achieves 96.7%.

The resource usage for at the precision scaling of the intermediary derivative output and the affected sub-modules is shown in Table 4.3. Once again the reduction in precision follows the reduced resource usage. Precision scaling of the intermediary derivative results frees a DSP block

in the Compute **G** function. This happens already at the first truncation. The iteration latency is also lowered for the Compute **G** function, going from 15 cycles to 14 cycles. The Update θ function uses more resources for the 3, 4 and 5 bit approach than the 6 bit approach. From 6 to 5 bit in the intermediary output corresponds to going from 3 to 2 bit at the input. As was seen a prior the same increase in resource usage happened then. This indicate that the HLS tool change the CORDIC algorithm implementation when going from a six to five bit input.

Table 4.3: Latency and resource estimates for precision scaling of the derivative intermediary output.

Modules & Functions	TP (FPS)	IL (Cycles)	BRAM (Amount)	DSP (Amount)	FF (Amount)	LUT (Amount)
Reference system						
11 bits	97.532		2	3	3683	6443
Compute **G**		15	-	2	600	1255
Compute θ		17	-	1	1548	3486
10 bits	97.532		2	2	3680	6495
Compute **G**		14	-	1	599	1315
Compute θ		17	-	1	1546	3478
9 bits	97.532		2	2	3620	6457
Compute **G**		14	-	1	589	1299
Compute θ		17	-	1	1496	3456
8 bits	97.532		2	2	3651	6463
Compute **G**		14	-	1	579	1287
Compute θ		17	-	1	1537	3474
7 bits	97.532		2	2	3584	6396
Compute **G**		14	-	1	569	1273
Compute θ		17	-	1	1480	3421
6 bits	97.532		2	2	3560	6374
Compute **G**		14	-	1	559	1266
Compute θ		17	-	1	1466	3406
5 bit	97.532		2	2	3603	6377
Compute **G**		14	-	1	549	1256
Compute θ		17	-	1	1519	3419
4 bits	97.532		2	2	3587	6355
Compute **G**		14	-	1	539	1249
Compute θ		17	-	1	1513	3404
3 bits	97.532		2	2	3571	6330
Compute **G**		14	-	1	529	1241
Compute θ		17	-	1	1507	3387

4.3.2 Iterative techniques

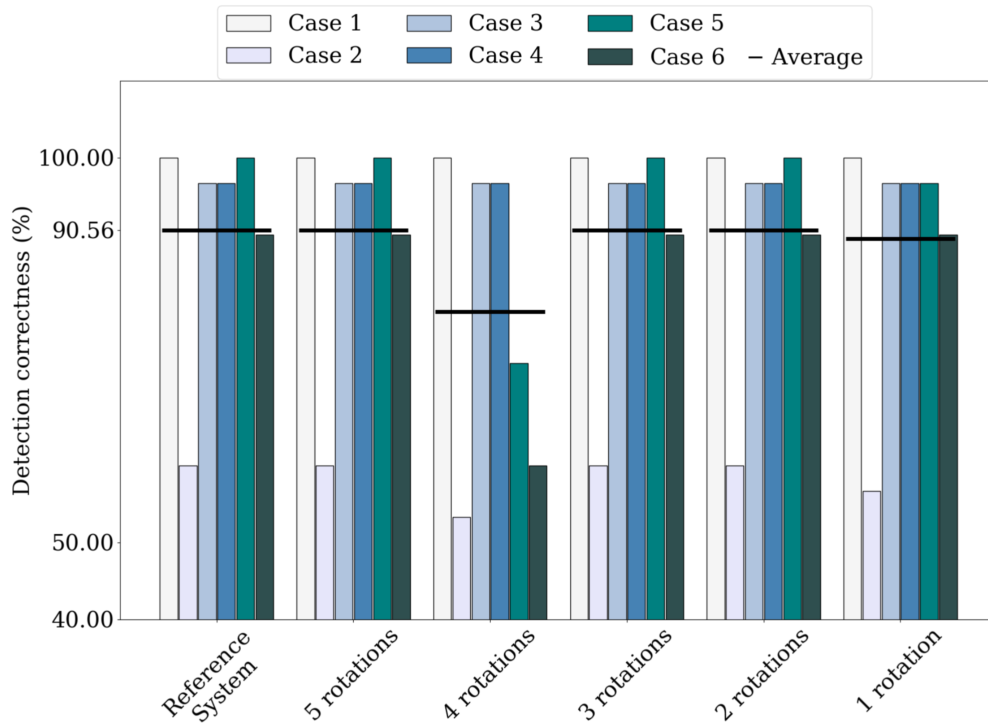


Figure 4.12: Detection correctness for the CORDIC algorithm with reduced rotation steps. The reference system uses nine rotation steps.

Table 4.4 show the latency and resource estimates for the CORDIC algorithm at varying iteration depth. With two rotation steps the required number of LUTs is effectively halved, using only 55% of the reference system, the required amount of flip-flops being 61% and the BRAM and DSP remaining unchanged. The iteration latency also decreases as the iteration count is lowered, which is to be expected. The delay is reduced from 17 to eight cycles for both the single rotation and two rotation approach.

Using two rotation steps does not effect the overall detection performance, as can be seen from Figure 4.12. This being the case with three and five rotations as well. However, at four rotations the detection performance drops significantly to 80%. Thus being outperformed at all other iteration steps, even those with a lower rotation count. This seems counter intuitive at first. However, the CORDIC algorithm has some inherent properties that are uncovered when using four rotations.

The five first steps of CORDIC algorithm are : 45°, 26.56°, 14.03°, 7.12°, 3.57°. At four rotations the final angle is within the range from −2.71° to

92.71° at steps of 7.12°. The representation of angles is used to determine what pixel to walk to next. For the detection algorithm the direction can be divided into 16 distinct states. Either the direction points exactly to one of the eight neighbouring pixels or in between the two. At four rotations when the correct angle is close to 0° or 90° the algorithm will overstep the bounds of the quadrant. Which in turn gives a possible error magnitude of two steps for the pixel walk. At a higher rotation level it corrects itself, and at a lower it is not possible to overstep the edge of the quadrant.

Table 4.4: Latency and resource estimates for the CORDIC algorithm with reduced rotation steps.

Modules & Functions	TP (FPS)	IL (Cycles)	BRAM (Amount)	DSP (Amount)	FF (Amount)	LUT (Amount)
Reference system						
9 rotations	97.532		2	3	3683	6443
Compute θ		17	-	1	1548	3486
5 rotations	97.532		2	3	3024	4726
Compute θ		12	-	1	889	1769
4 rotations	97.532		2	3	2835	4354
Compute θ		10	-	1	700	1397
3 rotations	97.532		2	3	2594	3943
Compute θ		9	-	1	459	986
2 rotations	97.532		2	3	2262	3564
Compute θ		8	-	1	127	607
1 rotation	97.532		2	3	2217	3406
Compute θ		8	-	1	82	449

4.3.3 Memory techniques

The resource usage for the two LUT based techniques are presented in Table 4.5. Using lookup in memory instead of computing the values at runtime reduces the iteration latency for the Compute **G** function significantly, going from 17 cycles to four. The larger 128 by 128 LUT uses eleven BRAM, whereas the modified 30 by 30 LUT can be implemented in the logic and use no BRAM. Both approaches remove the need for DSP blocks for the Compute **G** function.

Table 4.5: Latency and resource estimates for the LUT approaches.

Modules & Functions	TP (FPS)	IL (Cycles)	BRAM (Amount)	DSP (Amount)	FF (Amount)	LUT (Amount)
Reference system	97.532		2	3	3683	6443
Compute **G**		15	-	2	600	1255
128x128 LUT	97.532		13	1	3123	5321
Compute **G**		4	11	-	42	139
30x30 LUT – 1 bit	97.532		2	1	3136	5378
Compute **G**		4	-	-	55	196

The performance for the LUT based approaches are found in Figure 4.13. The 128 by 128 LUT has decreased overall performance, with a correct detection rate of 86.67%. The 30 by 30 LUT show the same detection performance as the reference system, at 90.56% correct detections. That the 128 by 128 LUT produce less accurate results than the 30 by 30 LUT is as expected. The smaller LUT has added functionality and uses the exact value representation that corresponds to the detection algorithm. Whereas the larger LUT provide correct values only at certain steps. That being when the input exactly match the discretization level. No interpolation scheme is implemented to mitigate this margin of error either.

4.3.4 Error techniques

The system performance for the error inducing techniques are presented in Figure 4.13. Loop perforation reduces the system performance for both the un-scaled and scaled approach, with a performance of 87.78% and 64.44% respectively. The Kernel reduction approaches show some interesting behavior. The scaled approach outperforms the reference system across all cases and make correct detections 96.67% of the time. The system performance for the un-scaled approach is 83.35%, yet for cases 2 and 3 the kernel reduction obtains higher detection correctness than the reference system.

Examining the intermediary images give some indication as to why the scaled approach outperforms the un-scaled kernel reduction approach. Figure 4.14 show the gradient magnitude images for case 4 for both approaches. The outline of the un-scaled approach is too thin to pass the kernel that finds suitable seed points. It needs an edge piece with at least 3 by 3 valid pixels.

