Accuracy Configurable FPGA Implementation of Image Processing Algorithms

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Electronics and Communication Engineering
by Research

by

Shivani Maurya
201234001
shivani.maurya@research.iiit.ac.in

International Institute of Information Technology, Hyderabad
(Deemed to be University)
Hyderabad - 500 032, INDIA
January 2023

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "**Accuracy Configurable FPGA Implementation of Image Processing Algorithms**" by **Shivani Maurya**, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Prof. Suresh Purini

To my parents and sister

Acknowledgments

I would like to express my gratitude to my primary supervisor, Dr. Suresh Purini, for his continuous support and encouragement during my graduate study and helping me finalize my project. I am grateful for all the kindness and patience he has shown me through the tough transition. This thesis owes its completion to his profound knowledge and valuable feedback on my research.

I thank my mentor Ziaul Choudhury for sharing his knowledge and insights. I would like to thank him for answering my doubts and sounding out my ideas. He is an inspiration to all budding researchers. The discussions with him were a great learning opportunity for me.

I am also grateful to my friends Anupriya, Sunayna and Naresh for their support and words of encouragement. To have them celebrate my good times and share in my bad ones was a blessing. I will be forever grateful to my oldest friend from school days and perhaps my ever constant pillar of support, Shruti. My formative years would have been very different but for her presence.

I want to sincerely thank my parents for believing in me and keeping me motivated every time I doubted myself. They made sure I remained optimistic and confident. Lastly, I would thank my sister Divya, for having my back. Her constant support throughout my life has changed me for the better and kept me going through ups and downs. I am fortunate to have been blessed with her.

Abstract

For decades, process technology scaling has catered to ever increasing demands of high-speed integrated circuits. Many applications, such as image and multimedia data processing, artificial intelligence, machine learning etc., depend on these circuits to process huge swaths of data in real time. Be it in the data centers, the cloud networks or the mobile devices, the computing efficiency of the circuits has had to play catch-up with this ever-increasing demand. However, it has become abundantly clear in recent years that cranking the technology knob isn't a feasible solution to future needs of high-speed computing.

Across the spectrum, data processing applications are endowed with resilience to acceptable levels of errors in the underlying computations. These applications incorporate, what can only be called a "forgiving" nature, into their algorithms without compromising with the end-user experience. And this intrinsic robustness to errors can be leveraged even further by the design methodology of Approximate Computing. It has now become an independent field that explores methods to reduce computation costs by allowing minor degradation in intermediate computations. Several approximate arithmetic units have been extensively studied and implemented with significant impacts on the system costs in terms of power, area and speed.

Image processing algorithms with intrinsic robustness to errors can be approximated for significant resource and energy savings while still meeting the end-user requirements. FPGA-based implementations can increase their suitability for real-time high-speed multimedia applications by leveraging Approximate Computing. With high volume of pixel level computations, algorithms such as the Harris Corner Detector (HCD) and Unsharp Making (USM), become targets for such approximation strategies. In this thesis, we propose hardware implementations of these algorithms that rely on approximating the intermediate multiplication operations using Dynamic Range Unbiased Multiplier (DRUM). With run-time configurable bit-width control of DRUM instances, their quality of outputs is shown to depend on the varying accuracy. We explore how the errors due to approximate operations propagate to the output. Further,the experimental results from implementations on Virtex-7 and Zynq-7000 FPGA devices are documented and an analytical approach based on quality metric comparisons with base implementation is presented.

Keywords **–** Approximate Computing, DRUM Multiplier, Image Processing Benchmarks, Harris Corner Detection, Unsharp Mask

Contents

List of Figures

List of Tables

List of Abbreviations

ACU	Accuracy
BSV	Bluespec SystemVerilog
$DRUM$	Dynamic Range Unbiased Multiplier
HCD	Harris Corner Detector
USM	Unsharp Masking
MI	Mis-classification Index
$SSIM$	Structural Similarity Index
$FPGA$	Field Programmable Gate Array
LUT	Look-Up Tables
FF	Flip Flops
$BRAM$	Block Random Access Memory
DSP	Digital Signal Processors
$SAIF$	Switching Activity Interchange Format
DAG	Directed Acyclic Graph

Chapter 1

Introduction

1.1 Approximate Computing

Concerns of energy efficiency are front and center in the design of computing systems. As more and more aspects of our lives become computerized and connected, today's designs continue to increase their energy use. With the increase in computations, Approximate Computing has emerged as a novel architecture that aims to address these energy utilization concerns. Many applications do not need highly accurate computations to provide acceptable user experience. This acceptance of permissible inaccuracy, results in significant energy and performance improvements. Applications in fields like computer vision, media processing, artificial intelligence etc. already incorporate imprecision into their design. The focus is shifted from accuracy of individual data elements to the aggregate effects of their inaccuracies.