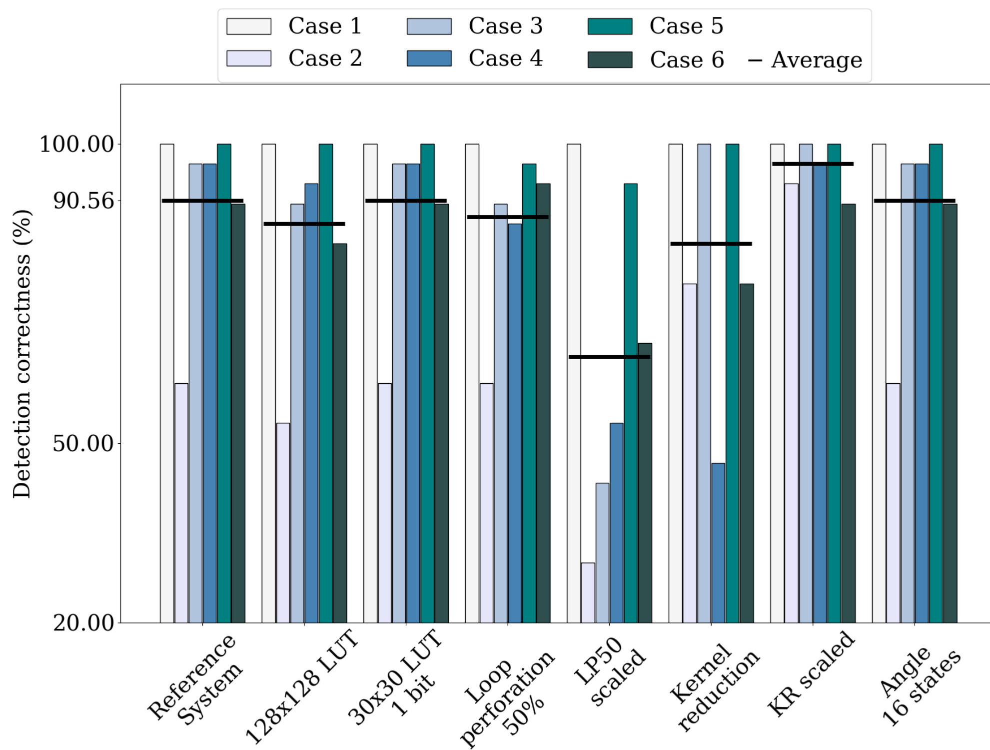
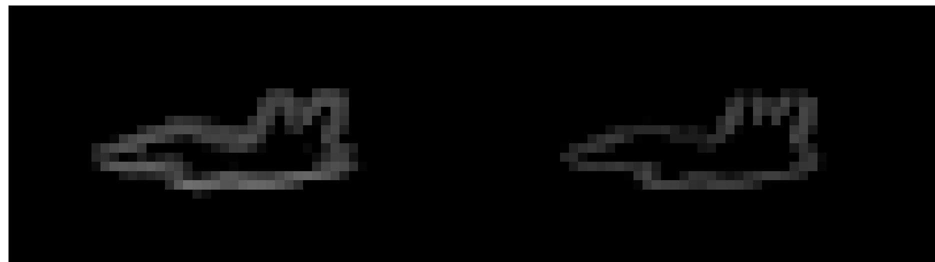


Figure 4.13: Detection performance for the memory and error inducing techniques.



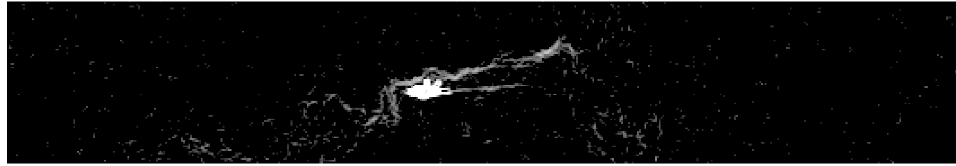
(a) Gradient magnitude scaled approach.

(b) Gradient magnitude un-scaled approach.

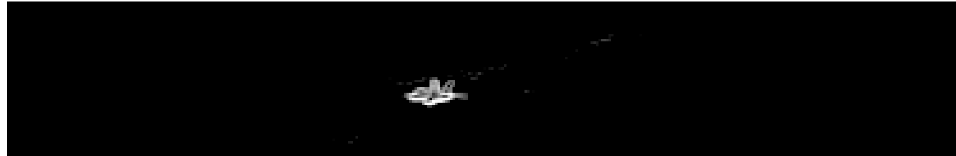
Figure 4.14: The gradient magnitude images for case 4 using scaled and un-scaled kernel reduction. The cropped image content show that the object outline becomes too thin if the y derivative is not scaled. The scaled approach produces a correct detection, while the un-scaled approach fails to extract the shape.

The kernel reduction approaches perform well for the second case. The gradient magnitude image is shown in Figure 4.15. The noise and undesired edges produced by the clouds are gone leaving only the outline of the AV. The image frame is the same as presented earlier, where the plane

overlaps the edge of the cloud. This did not produce a valid detection for the reference system, yet it does here.



(a) Reference system.



(b) Scaled kernel reduction.

Figure 4.15: Gradient magnitude image for the second detection case. The noise is gone when using the scaled kernel reduction technique.

The reduced angle representation to 16 states perform as expected. It does not differ from the reference system.

The latency and resource estimates for the approaches are presented in in Table 4.6. Loop perforation increases the throughput significantly. Going from 97.532 FPS to 195.059 FPS. The iteration latency for the Update window function is also halved, going from 1285 cycles to 645 cycles. The approaches does not incur any significant resource savings. However, there are some routing changes and the approaches use FF 16 less and 17 LUTs more than the reference system.

The kernel reduction approaches greatly improves the iteration latency for the Update window function, going from 1285 to 5 cycles. It completely removes the buffering delay, except for storing one previous sample. These approaches also halves the BRAM usage. Which is an expensive resource. The FF requirement in the Update window function is also reduced to 124, as opposed to 169 FF for the reference system.

The angle to 16 state representation reduces the resource requirement and iteration latency for the Compute θ function. The approach use 57 FF and 318 LUTs for the function and removes the need for a DSP block. The iteration latency is only three cycles.

Table 4.6: Latency and resource estimates for the error inducing techniques.

Modules & Functions	TP (FPS)	IL (Cycles)	BRAM (Amount)	DSP (Amount)	FF (Amount)	LUT (Amount)
Reference system	97.532		2	3	3683	6443
Update window		1285	2	-	169	241
Compute D_x		4	-	-	183	307
Compute D_y		4	-	-	182	298
Compute **G**		15	-	2	600	1255
Compute θ		17	-	1	1548	3486
Read out 1		2	-	-	43	77
Read out 2		2	-	-	43	77
Loop perforation 50% & Loop Perforation 50% scaled	195.059		2	3	3667	6457
Update window		645	2	-	165	237
Compute D_x		4	-	-	181	303
Compute D_y		4	-	-	180	294
Compute **G**		15	-	2	599	1253
Compute θ		17	-	1	1545	3478
Read out 1		2	-	-	41	95
Read out 2		2	-	-	41	95
Kernel reduction	97.654		1	3	3636	6436
Update window		5	1	-	124	240
Compute D_x		4	-	-	183	307
Compute D_y		4	-	-	182	298
Compute **G**		15	-	2	600	1255
Compute θ		17	-	1	1546	3480
Kernel reduction scaled	97.654		1	3	3626	6435
Update window		5	1	-	124	240
Compute D_x		4	-	-	183	307
Compute D_y		4	-	-	172	297
Compute **G**		15	-	2	600	1255
Compute θ		17	-	1	1546	3480
Angle 16 states	97.532		2	2	2192	3275
Compute θ		3	-	-	57	318

4.4 Combined techniques

For the combined techniques the resource savings is not examined in detail. The approaches incur savings throughout the system and consistently affect several of the sub-functions. The approaches are combinations of techniques that are already examined in detail. The reduction in resources follows the reduction found for the single technique approaches to a high degree, and accumulates as they are combined. However, there are some reduction in resources that can not be traced back directly.

Another aspect that must be taken into consideration is the effect on the iteration latency, and thereby the throughput. Techniques that reduce the iteration latency for one sub-function, only propagate through the system and increase the throughput if the function incur the highest delay. The Compute **G** and Compute θ functions run in parallel. The techniques that reduce the iteration latency within either, without affecting the other, do not necessarily see a change in the throughput. When a combination of techniques reduce the delay in both, the throughput increases more than what can be seen from the accumulated gain.

The focus in this section is on the system performance and how combing techniques affect it. The combined techniques amount to 21 distinct approaches. An overview of the combined techniques is found in Table 4.7. The approaches are named after the techniques they utilize. The full overview of the saved resources can be found in the last section of this chapter in Table 4.9.

4.4.1 Precision scaling and iterative techniques

Looking at the system performance there are some things to keep in mind, such as: Is the performance decided by the lowest denominator? Or, is there an accumulation of error? And, if so, how does the error accumulate?

To explore this five approaches are tested that combine the CORDIC reduced rotation and precision scaling of the input. Both techniques can scale the approximation level. The plot for the detection correctness is seen in Figure 4.16. The approach that use the CORDIC at two or more rotations are only affected by the performance change given by the precision scaling. However, for the approaches using the single rotation CORDIC the combined performance changes. The combined outcome is more than the accumulated error added and more than lowest denominator.

Table 4.7: Overview of combined techniques

Name	Description
In7 · C3	Precision scaling of the input to 7 bits and CORDIC with 3 rotations.
In7 · C2	Precision scaling of the input to 7 bits and CORDIC with 2 rotations.
In7 · C1	Precision scaling of the input to 7 bits and CORDIC with 1 rotation.
In6 · C2	Precision scaling of the input to 6 bits and CORDIC with 2 rotations.
In6 · C1	Precision scaling of the input to 6 bits and CORDIC with 1 rotation.
In6 · Im7	Precision scaling of the input to 6 bits and precision scaling of the intermediary derivative to 7 bits.
GL30 · C2	Gradient LUT 30 by 30 and CORDIC with 2 rotations.
Im9 · GL30 · C2	precision scaling of the intermediary derivative to 9 bits, gradient LUT 30 by 30 and CORDIC with 2 rotations.
Im9 · GL30 · Ang16	precision scaling of the intermediary derivative to 9 bits, gradient LUT 30 by 30 and the angle representation to 16 states.
GL30 · KRS · Ang16	Gradient LUT 30 by 30, kernel reduction scaled and the angle representation to 16 states.
GL30 · C2 · LP50	Gradient LUT 30 by 30, CORDIC with 2 rotations and loop perforation by 50%.
GL30 · C2 · LP50 · KR	Gradient LUT 30 by 30, CORDIC with 2 rotations, loop perforation by 50% and kernel reduction.
Im9 · GL30 · C2 · KR	Precision scaling of the intermediary derivative to 9 bits, gradient LUT 30 by 30, CORDIC with 2 rotations and kernel reduction.
GL30 · C2 · LP50S · KRS	Gradient LUT 30 by 30, CORDIC with 2 rotations, loop perforation by 50% scaled and kernel reduction scaled.
C2 · LP50S · KRS	CORDIC with 2 rotations, loop perforation by 50% scaled and kernel reduction scaled.
GL30 · LP50 · KRS · Ang16	Gradient LUT 30 by 30, loop perforation by 50%, kernel reduction scaled and the angle representation to 16 states.
Im9 · C2 · LP50	Precision scaling of the intermediary derivative to 9 bits, CORDIC 2 rotations and loop perforation %.
Im9 · GL30 · C2 · LP50	Precision scaling of the intermediary derivative to 9 bits, gradient LUT 30 by 30, CORDIC 2 rotations and loop perforation 50%.
Im9 · GL30 · C2 · LP50 · KR	Precision scaling of the intermediary derivative to 9 bits, gradient LUT 30 by 30, CORDIC 2 rotations, loop perforation 50% and kernel reduction.
Im9 · GL30 · KRS · Ang16	Precision scaling of the intermediary derivative to 9 bits, gradient LUT 30 by 30, kernel reduction scaled and the angle representation to 16 states.
Im9 · GL30 · LP50 · KRS · Ang16	Precision scaling of the intermediary derivative to 9 bits, gradient LUT 30 by 30, loop perforation 50% scaled, kernel reduction scaled and the angle representation to 16 states.

Both the precision scaling to 7 bits and CORDIC using one rotation reduce the performance by $90.56 - 89.45 = 1,11\%$. The combined approach obtains a performance of 87.78%. Which differs from the reduction the accumulated error gives at 88.34%. For the 6 bit input the performance increases. The combined performance with the CORDIDC at one rotation is at 88.88% correct detections. Again, this is neither determined by the lowest denominator nor the accumulated value.

4.4.2 Performance characteristics of combined techniques

It is clear that the performance characteristic affect how the combined approaches interact. Based on observations from the single technique approaches the performance characteristics fall into five distinct groups:

1. Techniques that reduce the overall performance, thus making the system approximate. Where the case by case performance is either equal or lower compared to the reference system. The techniques or combination of techniques that fall into this group are fully approximate. Techniques with this performance characteristics are referred to as **approximate**.
2. Techniques that reduce the overall system performance, but show higher performance than the reference system for one or more detection case. Techniques that show this performance characteristic are referred to as **semi-approximate**.
3. Techniques that do not impact the detection performance. Meaning that the system has still not become approximate. Thereby operating with **redundancy**.
4. Techniques with increased overall performance. Where the case by case performance need not be greater than or equal to the reference system. The only criteria is that the average performance is higher. These techniques have an **enhanced** performance characteristic.
5. Techniques where the overall system performance has increased and the performance for all cases are equal to or greater than the reference system. A Pareto optimal system performance is achieved compared to the reference system. Techniques with this performance characteristic are referred to as having a **Pareto optimal** characteristic.

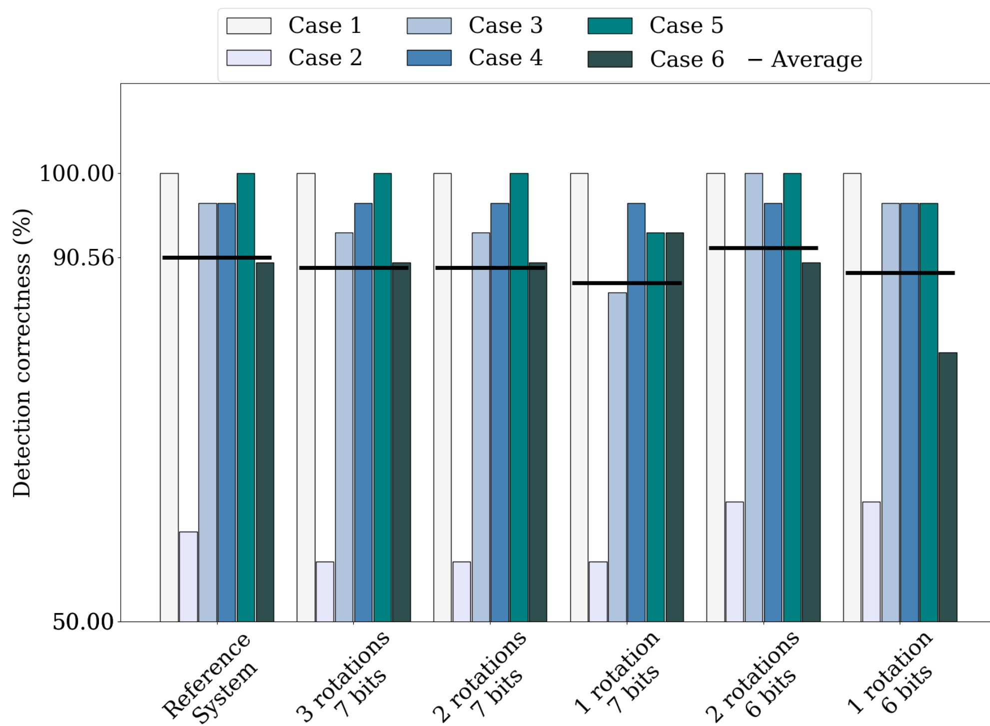


Figure 4.16: CORDIC algorithm with reduced rotation steps and precision scaling of the input pixel depth.

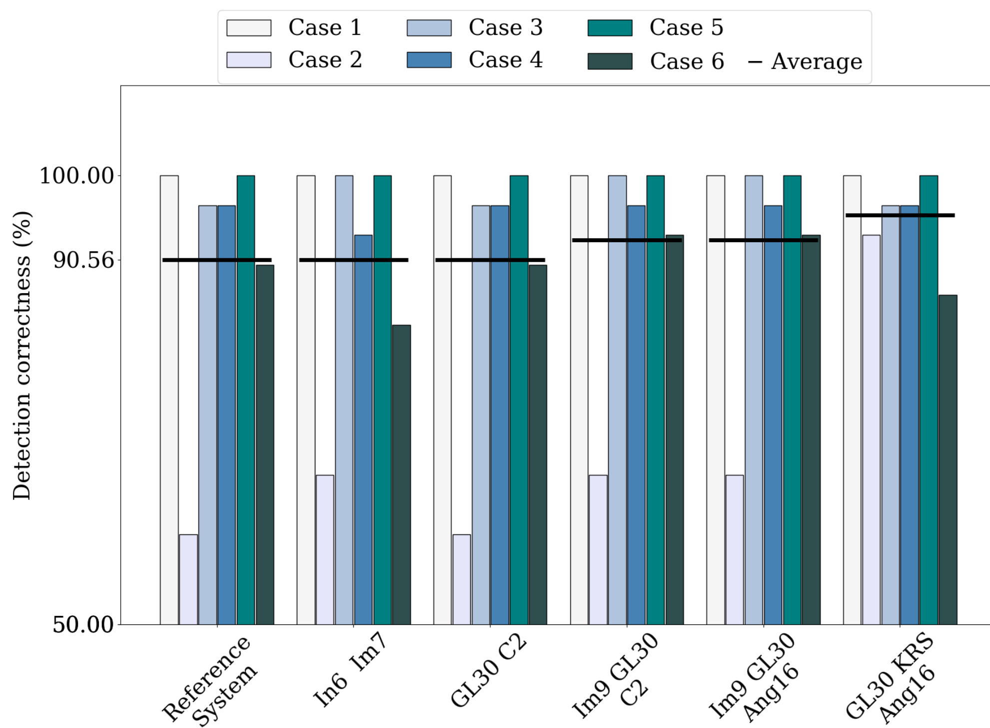


Figure 4.17: Combined techniques with high overall system performance. The average detection correctness being equal or greater than the reference system.

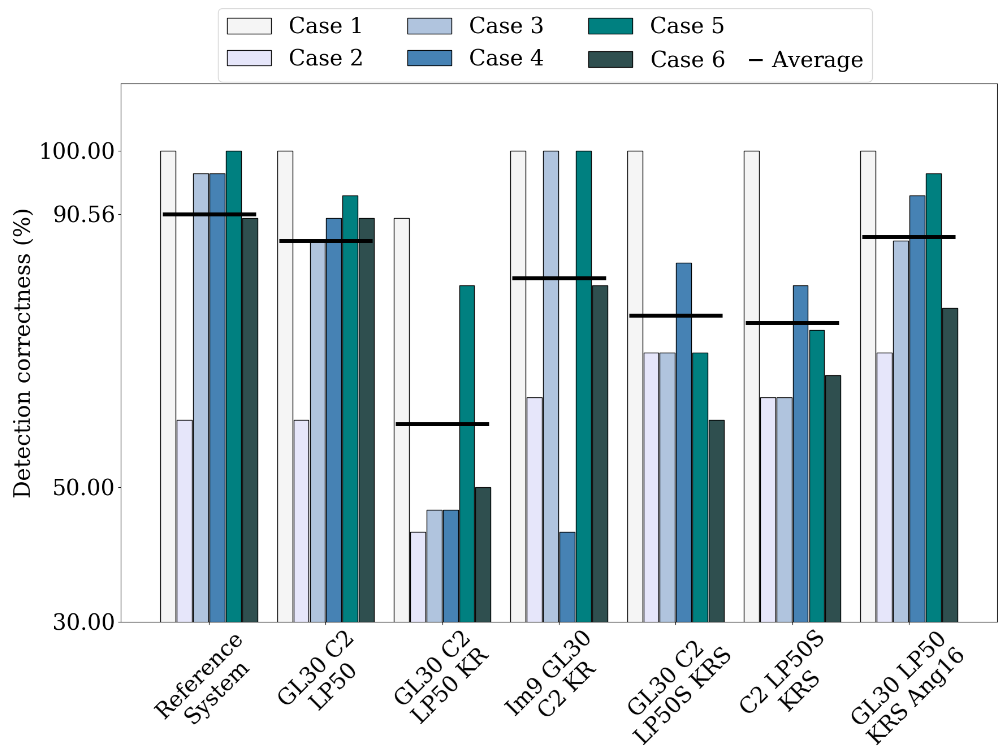


Figure 4.18: Combined techniques with reduced overall system performance. The techniques are either semi-approximate or approximate.