1.2 Need for Approximation

Traditional design architectures of real-time computer vision algorithms have failed to deliver on reducing resource, power and area costs. With increasing demands for low-power and high-performance, a shift from traditional accurate computing to approximate computing [7] was inevitable. A wide range of applications such as interest point detection [8], image compression [14], image-guided navigation etc. favour having their computations performed in an approximate or imprecise manner. This allowance of permissible inaccuracy not only results in significant power and area savings but also provides acceptable user experience. The focus is shifted from accuracy of individual data elements to the aggregate effects of their inaccuracies.

This methodology, however, has its fair share of challenges. The central challenge being the ability to provide a disciplined control on imprecision. Blanket approximation to the design, whereby every multiplication or addition operation is approximated, produces huge output quality loss. Effective approximation, on the other hand, requires careful estimation of error propagation and judicious application of approximation to operations that result in tangible savings in resources. Quality specifications

need to be met as well. These require additional scrutiny of the output and re-execution of the approximate design portions with refined programmable parameters. There is no shortage of applications that allow such guided approximations while still providing acceptable results to the perceptually limited humans. For our purposes we chose corner detection and unsharp masking.

1.3 Current Research on Harris Corner Detection and Unsharp Mask

Corner detection is one of the fundamental pre-processing steps in object detection, motion-tracking, SLAM (simultaneous localization and mapping) [9], camera calibration [19] and image stitching [20]. With its strong robustness to noise and invariance to image transformations, Harris Corner Detection (HCD) [5] has emerged as the most widely used algorithm for detecting sharp corners. It computes the corner response for every pixel in the image, whereby a pixel is considered a corner if its response is the local maxima and exceeds a given threshold. In comparative survey [21], HCD has achieved the best performance among other feature detectors. Motivated by the need to refine the corner search space, thresholding [18] has also garnered attention. Attempts have also been made at the hardware implementation of HCD [2, 17, 4], aiming to reduce its computational complexity and resource utilization while ensuring high throughput [1] and quality. Techniques to refine the corner search space using thresholding [18] and to reduce power [10] based on data-path optimizations have also been explored.

On the other hand, Unsharp Masking (USM) finds widespread application in photography [11, 12] and medical imaging [16]. In order to digitally improve the image quality, the algorithm emphasizes the high frequency contents to enhance the edge and detail information in it. The comparative survey [3] of different unsharp masking methods documents the shift from linear masking in order to reduce the effects of noise and appearance of overshoot artefacts. Various adaptive approaches [22] have also been suggested to reduce the noise sensitivity of the linear USM techniques.

All this constitutes a huge body of literature aimed at improving the algorithms in both the software and hardware space. However, there has been little effort undertaken to investigate the aforementioned challenges against the backdrop of approximate computing strategies. Harnessing the inherent parallelism of HCD and USM, we demonstrate how the approximation at different pipeline stages affects the corner response accuracy and the image sharpness respectively. We outline an approximation strategy that ensures acceptable throughput-quality trade-off. We also propose modification in the original response function of HCD algorithm which helps decide the threshold across images for robust corner detection. While we only focused on HCD and USM algorithms in this work, the proposed strategies can be applied to other computer vision algorithms (Gaussian pyramid, optical flow etc.).

1.4 Contributions of this Thesis and Related Work

In this thesis, we explore the application of an accuracy configurable approximate multiplier in pipelined hardware implementations of image processing algorithms. To the best of our knowledge, the hardware implementation of HCD using approximated multiplication has not been published before. Thus, the study reported in this dissertation constitutes a novel contribution.

A mathematical characterization of accuracy based on operational error propagation analysis is first presented. To this end, the configurable input bit-width of the approximate multiplier is parameterized for different operations and the resulting mean multiplication error along with the errors in the multiplier inputs is used in the analysis of operations at different pipeline stages. The resulting equations provide insights into the approximation to be introduced at each pipeline stage for achieving desired output quality.

The approach to threshold selection is presented and compared across different images for the two algorithms. We propose modification in the original response function of the HCD algorithm which helps decide the threshold across images for robust corner detection. On the other hand, the Structural Similarity Index (SSIM) is used to adaptively select the threshold for permissible output quality from USM algorithm.

Application specific metrics are used to qualify the different accuracy implementations against the base implementation of respective algorithms. Power, speed and resource utilization data has been compiled to assess the comparative benefits of different accuracy setups. A comparative study with other hardware implementations is performed to evaluate the proposed approximate architectures. The baseline and proposed architectures were implemented using Bluespec SystemVerilog (BSV) and synthesized on Zynq-7000 (xc7z045ffv900-2) and Xilinx Virtex-7 (xc7v585ttfg1157-2) FPGA devices using Xilinx Vivado 2018.3 ISE software. Comparisons of performance characteristics provide unique insights into the applicability of the different configurations.