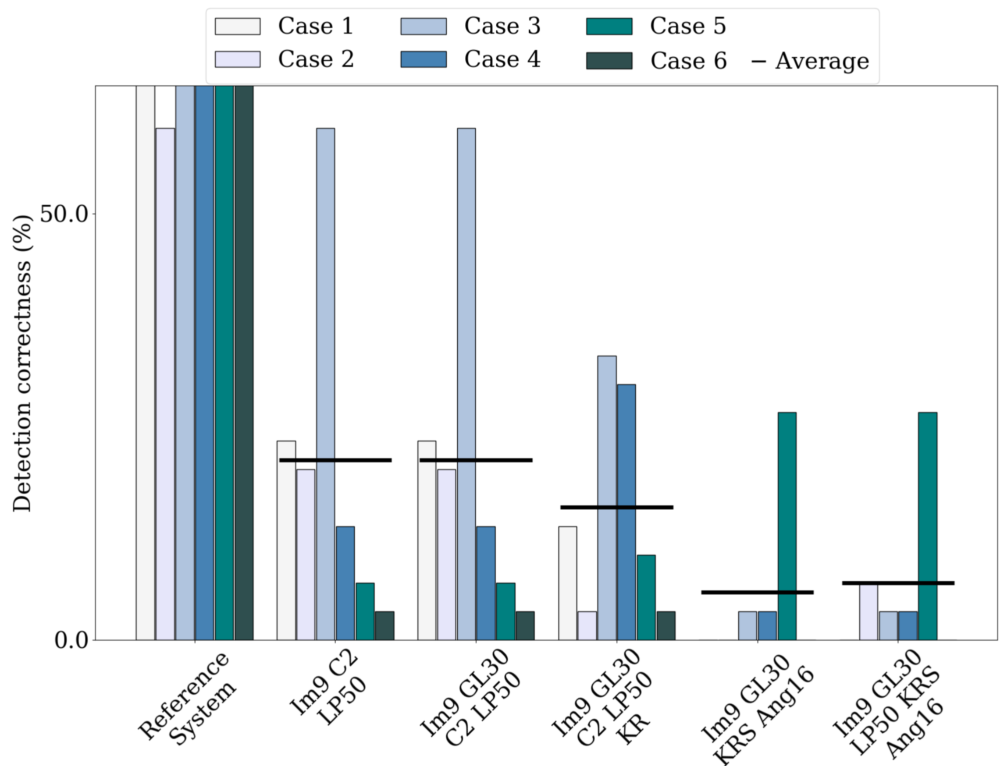


Figure 4.19: Combined techniques with greatly reduced system performance. All techniques are approximate. The range of the y-axis is set from 0 to 65 %.

Figure 4.17 show approaches that perform better or equal to the reference system. Thus having Enhanced, Pareto optimal or a Redundancy performance characteristic. Approaches with approximate or semi-approximate characteristics are shown in Figure 4.18. In Figure 4.19 approaches with significant reduction in system performance are shown. Where all techniques have an approximate performance characteristic.

The precision scaling techniques showed the largest increase in performance for the input at 6 bits and the derivatives at 9 bits. The precision scheme of the system expands the derivatives by three bits. Scaling the input by two bits down to 6 bits, scales the derivative output down to 9 bits. However, there is no loss of information when finding the derivative values, the width is simply determined by the necessary precision scheme. Is the performance increase caused by the 9 bit representation or is this a property when reducing the derivative representation by two bits?

To test this a two bit reduction is used on both the input and the derivative. Meaning that the derivative output is set to 7 bit, i.e. two bits lower than the required precision scheme. The approach (In6 · Im7 Figure 4.17) obtains a system performance of 90.56%. It seems that the positive effect of the input precision scaling is mitigated by the precision scaling of the derivative. This points towards a 9 bit representation being favorable.

Combining techniques with a redundancy characteristic did not impact the performance. As was seen when combining the reduced CORDIC rotations and precision scaling. Does this extend to an approach that uses two or more techniques with a redundancy characteristics?

A combined approach using the gradient LUT 30 by 30 and the CORDIC at two rotations (GL30 · C2) is shown in Figure 4.17. Both techniques have a redundancy performance characteristic. It is observed that the combined performance has a redundancy characteristic as well. When expanding the approach to use precision scaling of the input (Im9 · GL30 · C2 Figure 4.17), the performance becomes the same as for the precision scaling approach alone. Replacing the CORDIC algorithm with the angle to 16 states does not affect the system performance either (Im9 · GL30 · Ang16 Figure 4.17).

When this approach is expanded to also include the kernel reduction (Im9 · GL30 · C2 · KR Figure 4.18) the performance drops to 81.11%. For the scaled kernel reduction (Im9 · GL30 · KRS · Ang16 Figure 4.19) it deteriorates even more. The system performance becomes 5.57%. This is

also the case when the approach is expanded to use the loop perforation (Im9 · GL30 · C2 · LP50 Figure 4.19). Which obtains a performance of 21.11%.

4.4.3 Unaccounted resource savings

There are four approaches that completely removes the need for BRAM components. The line buffer is instead implemented using FF and LUTs. The approaches and the resulting resource usage of FF and LUTs is given in Table 4.8.

Table 4.8: Resource estimates for techniques without BRAM

Techniques	FF	LUT
Reference system	3683	6443
GL30 · C2 · LP50 · KR	7130	5668
GL30 · C2 · LP50S · KRS	7120	5667
C2 · LP50S · KRS	7665	6726
Im9 · GL30 · C2 · LP50 · KR	7034	5594

4.4.4 Observations on generality

There are some overarching trends. One is that no enhanced or Pareto optimal characteristic is obtained when loop perforation is involved. Another is that loop perforation combined with precision scaling severely deteriorates the performance.

Techniques that by themselves had no loss in accuracy, being in the enhanced or Pareto optimal performance characteristic groups, become approximate when combined. As is the case when combining kernel reduction and precision scaling. Any combination that is based on techniques with semi-approximate or approximate characteristic ends up with a semi-approximate or approximate characteristic.

To summarize this, it is observed that only techniques that affect performance change the overall performance when combined. And, that techniques with a redundancy performance characteristic do not change the overall performance when combined. This indicate that techniques with a redundancy performance characteristic can be combined freely, without worrying about the resulting performance.

4.5 Correlation

When the Sobel filer is used as a testbench for AC it is usually the intermediary results, the gradient image and direction image, that is examined. Different metrics of degradation in image quality are used to measure the accuracy. Here the results are viewed at a higher level. However, the intermediary degradation in image quality is still accounted for. This means that it the accuracy of the intermediary results can be examined in relation to the overall performance. The metric used to represent the accuracy of the intermediary results is the APD.

The relation is examined in four plots. The tendency can be verified by visual inspection by dividing the plots into four quadrants: the left-bottom, right-bottom, left-top and right-top regions. The data almost always fit into three of the quadrants, leaving one empty. Using this crude estimation the general left-right and up-down relation can be determined.

In Figure 4.20 the APD for the gradient magnitude image is plotted against the system performance. When the APD increases the likelihood of obtaining a correct detection decreases. This is as expected, and indicate that the accuracy loss for the intermediary gradient image can be used to estimate the overall accuracy. In Figure 4.21 the APD for the gradient magnitude image is plotted against the correlation to the reference system. The likelihood of a high correlation in detections is still higher when the APD is low. However, compared to the system performance the result are more spread out. And, certain outliers obfuscate the plot.

In Figure 4.22 the angle difference is plotted against the system performance. The likelihood of obtaining a correct detection is higher if the difference is low. However, there is no guarantee that the performance decrease. The quadrant with no cases is the left-bottom one. This indicate that techniques which greatly reduces the intermediary accuracy of the gradient direction still have high chance of obtaining high overall accuracy. In Figure 4.21 the angle difference is plotted against the correlation to the reference system. The plot corresponds to high degree with the plot against the system performance.

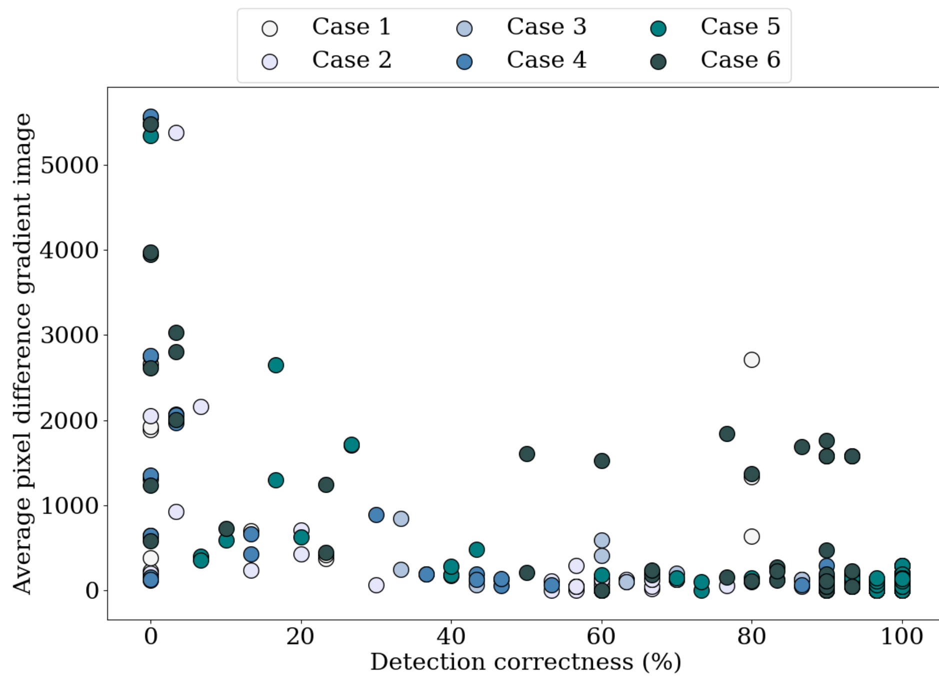


Figure 4.20: The average pixel difference for the gradient image plotted against the correct detection.

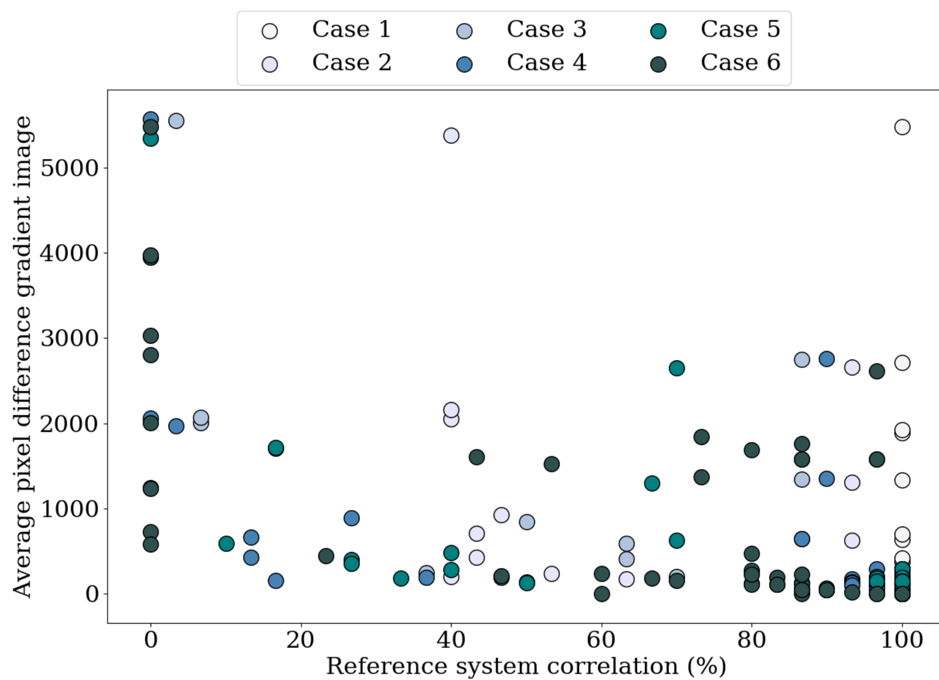


Figure 4.21: The average pixel difference for the gradient image plotted against the detection correlation to the reference system.

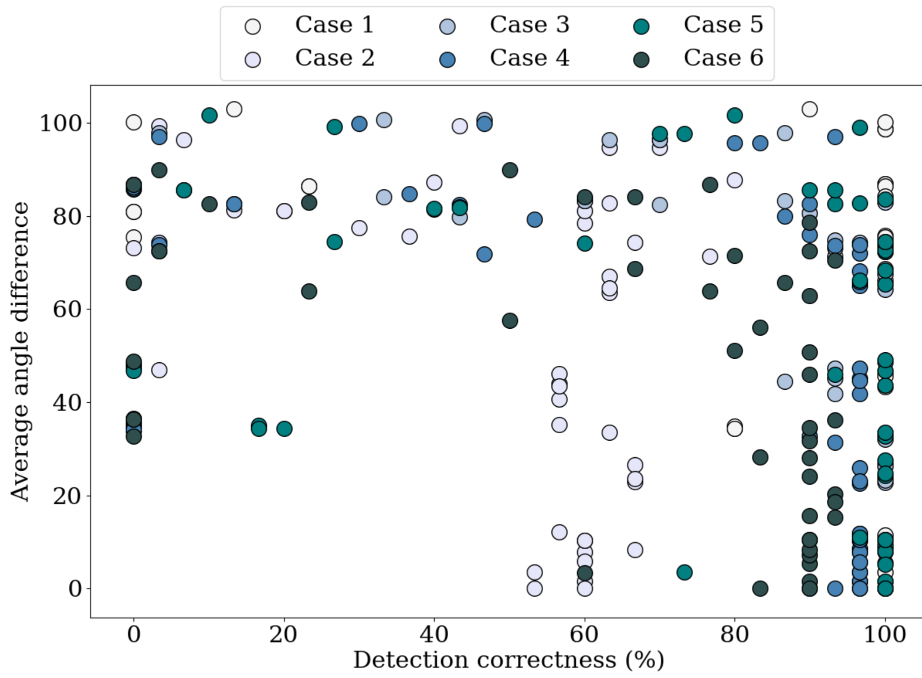


Figure 4.22: The average angle difference plotted against detection correctness.

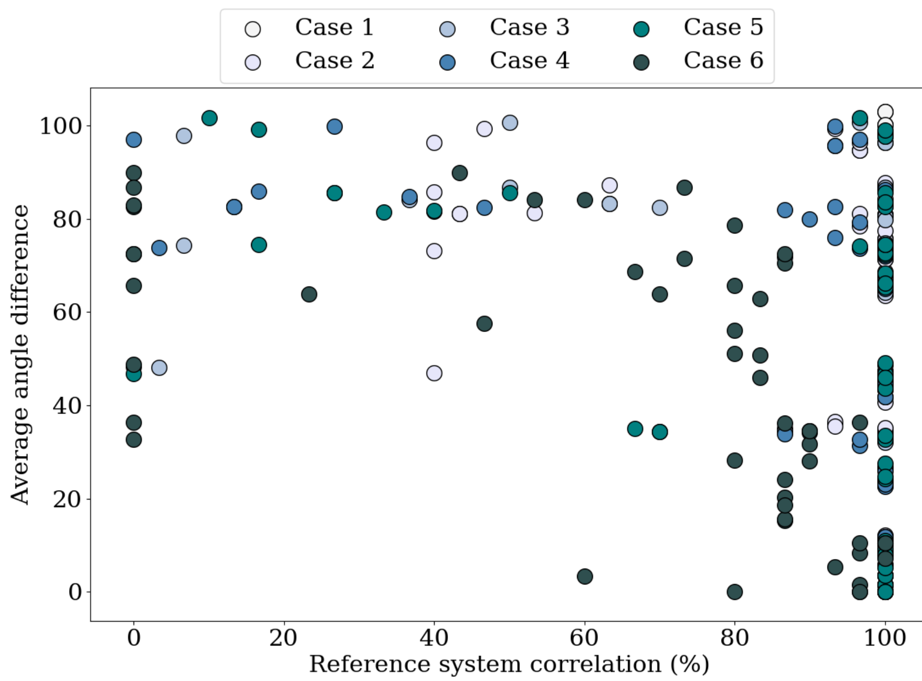


Figure 4.23: The average angle difference plotted against the detection correlation to the reference system.

A last plot is include that show the correspondence between correct detections and the correlation to the reference system. The results are shown in Figure 4.24. It is observed that a high correlation is always obtained when the overall system performance is high. However, it is possible to have an approach that obtains a high correlation that does not show a satisfactory system performance. This means that the correlation to the reference system can not be used as a viable measure for the system performance.

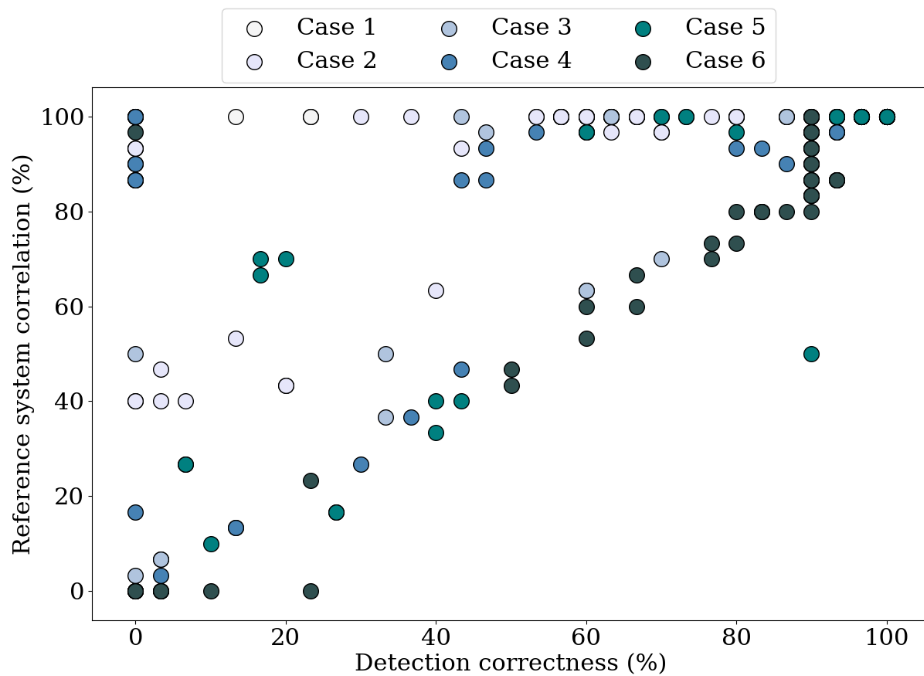


Figure 4.24: The detection correlation to the reference system plotted against the correct detection.

4.6 Comparison

A comparison of all the implemented techniques is presented in Table 4.9. The table uses the performance characteristics to differentiate the techniques, and indicate the group association by color coding of the rows. The groups are the same as presented earlier: Approximate, Semi-approximate, Redundancy, Enhanced and Pareto optimal performance characteristics.

The table is divided in five. The first four parts show the single technique approaches and divide by the class affiliation. The last part show the combined techniques. The overall system performance is included. And,

is presented as the performance gain relative to the reference system. The relative performance gain (RPG) is given by equation 4.5.

$$RPG = \begin{cases} \frac{P_{RS}-P_{AT}}{P_{RS}} \cdot 100, & \text{if } P_{RS} < P_{AT} \\ \frac{P_{AT}-P_{RS}}{P_{RS}} \cdot 100, & \text{if } P_{RS} > P_{AT} \\ 0, & \text{if } P_{RS} = P_{AT} \end{cases} \tag{4.5}$$

P_{RS} represent the system performance for the reference system.
P_{AT} represent the performance for the AC technique in question.

The throughput gain for each approach is given in frames per seconds and is found using equation 4.6. The FPS estimate was found a prior using equation 4.1.

$$\text{FPS gain} = FPS_{AT} - FPS_{RS} \tag{4.6}$$

FPS_{RS} represent the throughput in FPS for the reference system.
FPS_{AT} represent the throughput in FPS for the AC technique in question.

The amount of resources (BRAM, DSP, FF and LUT) used is given as the amount saved in comparison to the reference system. The table use a color coding of the values to indicate positive and negative trends. For the resource usage positive savings are coded in green and indicate a reduction in resource usage. While negative trends are coded in red and indicate an increase in resource usage. The RPG is also color coded. However, the RPG show a positive trend when there is an increase in system performance, and a negative trend when the performance decreases.

For the combined techniques certain approaches that are of particular interest are underlined. To distinguish between them, these approaches are also given an annotation using roman numerals.

For the precision scaling the performance deteriorates and the resource usage decreases on a general basis as the approximation level is increased. The memory techniques completely absolves the need for DSP blocks in the sub-functions they affect. The techniques reduce the iteration latency down to what is presumably the minimum. However, the throughput remains unchanged as the latency bottleneck is found in the Compute θ function, that is unaltered.