1.5 Thesis Organization

The outline of the remainder of the thesis is as follows. Chapter 2 summarizes the different image processing algorithms to introduce notations and main concepts. The approximate multiplier of choice is also discussed. Chapter 3 outlines the improved approximation strategy for HCD algorithm based on operational error propagation analysis. The challenges faced during threshold selection and solutions proposed for the same are discussed. It also presents simulation results compared to base hardware implementation. Chapter 4 outlines the approximation strategy for USM algorithm using the same operational error analysis. Approach to optimal threshold selection is discussed and comparative results are presented. Chapter 5 concludes the thesis.

Chapter 2

Related Algorithms

2.1 Image Processing Algorithms

Given the ubiquitous application of image processing algorithms in the current world, they are our prime targets for approximation in this thesis. Harris Corner Detection, Unsharp Masking etc. are some of the algorithms conducive to approximation and will be discussed briefly in this chapter.

2.1.1 Harris Corner Detection Algorithm

In the domain of computer vision, a point qualifies as a corner if its gradient is largest in all directions. Visually this appears as a drastic change in brightness. This simple idea behind what constitutes a corner was leveraged by Harris and Stephens [5], who improved upon the Moravec corner detector by making it less sensitive to noise. The algorithm determines whether a small window around each pixel p in an image contains a corner feature or not. This is accomplished by shifting each window by a small amount and measuring the amount of change that occurs in the pixel values. The response value R is computed for each pixel and if it exceeds a threshold value, the pixel is considered to be a corner. The HCD algorithm is implemented as follows:

Step 1 . Calculating the gradients I_x, I_y :

$$I_x = \frac{\delta I}{\delta x} = I \otimes K_{Sobelx}$$

$$I_y = \frac{\delta I}{\delta y} = I \otimes K_{Sobely} \tag{2.1}$$

where I_x and I_y are the gradients of the image (I) at pixel (x, y) in horizontal and vertical directions respectively.

Step 2 . Calculating the products of the gradients :

$$I_{xx} = I_x.I_x, I_{yy} = I_y.I_y, I_{xy} = I_x.I_y \tag{2.2}$$

Step 3 . These calculations are followed by Gaussian smoothing over window W to obtain matrix M :

$$M = \begin{bmatrix} \sum_{x,y \in W} I_{xx} & \sum_{x,y \in W} I_{x,y} \\ \sum_{x,y \in W} I_{x,y} & \sum_{x,y \in W} I_{yy} \end{bmatrix} = \begin{bmatrix} S_{xx} & S_{x,y} \\ S_{x,y} & S_{yy} \end{bmatrix} \tag{2.3}$$

Step 4 . Once the matrix M is obtained, the response value R for each pixel is computed :

$$R = det(M) - k.tr(M)^2 \tag{2.4}$$

where k is a constant whose value lies between 0.04 and 0.06.

Step 5 . Lastly, the corner measure R of each pixel is compared with a threshold T. The non-corners get there R value replaced by 0 while the corner candidates retain their measure. The refined R is then subjected to non-maximal suppression (NMS), which determines whether the center pixel's R is the local maxima among the 8-connected pixels. If true, the center pixel is deemed a corner. The HCD pipeline, visualized as a directed acyclic graph (DAG) of computations, is illustrated in Fig. 2.1. The output from one stage gets consumed by the subsequent stages in the pipeline.