The iterative approaches reduce the iteration count for the CORDO algorithm. The effect on performance is not seen until a single rotation

is used. That is excluding the four rotation approach. The resources saved scales nicely with the reduced rotation count and substantial savings are attained without affecting the performance.

The error inducing techniques does not show any common trends. This is not surprising. The techniques operates at different functions and at different levels in the system.

It is the loop perforation and kernel reduction approaches that produce the best results for throughput and iteration latency. However, it is not possible to obtain a performance characteristic that is optimal or Pareto optimal when combining these techniques.

The approach with the overall highest system performance is for the scaled kernel reduction. Combining this technique with the gradient LUT 30 by 30 and the angle to 16 states gives the best resource reduction without becoming approximate. The approach is given annotation I. The resulting system performance is 5.52% higher than the reference system. It uses half the amount of BRAM, no DSP blocks and reduces the FF and LUTs by 56.77% and 65.64% respectively. The throughput gain is at 0.123 FPS.

The expansion of this approach to also use loop perforation is given annotation III. It is 3.68% less accurate than the reference system and has an approximate performance characteristic. The resource usage is halved or more for all resource primitives, and no DSP blocks are utilized. The system (III) use 513293 cycles less than the reference system to output a full frame. The throughput is increased from 98 FPS to 195 FPS. This is the highest observed throughput. There is another combined technique that achieves the same result. That is when expanding the approach to also use precision scaling for the intermediary derivative. The approach is given annotation IV. However, this combination of techniques (IV) greatly reduce the system performance. The approach is 92.62% less accurate than the reference system.

Certain combinations of loop perforation and kernel reduction absolves the need for BRAM. The approach that achieves the highest system performance without BRAM is given annotation II. The FF requirement is 93.3% more than the reference system and the performance is 16.55% lower.

Table 4.9: Comparison of techniques and classes.

Class affiliation	Techniques	(Abbreviation)	Performance (rel. %)	TP (FPS gain)	BRAM	DSP	FF	LUT
	Reference system		90.56	97.53	2	3	3683	6443
Precision	Input 7 bits	In7	-1.23%	-	0%	-33%	-0.71%	+0.65%
	Input 6 bits	In6	+1.23%	-	0%	-33%	-2.96%	-0.09%
	Input 5 bits	In5	-20.86%	-	0%	-33%	-2.74%	-0.16%
	Input 4 bits	In4	-65.03%	9.51E-5	0%	-33%	-9.42%	-2.00%
	Input 3 bits	In3	-63.20%	9.51E-5	0%	-33%	-10.53%	-2.44%
	Input 2 bits	In2	-64.42%	9.51E-5	0%	-33%	-9.83%	-2.48%
	Input 1 bit	In1	-46.01%	9.51E-5	0%	-33%	-10.78%	-3.12%
	Intermediary 10 bits	Im10	+1.23%	-	0%	-33%	-0.08%	+0.81%
	Intermediary 9 bits	Im9	+2.45%	-	0%	-33%	-1.71%	+0.22%
	Intermediary 8 bits	Im8	+0.61%	-	0%	-33%	-0.87%	+0.31%
	Intermediary 7 bits	Im7	-7.98%	-	0%	-33%	-2.69%	-0.73%
	Intermediary 6 bits	Im6	-81.59%	-	0%	-33%	-3.34%	-1.07%
	Intermediary 5 bits	Im5	-82.21%	-	0%	-33%	-2.17%	-1.02%
	Intermediary 4 bits	Im4	-82.21%	-	0%	-33%	-2.61%	-1.37%
	Intermediary 3 bits	Im3	-99.38%	-	0%	-33%	-3.04%	-1.75%
Memory	Gradient LUT 128	GL128	-4.30%	-	+550%	-67%	-15.20%	-17.41%
	Gradient LUT 30	GL30	0%	-	0%	-67%	-14.85%	-16.53%
Iterative	CORDIC 5 rotations	C5	0%	1.90E-4	0%	0%	-17.89%	-26.65%
	CORDIC 4 rotations	C4	-11.66%	1.90E-4	0%	0%	-23.02%	-32.42%
	CORDIC 3 rotations	C3	0%	1.90E-4	0%	0%	-29.57%	-38.80%
	CORDIC 2 rotations	C2	0%	1.90E-4	0%	0%	-38.58%	-44.68%
	CORDIC 1 rotation	C1	-1.23%	1.90E-4	0%	0%	-39.80%	-47.14%
Error	Loop perforation 50%	LP50	-3.07%	9.753E1	0%	0%	-0.43%	+0.22%
	LP50 scaled	LP50S	-28.84%	9.753E1	0%	0%	-0.71%	+0.65%
	Kernel reduction	KR	-7.96%	1.22E-1	-50%	0%	-1.28%	-0.11%
	Kernel reduction scaled	KRS	+6.75%	1.22E-1	-50%	0%	-1.55%	-0.12%
	Angle 16 states	Ang16	0%	1.90E-4	0%	-33%	-40.48%	-49.17%
Combined	In7 · C3		-1.23%	2.85E-4	0%	-33%	-30.25%	-38.10%
	In7 · C2		-1.23%	2.85E-4	0%	-33%	-39.26%	-43.99%
	In7 · C1		-3.07%	2.85E-4	0%	-33%	-40.48%	-46.44%
	In6 · C2		+1.23%	2.85E-4	0%	-33%	-40.24%	-44.62%
	In6 · C1		-1.86%	2.85E-4	0%	-33%	-41.41%	-47.00%
	In6 · Im7		0%	-	0%	-33%	-3.94%	-1.04%
	GL30 · C2		0%	8.56E-4	0%	-67%	-53.43%	-61.21%
	Im9 · GL30 · C2		+2.45%	8.56E-4	0%	-67%	-53.92%	-62.05%
	Im9 · GL30 · Ang16		+2.45%	1.24E-3	0%	-100%	-55.80%	-66.54%
	[I] GL30 · KRS · Ang16		+5.52%	1.23E-1	-50%	-100%	-56.77%	-65.64%
	GL30 · C2 · LP50		-4.30%	9.753E1	0%	-67%	-44.58%	-54.03%
	GL30 · C2 · LP50 · KR		-34.36%	9.777E1	-100%	-67%	+93.6%	-12.03%
	Im9 · GL30 · C2 · KR		-10.44%	1.23E-1	-50%	-67%	-55.15%	-62.07%
	[II] GL30 · C2 · LP50S · KRS		-16.55%	9.777E1	-100%	-67%	+93.3%	-12.04%
	C2 · LP50S · KRS		-17.81%	9.777E1	-100%	0%	+108%	+4.39%
	[III] GL30 · LP50 · KRS · Ang16		-3.68%	9.778E1	-50%	-100%	-57.10%	-65.33%
	Im9 · C2 · LP50		-76.69%	9.753E1	0%	-33%	-32.26%	-37.44%
	Im9 · GL30 · C2 · LP50		-76.69%	9.753E1	0%	-67%	-47.19%	-55.18%
	Im9 · GL30 · C2 · LP50 · KR		-82.82%	9.777E1	-100%	-67%	+ 91%	-13.18%
	Im9 · GL30 · KRS · Ang16		-93.85%	1.23E-1	-50%	-100%	-57.97%	-66.79%
	[IV] Im9 · GL30 · LP50 · KRS · Ang16		-92.62%	9.778E1	-50%	-100%	-58.29%	-66.48%

Performance characteristics: Approximate, Semi-approximate, Redundancy, Enhanced, Pareto optimal

Chapter 5

Discussion

This chapter conducts a discussion regarding the analysis presented in the previous chapter, and evaluates the research goals. Remarks on the validity of the results and the simulator is presented as well.

The evaluation metrics. When the effects of the approximations are examined higher in the system hierarchy preconceptions of performance falls short. The emergent behavior does not correlate with the degradation in intermediary image quality at all times. This was seen in Figure 4.24. Where the correlation to the reference system did not indicate a viable measure for the system performance. In Figure 4.20 and Figure 4.22 the detection correctness was plotted against the intermediary degradation in accuracy for the gradient and direction images, respectively. The likelihood of obtaining a high detection rate is higher when the reduction in the intermediary accuracy is low. However, it is not given that a small degradation in image quality results in high system performance.

The metric used to determine the intermediary accuracy is crude. There are other more sophisticated metrics, such as the PSNR or the structural similarity index. If there is a metric of the intermediary quality that gives a better indication of how the loss in accuracy propagates through the system. Then, these observations would be made obsolete.

Implementation constraints. The HLS tool was not given any constraints regarding the use of resource primitives. The HLS tool interpret functionality and transform the behavior into timed RTL implementations. A consequence of this is that the details of the implementations are unknown to the designer. If it is desired to have full control of the implementation details then a HLS approach is obsolete.

The interpreter may not be able to realize an optimum solution for each implementation of the AC techniques. A way to mitigate this is to limit the design to FF and LUTs. Which makes the comparison easier as well. However, this might skew the results and favor certain techniques, as some techniques may benefit more than others. Also, for a real world application, putting artificial constraints on the design is unwanted. The objective of the system and the overall performance is of importance, not the internal mechanisms. This is essentially what a system approach to AC is meant to do, put the system first and the AC second. Consequently, the design choices must favor the objective and performance over comparability.

Validity of the simulator. Since the system is simulated there are design aspects that have not been taken into consideration. If the Sobel filter where to be exported and implemented on an FPGA it would require added functionality. A design choice must be made whether the images are read in to a memory mapped location or not. Or, if the samples are to be streamed in directly. The storage choices greatly affect the resource savings from the precision scaling. If the images are stored then each bit truncated saves $1280 \times 800 = 1024000$ bits in storage per image.

The resources used and the timing estimates are only given as estimates. The results are therefore reliant on the validity of the HLS tool. The actual routing and latency in the system is only attained when the system is run on the FPGA.

Another aspect of consideration is the level of optimization done before AC techniques are applied. The Sobel filter implementation was not fully optimized before being introduced to AC techniques. This can increase the gap in resource savings between the reference system and the AC approaches. However, finding a fully optimized system is a demanding endeavor in itself.

The optimal system encoding. Some of the approaches incur increased performance and substantial savings in relation to the reference system. However, they are not optimal. They just so happened to be the highest performing approaches of those tested. A full search towards an optimal encoding of the system was not conducted. Doing this for the techniques presented would require testing thousands of combinations and turns into a multi-objective optimization problem.

The techniques that operate whit a redundancy performance characteris-

tic never incur performance degradation. Even when combined. Whereas techniques that by themselves have an enhanced or Pareto optimal characteristic, can deteriorate the performance when combined. If this observation holds true for all combination, then the number of possible combinations needed for testing decreases. The techniques with a redundancy characteristic can be applied afterwards, without consideration to the performance.

Performance characteristics. A reason the performance characteristics are of interest is because they highlight the effect of the input characteristics. In how the pixel intensity distribution and the contrast to the AV affect the results. That for some approaches this gives a favorable outcome. This is what is seen when the approaches have an enhanced, Pareto optimal or semi-approximate performance characteristic. It points to one of the major issues any general purpose system faces. How to optimize the system when certain optimizations are only applicable for a given input?

Cognitive computing is a paradigm that can solve this issue. A cognitive computing system is not faced with the problem of finding a general solution for all inputs. It can change its processing to accommodate the input. If, it possible to ascertain what optimization that make the system approximate for the different input characteristics. Then, the system can simply adapt its processing capabilities such that it always operates with the lowest resource usage. Even if it is possible to find a general purpose system that is fully accurate for the full input range, the cognitive computing system could outperform it in resources saved.

Generality of the results. Is this a closed system or does this prove general trends applicable to all systems? In short the answer is no. To adequately answer this requires testing on a larger body of systems. Yet there are some observed trends that are presumed to be general. The first is that any technique that does not alter the performance can be implemented freely. That does not mean that the CORDIC algorithm with 2 rotations can be freely implemented on any system. It refers to the fact that once the performance characteristic of a technique is uncovered it can be used to drive design of combined approaches.

Another general trend is regarding the data representation. The system approach to AC uncovers some properties that are only apparent when the accuracy is examined higher in the system hierarchy. One of these proper-

ties is the relation between accuracy, precision and representation. When the precision is lowered the intermediary accuracy deteriorates. In general this increases the likelihood of a reduced overall system performance. However, when the data representation is altered the intermediary accuracy deteriorates without reducing the system performance. The output of a sub-system is examined as the input to another system. The representation needed in the system that takes the input is what determines the representation in the sub-system.

This is actually just a re-formulation of the one of the core principles in information theory. Of course data should be encoded so that it uses the least amount of bits to represent the information quantity. However, it is not always clear how this is to be done in a larger system and when transformation of data occurs.

Validity of the taxonomy. For the validity of the taxonomy it must be considered if there are AC techniques that can not be classified. Are there any problematic techniques that at first look does not fit into any of the categories? Elision relaxation (relaxed inter task communication correctness) and voltage over-scaling (overclocking) are hard to classify at first glance. However, that is only due to these techniques being viewed outside of the computational description. Describing a transformation of data at a high level is not concerned with the clock frequency nor the scheduling. Once this is viewed as part of the system description classifying the techniques pose no problem. Both techniques create error in the system, and as such falls in under the error inducing category.

Research Question I *When does a computation system become approximate?*

There are two ways to approach this. Either the system becomes approximate whenever the accuracy is lower than the designated reference system. Or any system that is not fully accurate as described by the global accuracy is approximate. It is not certain that for a given input a system has a metric for the global accuracy. Thus it is natural to use the reference system as the measure of accuracy. If the global measure is used instead then all approaches are approximate, even the reference system.

In the simulator used here all approaches that have an approximate or semi-approximate performance characteristic are approximate, while the rest operate with full accuracy. The accuracy gain is redundant as the

system is only required to operate at the accuracy set by the reference system.

For the iterative and precision scaling techniques the point where the system becomes approximate can be determined. The CORDIC algorithm becomes approximate when going from two to one rotation. That is excluding the special case when using four rotations.

For the bit-narrowing of the derivative the approximation point is when going from eight to seven bits. The input bit-narrowing has a performance dip at the first truncation. However, the performance recovers on the next truncation and it is not until a five bit representation that the performance stays consistently low.

Research Question II *Does an approximate computing taxonomy exist that adheres to a system approach?*

The comparison in Table 4.9 show how AC techniques from the different classes can be combined. It is clear that the combined approaches has the potential to reduce the resource usage and increase the throughput far more than any single technique. Thus, the taxonomy aids in the design of AC for larger systems, and adheres to the system approach.

An interesting question that appears when several AC techniques are implemented in the same system is: At what point does incorporating error or changing the system for performance gains turn it into something else? If reducing an algorithm turns it into some other known algorithm, at what point did it cease to be "approximated" and turned into something else? For the system approach and applying the taxonomy this is irrelevant. All internal functionality is trivial, as long as the objective of the system is obtained.

Chapter 6

Conclusion

The main goal of the thesis was to propose a taxonomy that adheres to a system approach for AC. A novel taxonomy was proposed that view a computation system by its key components, and uncovers where approximations can occur. That being either in the data structure or in the transformation of data. The proposed taxonomy classifies AC into four categories: precision, memory, iterative and error techniques. The taxonomy was tested for a system approach to AC on the Sobel filter. Where the Sobel filter in turn was put in the context of a larger object detection system.

The effect of AC when viewed higher in the system hierarchy was examined. It is found that the reduction in intermediary accuracy does not indicate a reduction in the overall system performance. The system was tested with a combination of AC techniques applied. The combined techniques showed greatly improved resource and throughput estimates compared to the single technique implementations.

The system does not necessarily become approximate even though AC techniques have been applied. Significant resource savings can be attained without reducing the final accuracy. For a small reduction in the final accuracy the throughput can be more than doubled while still obtaining considerable resource savings.

6.1 Further work

Physical implementation on hardware. A natural next step is to test the system on actual hardware. This would give the proper timing estimates. It also has the potential to uncover limitations that are otherwise unaccounted for.

Applying the taxonomy to other systems. The taxonomy was only tested for a single system. This makes it hard to determine general trends. Using a system approach for AC on other problems has the potential to uncover properties that are hidden in this system. And, it would be useful to validate or discredit the results found here.

Finding the optimal encoding. It was shown that the system could be improved both in regards to performance and the resource usage. The full width of possible combinations was not tested. Furthermore, only a selection of AC techniques was applied. The optimal encoding is therefor unaccounted for.

AC for cognitive computing systems. Cognitive computing is a field where approximate computing techniques hold great promise, for two primary reasons. The first is pointed out in [4] by Agrawal, Choi, Gopalakrishnan *et al.*, wherein they state that "cognitive applications continue to be executed on general-purpose ... platforms that are highly precise"[4, p. 1]. Thus, they are operating on unnecessary high precision to begin with. The other reason AC is of interest, is because it provides a way to enable different modes of operation. Within the fields of *cognitive radar* and *cognitive radio* a key component is the ability to adapt so that the system does not waste resources running a process that is more computationally expensive than it needs to be [57][58][59]. Efficient resource usage is central for cognitive systems and AC holds the key to enable adaptive processing for any processing task in the cognitive system.

Axiomatic scheme for computational components. The proposed taxonomy adheres to a system approach for AC. However, the taxonomy is less a classification of techniques and more a division of computing into its principal components. For the highest abstraction level this is either data or the transformation of data. Where these components in turn are applicable to certain AC techniques. A natural continuation of the taxonomy is therefor to extend it to an axiomatic scheme that is universal to all computations.

Bibliography

[1] H. B. Barua and K. C. Mondal, "Approximate Computing: A Survey of Recent Trends—Bringing Greenness to Computing and Communication," *Journal of The Institution of Engineers (India): Series B*, vol. 100, no. 6, pp. 619–626, Dec. 2019, ISSN: 2250-2114. DOI: 10.1007/s40031-019-00418-8.

[2] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys (CSUR)*, Mar. 2016. [Online]. Available: https://dl-acm-org.ezproxy.uio.no/doi/abs/10.1145/2893356.

[3] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[4] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, V. Srinivasan and Z. Sura, "Approximate computing: Challenges and opportunities," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, IEEE, Oct. 2016, pp. 1–8. DOI: 10.1109/ICRC.2016.7738674.

[5] W. Liu, F. Lombardi and M. Shulte, "A retrospective and prospective view of approximate computing [point of view," *Proceedings of the IEEE*, vol. 108, no. 3, pp. 394–399, 2020.

[6] A. Aponte-Moreno, A. Moncada, F. Restrepo-Calle and C. Pedraza, "A review of approximate computing techniques towards fault mitigation in HW/SW systems," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, IEEE, Mar. 2018, pp. 1–6. DOI: 10.1109/LATW.2018.8347241.

[7] Q. Xu, T. Mytkowicz and N. S. Kim, "Approximate Computing: A Survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, Dec. 2015, ISSN: 2168-2364. DOI: 10.1109/MDAT.2015.2505723.

[8] T. Moreau, J. S. Miguel, M. Wyse, J. Bornholt, A. Alaghi, L. Ceze, N. E. Jerger and A. Sampson, "A Taxonomy of General Purpose Approximate Computing Techniques," *IEEE Embedded Systems Letters*, vol. 10, no. 1, pp. 2–5, Oct. 2017. DOI: 10.1109/LES.2017.2758679.

[9] D. Ma, R. Thapa, X. Wang, X. Jiao and C. Hao, "Workload-aware approximate computing configuration," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021, pp. 920–925.

[10] A. Zarei and F. Safaei, "Power and area-efficient design of vcma-mram based full-adder using approximate computing for iot applications," *Microelectronics journal*, vol. 82, pp. 62–70, 2018.

[11] Y. Chung and Y. Kim, "Comparison of approximate computing with sobel edge detection," *IEIE Transactions on Smart Processing & Computing*, vol. 10, no. 4, pp. 355–361, 2021.

[12] G. Ndour, T. T. Jost, A. Molnos, Y. Durand and A. Tisserand, "Evaluation of variable bit-width units in a RISC-V processor for approximate computing," in *CF '19: Proceedings of the 16th ACM International Conference on Computing Frontiers*, New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 344–349, ISBN: 978-1-45036685-4. DOI: 10.1145/3310273.3323159.

[13] A. Aponte-Moreno, C. Pedraza and F. Restrepo-Calle, "Reducing Overheads in Software-based Fault Tolerant Systems using Approximate Computing," in *2019 IEEE Latin American Test Symposium (LATS)*, IEEE, pp. 11–13. DOI: 10.1109/LATW.2019.8704586.