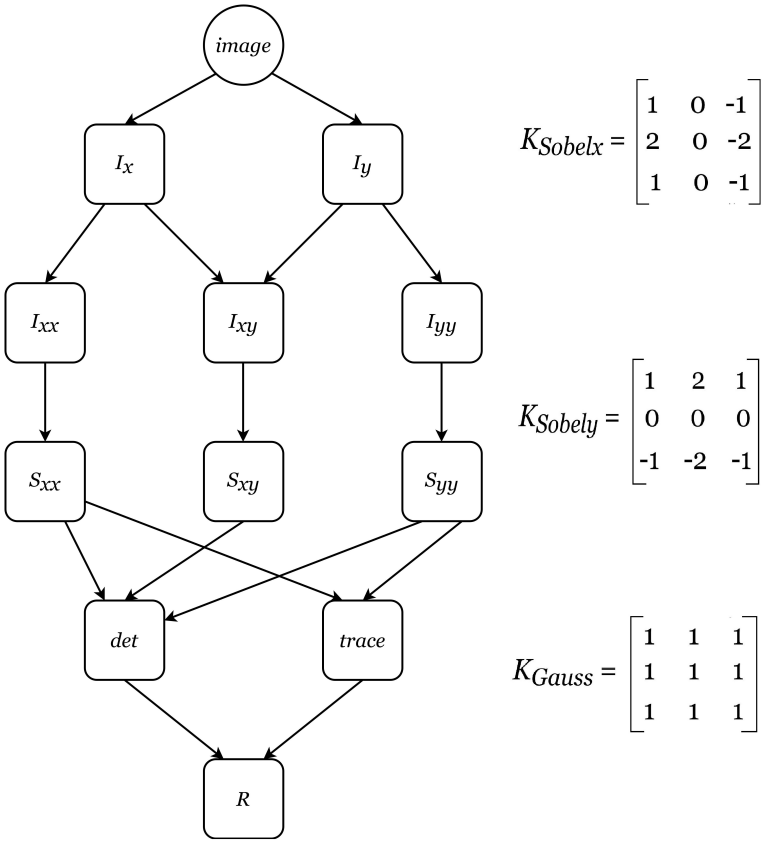


Figure 2.1: DAG representation of the Harris Corner Detection (HCD) algorithm

The final R value obtained from the last stage is subjected to a threshold, followed by non-maximum suppression. The quality of the detected corner candidates depends not only on the response function

computation but also on the threshold used to discern them. A high threshold will detect only very strong corner candidates, while a low threshold will detect many false ones. Thus, finding the ideal threshold becomes a trade-off between the number of undetected true corners and detected false corners. This threshold varies from image to image. Fig. 2.2 shows corners detected for two grayscale test images of 256×256 resolution.

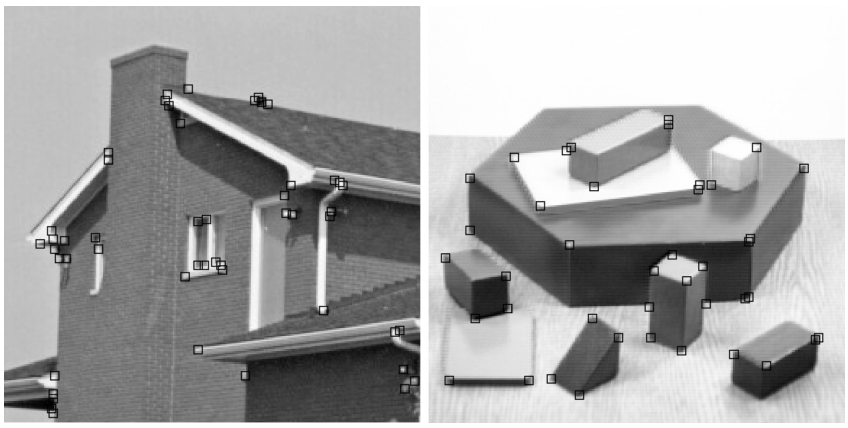


Figure 2.2: Corner detection on house and block test images

2.1.2 Unsharp Masking Algorithm

The visual appearance of an image can be improved by sharpening the edges in the image. The amount of unsharpness of its blurry duplicate becomes critical in determining the quality of the final image. The blurred duplicate is subtracted away from the original to detect the presence of edges, creating the unsharp mask (effectively a high-pass filter). Computing a weighted sum of the image and its blurred counterpart at pixels where both differ significantly in intensity, exaggerates the light and dark edges of the transition. Contrast gets selectively increased along these edges resulting in a sharpened image. The USM algorithm is implemented as follows :

Step 1 . The image is passed through Gaussian filters to produce blur across x-axis and y-axis directions.

$$I_{blurx} = I_{org} \otimes GaussianfilterX$$
$$I_{blury} = I_{blurx} \otimes GaussianfliterY \tag{2.5}$$

Step 2 . Then it is sharpened by taking weighted sum of the original and the resultant blurry image.

$$I_{sharp} = I_{org}.W + I_{blury}.W' \tag{2.6}$$

where W and W' are scaling factors that control the level of sharpness to be introduced at later stage.

Step 3 . Finally, depending on whether the value of the mask is greater or less than the threshold, the corresponding pixel from either the original input image or the sharpened image is chosen for the output composite.

$$I_{mask} = I_{org} - I_{blury} \tag{2.7}$$

$$I_{final} = \begin{cases} I_{org}, I_{mask} < Threshold \\ I_{sharp}, I_{mask} > Threshold \end{cases} \tag{2.8}$$

The USM pipeline, visualized as DAG, is illustrated in Fig. 2.3. The mask constitutes an interest point descriptor. When compared with the threshold, it allows selection of pixels to be sharpened. This makes threshold an important parameter in pruning such candidate pixels in order to avoid undesirable artefacts. The algorithm was tested on images with the resolution of 1024 x 1024. Fig. 2.4 illustrates the change from original image to the sharpened image.

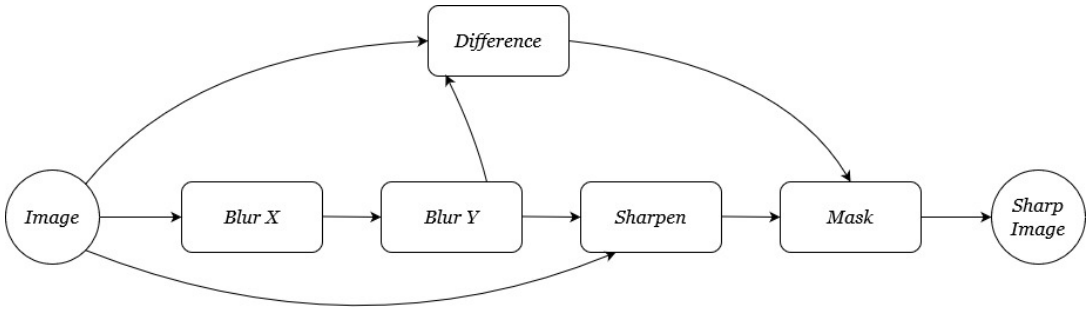


Figure 2.3: DAG representation of the Unsharp Mask (USM) algorithm

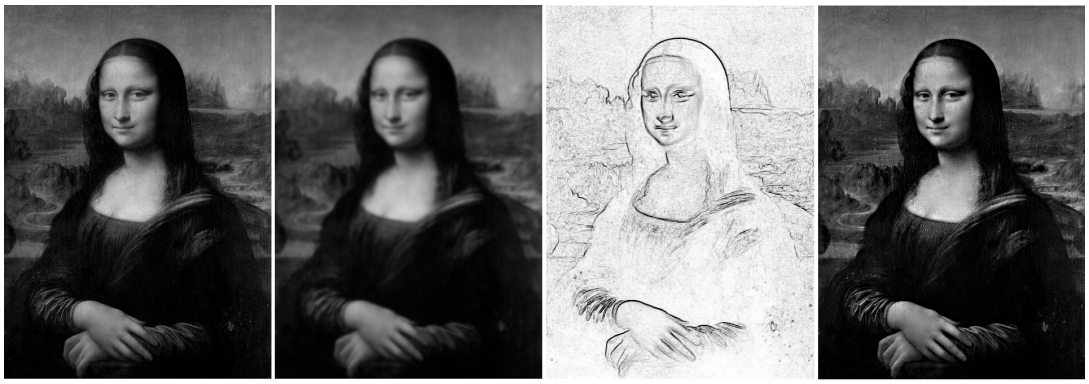


Figure 2.4: From left to right : Original Image, Blurred Image, Image Mask, Sharpened Image

2.1.3 Lucas Kanade Algorithm

Motion estimation is the process of determining motion vectors that describe the transformation from one image frame to the next. Motion estimation techniques are used in target tracking and monitoring. The motion vectors may relate to the image or its pixels etc. whose projected motion is recovered from intensity variation across frames. The Lucas–Kanade method is one such widely used differential method for optical flow estimation. This method solves the basic optical flow equations for all the pixels in a neighbourhood by the least squares criterion. It assumes that the flow is small and essentially constant in the local neighbourhood of the pixel under consideration and provides an estimate of its movement in successive frames. By looking at changes in pixel intensity which can be obtained from the intensity gradients of the image in that neighborhood, it makes a "best guess" of its displacement.

Step 1 . Taking intensity gradients of the pixels in the observation window,

$$\begin{aligned} I_x(p_1)V_x + I_y(p_1)V_y &= -I_t(p_1) \\ I_x(p_2)V_x + I_y(p_2)V_y &= -I_t(p_2) \\ &.... \\ I_x(p_n)V_x + I_y(p_n)V_y &= -I_t(p_n) \end{aligned} \tag{2.9}$$

here, $p_1, p_2...p_n$ are pixels inside the window under consideration and $I_x(p_n), I_y(p_n), I_t(p_n)$ are the partial derivatives of the image I with respect to coordinates x, y and time t, evaluated at the pixel p_n and at the current time.