[14] B. S. Prabakaran, S. Rehman, M. A. Hanif, S. Ullah, G. Mazaheri, A. Kumar and M. Shafique, "Demas: An efficient design methodology for building approximate adders for fpga-based systems," eng, in *2018 Design, Automation & Test in Europe Conference & Exhibition*, vol. 2018-, EDAA, 2018, pp. 917–920, ISBN: 9783981926316.

[15] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan and K. Roy, "IMPACT: IMPrecise adders for low-power approximate computing," *IEEE*, pp. 1–3, 2011. DOI: 10.1109/ISLPED.2011.5993675.

[16] V. Gupta, D. Mohapatra, A. Raghunathan and K. Roy, "Low-power digital signal processing using approximate adders," eng, *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 32, no. 1, pp. 124–137, 2013, ISSN: 0278-0070.

[17] R. Ye, T. Wang, F. Yuan, R. Kumar and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 48–54. DOI: 10.1109/ICCAD.2013.6691096.

[18] P. Kulkarni, P. Gupta and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *2011 24th Internatioal Conference on VLSI Design*, 2011, pp. 346–351. DOI: 10.1109/VLSID.2011.51.

[19] S. R. Faraji, P. Abillama and K. Bazargan, "Low-cost approximate constant coefficient hybrid binary-unary multiplier for dsp applications," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 93–101. DOI: 10.1109/FCCM48280.2020.00022.

[20] A. B. Kahng, S. Kang, R. Kumar and J. Sartori, "Recovery-driven design: Exploiting error resilience in design of energy-efficient processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 404–417, 2012. DOI: 10.1109/TCAD.2011.2172610.

[21] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique and A. Kumar, "Area-optimized low-latency approximate multipliers for fpga-based hardware accelerators," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18, San Francisco, California: Association for Computing Machinery, 2018, ISBN: 9781450357005. DOI: 10.1145/3195970.3195996. [Online]. Available: https://doi-org.ezproxy.uio.no/10.1145/3195970.3195996.

[22] M. J. Wirthlin, "Constant coefficient multiplication using look-up tables," eng, *Journal of VLSI signal processing*, vol. 36, no. 1, pp. 7–15, 2004, ISSN: 0922-5773.

[23] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing," *University of Washington Technical Report UW-CSE-15-01*, vol. 1, no. 2, 2015.

[24] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze and M. Oskin, "SNNAP: Approximate computing on programmable SoCs via neural acceleration," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*,

pp. 603–614, Jul. 2011, ISSN: 2378-203X. DOI: 10.1109/HPCA.2015.7056066.

[25] R. Andraka, *A survey of CORDIC algorithms for FPGA based computers*. New York, NY, USA: Association for Computing Machinery, Mar. 1998, ISBN: 978-089791978. DOI: 10.1145/275107.275139.

[26] Y. Liu, L. Fan and T. Ma, "A Modified CORDIC FPGA Implementation for Wave Generation," *Circuits, Systems, and Signal Processing*, vol. 33, no. 1, pp. 321–329, Jan. 2014, ISSN: 1531-5878. DOI: 10.1007/s00034-013-9638-8.

[27] Q. Wen, Y. Liang, T. Adriano, T. Ricardo and C. Wu, "A Novel Iterative Velocity Control Algorithm and Its FPGA Implementation Based on Trigonometric Function," *Chinese Journal of Electronics*, vol. 28, no. 2, pp. 237–245, Mar. 2019, ISSN: 1022-4653. DOI: 10.1049/cje.2019.01.003.

[28] S. Lee and A. Gerstlauer, "Fine grain word length optimization for dynamic precision scaling in dsp systems," in *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013, pp. 266–271. DOI: 10.1109/VLSI-SoC.2013.6673287.

[29] A. Roldao-Lopes, A. Shahzad, G. A. Constantinides and E. C. Kerrigan, "More Flops or More Precision? Accuracy Parameterizable Linear Equation Solvers for Model Predictive Control," *Proceedings - IEEE Symposium on Field Programmable Custom Computing Machines, FCCM 2009*, pp. 209–216, May 2009. DOI: 10.1109/FCCM.2009.19.

[30] D.-U. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk and G. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, 2006. DOI: 10.1109/TCAD.2006.873887.

[31] H.-N. Nguyen, D. Menard and O. Sentieys, "Dynamic precision scaling for low power wcdma receiver," in *2009 IEEE International Symposium on Circuits and Systems*, 2009, pp. 205–208. DOI: 10.1109/ISCAS.2009.5117721.

[32] Y. Tian, Q. Zhang, T. Wang, F. Yuan and Q. Xu, "Approxma: Approximate memory access for dynamic precision scaling," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15, Pittsburgh, Pennsylvania, USA: Association for

Computing Machinery, 2015, pp. 337–342, ISBN: 9781450334747. DOI: 10.1145/2742060.2743759. [Online]. Available: https://doi.org/10.1145/2742060.2743759.

[33] W. Weaver, "Recent contributions to the mathematical theory of communication," *ETC: a review of general semantics*, pp. 261–281, 1953.

[34] J. Gleick, *The information: A history, a theory, a flood*. Vintage, 2011.

[35] S. R. Kulkarni, "Information, entropy, and coding," *ELE 201: Information Signals*, 2016. [Online]. Available: https://www.princeton.edu/~cuff/ele201/kulkarni.html.

[36] P. Grunwald and P. Vitányi, "Shannon information and kolmogorov complexity," *arXiv preprint cs/0410002*, 2004.

[37] A. Kolmogorov, "Logical basis for information theory and probability theory," *IEEE Transactions on Information Theory*, vol. 14, no. 5, pp. 662–664, 1968. DOI: 10.1109/TIT.1968.1054210.

[38] I. Sobel, "An Isotropic 3x3 Image Gradient Operator," *Presentation at Stanford A.I. Project 1968*, Feb. 2014. [Online]. Available: https://www.researchgate.net/publication/239398674_An_Isotropic_3x3_Image_Gradient_Operator.

[39] O. R. Vincent, O. Folorunso *et al.*, "A descriptive algorithm for sobel image edge detection," in *Proceedings of informing science & IT education conference (InSITE)*, vol. 40, 2009, pp. 97–107.

[40] T. Kittelsen, *Far, far away Soria Moria Palace shimmered like Gold – Nasjonalmuseet – Collection*, [Online; accessed 19. Apr. 2022], Apr. 1900. [Online]. Available: https://www.nasjonalmuseet.no/en/collection/object/NG.M.00546.

[41] S. Perri, F. Spagnolo, F. Frustaci and P. Corsonello, "Efficient Approximate Adders for FPGA-Based Data-Paths," *Electronics*, vol. 9, no. 9, p. 1529, Sep. 2020, ISSN: 2079-9292. DOI: 10.3390/electronics9091529.

[42] J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959, ISSN: 0367-9950. DOI: 10.1109/TEC.1959.5222693.

[43] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann and M. Rinard, “Managing performance vs. accuracy trade-offs with loop perforation,” *ESEC/FSE ’11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, Sep. 2011. DOI: 10.1145/2025113.2025133.

[44] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” in *2013 18th IEEE European Test Symposium (ETS)*, IEEE, 2013, pp. 27–30. DOI: 10.1109/ETS.2013.6569370.

[45] J. BROGAN, “What’s the deal with algorithms?” *Future Tense*, 2016.

[46] Y. Tian, Q. Zhang, T. Wang and Q. Xu, “Lookup table allocation for approximate computing with memory under quality constraints,” in *2018 Design, Automation Test in Europe Conference Exhibition*, 2018, pp. 153–158. DOI: 10.23919/DATE.2018.8341995.

[47] J. D. Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori and O. Mutlu, *Pluto: In-dram lookup tables to enable massively parallel general-purpose computation*, 2021. arXiv: 2104.07699 [cs.AR].

[48] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira and O. Mutlu, “Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture,” *arXiv preprint arXiv:2105.03814*, 2021.

[49] J. S. Miguel, M. Badr and N. E. Jerger, “Load value approximation,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 127–139. DOI: 10.1109/MICRO.2014.22.

[50] P. K. Meher, “Lut optimization for memory-based computation,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 57, no. 4, pp. 285–289, 2010. DOI: 10.1109/TCSII.2010.2043467.

[51] D. Boland and G. A. Constantinides, “An FPGA-based implementation of the MINRES algorithm,” *IEEE*, pp. 8–10, 2008. DOI: 10.1109/FPL.2008.4629967.

[52] Q. Zhang, F. Yuan, R. Ye and Q. Xu, “Approxit: An approximate computing framework for iterative methods,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6. DOI: 10.1109/DAC.2014.6881424.

[53] T. Kemp, Y. Yao and Y. Kim, "Mipac: Dynamic input-aware accuracy control for dynamic auto-tuning of iterative approximate computing," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2021, pp. 248–253.

[54] S. Ullah, S. Rehman, M. Shafique and A. Kumar, "High-performance accurate and approximate multipliers for fpga-based hardware accelerators," eng, *IEEE transactions on computer-aided design of integrated circuits and systems*, pp. 1–1, 2021, ISSN: 0278-0070.

[55] *Ensuring the Continuous Flow of Data and Data Reuse • Vitis High-Level Synthesis User Guide (UG1399) • Reader • Documentation Portal*, [Online; accessed 18. May 2022], May 2022. [Online]. Available: `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Ensuring-the-Continuous-Flow-of-Data-and-Data-Reuse`.

[56] Xilinx, *HLx_Examples*, [Online; accessed 21. May 2022], May 2022. [Online]. Available: `https://github.com/Xilinx/HLx_Examples/tree/master/Math/atan2_cordic`.

[57] J. Mitola and G. Maguire, "Cognitive radio: Making software radios more personal," *IEEE Personal Communications*, vol. 6, no. 4, pp. 13–18, 1999. DOI: `10.1109/98.788210`.

[58] R. J. Abad, M. H. Ierkic and E. I. Ortiz-Rivera, "Basic understanding of cognitive radar," *ResearchGate*, pp. 1–4, Oct. 2016. DOI: `10.1109/ANDESCON.2016.7836270`.

[59] S. Haykin, "Cognitive radio: Brain-empowered wireless communications," *IEEE journal on selected areas in communications*, vol. 23, no. 2, pp. 201–220, 2005.