Step 2 . Reducing above equations to matrix form,

$$\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ . & \\ .I_x(p_n) & I_y(p_n) \end{bmatrix} \begin{bmatrix} V_x & V_y \end{bmatrix} = \begin{bmatrix} -I_t(p_1) \\ -I_t(p_2) \\ . \\ . - I_t(p_n) \end{bmatrix} \tag{2.10}$$

$$I_{x,y}V_{x,y} = -I_t \tag{2.11}$$

Step 3 . Using least squares solution by giving more weight to the pixels that are closer to the central pixel p,

$$V_{x,y} = (I_{x,y}^T W I_{x,y})^{-1} I_{x,y}^T W I_t \tag{2.12}$$

here, W is the diagonal matrix containing weights for each pixel

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \frac{1}{D} \begin{bmatrix} \sum_i w_i I_y(p_n)^2 & -\sum_i w_i I_x(p_n) I_y(p_n) \\ -\sum_i w_i I_x(p_n) I_y(p_n) & \sum_i w_i I_x(p_n)^2 \end{bmatrix} \begin{bmatrix} -\sum_i w_i I_x(p_n) I_t(p_n) \\ -\sum_i w_i I_y(p_n) I_t(p_n) \end{bmatrix} \tag{2.13}$$

$$D = \sum_i w_i I_x(p_n)^2 \sum_i w_i I_y(p_n)^2 - \left(\sum_i w_i I_x(p_n) I_y(p_n) \right)^2 \tag{2.14}$$

Here $V_x(= d_x/d_t)$ denotes the x component of pixel velocity and $V_y(= d_y/d_t)$ denotes the y component of pixel velocity. Solving for the two variables completes the optical flow problem. The Lucas-Kanade method although linear in complexity, only works for small movements and fails when there is large motion.

Figure 2.5: Optical flow vectors from Lucas Kanade Algorithm

2.1.4 Gaussian Pyramid Algorithm

The Gaussian pyramid is a recursive algorithm for creating a multi-resolution version of an image until some stopping criteria are met. It consists of low-pass filtered, down-sampled images at successive levels, where the base level is defined as the original image. Each element of the pyramid represents a local average obtained by convolving the Gaussian kernel with the image from the previous level. Applied recursively, this algorithm generates a sequence of images, subsequent ones being smaller, lower resolution versions of the earlier ones in the processing. Thus, the Gaussian pyramid contains local averages at various scales. This feature of the pyramid has been leveraged for texture analysis and target localization.

Keeping the Gaussian kernel same, at each step up level the image resolution is down-sampled by 2. So if starting image size was 256 X 256 at level 0 in level 1 image size will be 128 X 128 and so on.

Figure 2.6: Images from Gaussian Pyramid Algorithm

Let the original image be denoted by I_{org}. The Gaussian pyramid is defined recursively as follows,

$$I_0(x, y) = I_{org}$$

$$I_l(x, y) = G.I_{l-1}$$

Alternatively, the same result can be obtained by applying an equivalent Gaussian kernel W_l directly to the original image, followed by l down-sampling operations (D_l), where l denotes the level number, such that each G_l is the blur-and-downsample operator for level l and doubles in scale with each level.

$$I_l(x, y) = D_l.W_l.I_{org} = G_l.I_{org}, \tag{2.15}$$

where D_l is the downsampling operator, W_l is the Gaussian filter.

2.2 Multiplication operation in algorithms

The high number of multiplications involved in the image processing benchmarks discussed in the previous section, makes them most amenable to usage of approximate multipliers. Algorithms involving convolutions scale up in computational complexity with the increase in the image or kernel size. Introducing approximation in these operations leads to significant savings in power and area which will be explored further in coming sections.

2.2.1 Dynamic Range Unbiased Multiplier

The DRUM proposed in [6] operates on n-bit operands and truncates them to k bits ($k < n$) from the location of the leading one in both the operands. Assuming each operand has n bits, the design uses two leading one detector (LOD) circuits to dynamically locate the most significant '1' in each of the two operands. For each operand, the location of the most significant '1' is then used to select the following $k-2$ consecutive number of bits based on the required accuracy. Here, k is a designer-defined value which specifies the bandwidth used in the core accurate multiplier. If the leading one is detected at index t, where $0 \leq t \leq n-1$, then we unbias the value of the remaining $t-k+2$ lower bits of the number by placing '1' at bit location $t-k+1$ (the highest bit) and the remaining bits are set to zeros and truncated leading to k-bits. If the leading '1' is found within the least significant k bits of the operand, then the least k bits are directly forwarded to the multiplier. These dynamically truncated k-bit operands are then steered to the inputs of a $k \times k$ core multiplier (Wallace-tree).

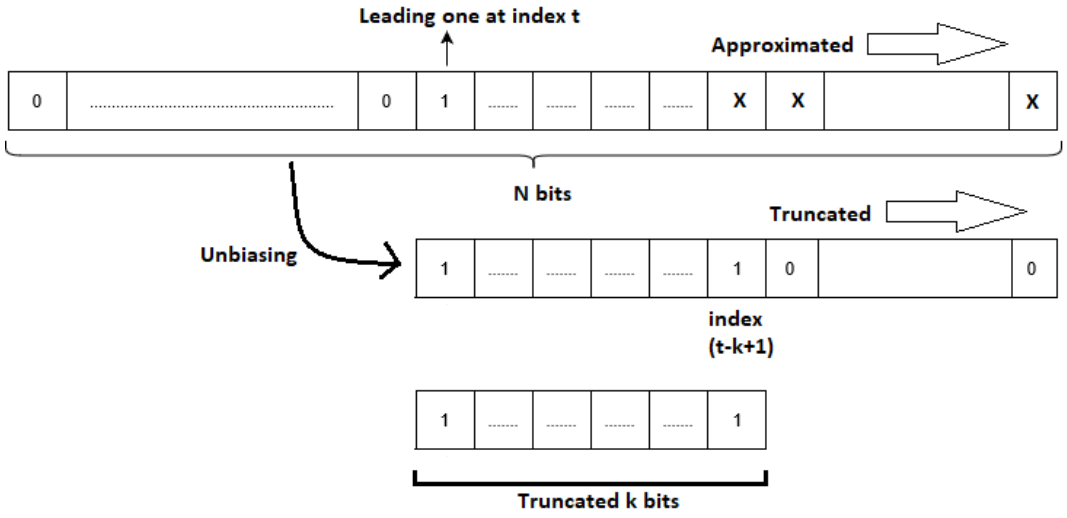


Figure 2.7: Truncation of operands to K-bits

Finally, the $2k$-bit result of this core multiplication is shifted to generate the final approximate result of $2n$ bits. This bit-width reduction scheme reduces the size of the multiplier to be used while introducing error in the final multiplication result. The multiplier has been further qualified to handle signed numbers by converting the signed operands to unsigned before forwarding them to the core multiplier. The sign for the result is then calculated separately by the sign prediction logic and the output is negated if necessary. The hardware design is composed of LOD blocks, encoders, multiplexers, barrel shifter and a core multiplier, seen in the Fig. 2.8.

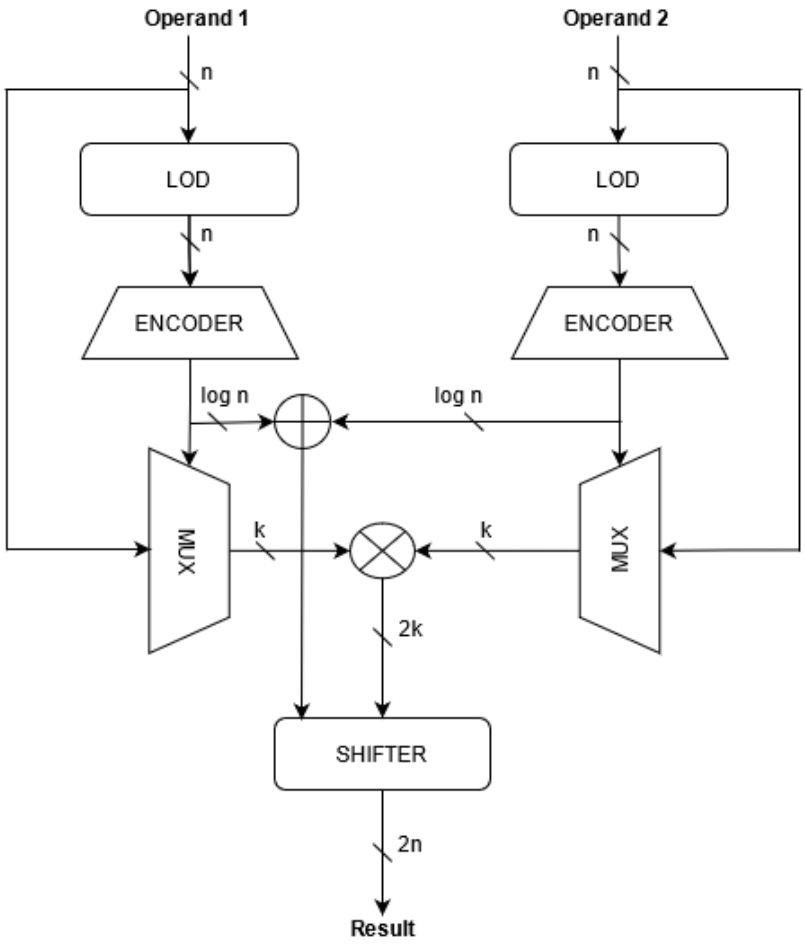


Figure 2.8: DRUM Multiplier

While there are a plethora of available approximate multipliers to choose from, DRUM was chosen because of its easy scalability. The architecture also allows use of preferred designs for the core multiplier. While its routing logic grows with the input size, the arithmetic logic does not need to grow to higher bandwidths to maintain computational accuracy, which reduces the increase in the area and power costs, making the achieved benefits more substantial.

Chapter 3

Approximation Methodology for Harris Corner Detection

From the baseline implementation of HCD, it can be observed that the bit-width increases due to the operations involved at each pipeline stage. In addition, the high number of multiplications result in increased computational complexity. Several previously reported attempts have aimed to optimize the inter-stage bit-widths via truncation or rounding-off during run-time. However, none have reported the use of an approximate multiplier. In this section, we aim to approximate HCD at the hardware level by dynamically changing the precision of intermediate multiplier inputs using the DRUM multiplier. We used fixed-point data type representation for the data produced and consumed at various stages of the pipeline, fixing the integral and fractional bit-widths to 10 and 6 respectively.

3.1 Implementation of Pipelined HCD

The HCD algorithm requires two kernels to perform convolution with the image pixels. There is the Horizontal Kernel X and Vertical Kernel Y:

(a) Convolution with X Kernel

(b) *Convolution with Y kernel*

Figure 3.1: Convolution with image

Image pixels from the test bench are sent to the main HCD module. Inside the HCD module a line buffer stores the values of the image pixels. The size of the line buffer is set to 3*256. This makes sure that the convolution with the kernels is stalled until atleast three rows of pixel data has been sent to the module. Once the required data is available in the line buffer, the convolution starts operating on each pixel. The data is fetched from the line buffer in tiles of size 3*3 and convolved with kernels X and Y. The results of each convolution are further processed to compute the response function. The comparison of the response function with the threshold records corner pixels with value 255 and non corner pixels with value 0. This data is then sent to the output stream. The quality of corner detection depends on the threshold selected to qualify the corner pixels. Upcoming section will discuss the process of threshold selection in detail.

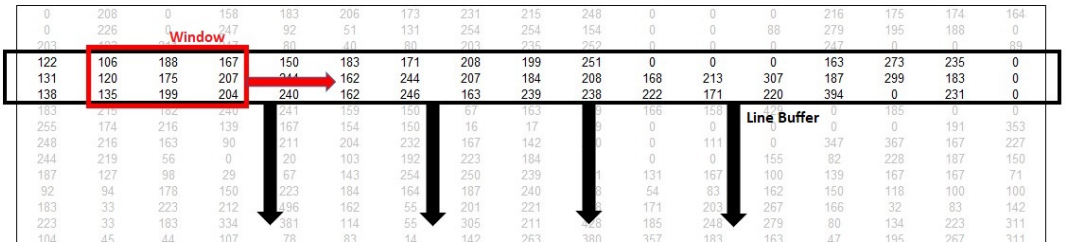


Figure 3.2: Pixels of an image processed using Line Buffer and Kernel Window

3.2 Proposed Approximation Strategy for HCD

The DAG from Fig. 2.1 makes use of both convolution and pointwise operations. The multiplications in these computations are independently targeted for approximation by using DRUM of different bit-width (k). Table 3.1 shows the computations at each stage of the DAG and the operations involved.

Operands	**Operation**	**Stage Output**
I, K_{sobel}	Convolution	I_x, I_y
I_x, I_y	Pointwise Multiplication	I_{xx}, I_{yy}, I_{xy}
I_{xx}, I_{yy}, I_{xy}, K_{gauss}	Convolution	S_{xx}, S_{yy}, S_{xy}
S_{xx}, S_{yy}, S_{xy}	Pointwise Multiplication	R

Table 3.1: Operations involved at each stage of HCD

To obtain the response R for a pixel P (bit-width 8) requires close to 50 multiplication operations, 45 of which are for the convolution stages alone. This heavily skews the balance of inaccuracies resulting from the two categorical operations towards the convolutions.

3.3 Operational Error Propagation

Analyzing the different operations involved in the HCD algorithm, we derived the equations for error in their outputs. In Table 3.2, errors resulting from inaccurate convolution (C) and pointwise multiplication (M) are used to derive the error propagation from one HCD stage to the next. To this end we applied the statistical error analysis for basic arithmetic operations and added the mean error due to approximate DRUM multiplications on top.

Operation	**Operands**	**Error**
C	I, K_{sobel}	$\Delta C = \sum_{K_s} \lvert K_i \rvert . \Delta I_i + \sum_{K_s} e_m$
M	I_x, I_y	$\Delta M = I_x . \Delta I_y + I_y . \Delta I_x + e_m$
C $\rightarrow$ **M**	I_x, I_y	$\Delta M' = \Delta C . (I_x + I_y) + e_m$
M $\rightarrow$ **C**	I_{xx}, K_{gauss}	$\Delta C' = \sum_{K_g} \Delta M' + \sum_{K_g} e_m$

Table 3.2: Error Propagation due to Approximated Operations

In addition to the operational analysis, error introduced by the DRUM multiplier was also considered. Fig. 3.3 plots the mean multiplication error e_m as a function of DRUM bit-width k, constrained between the maximum and minimum limits as discussed in [6].

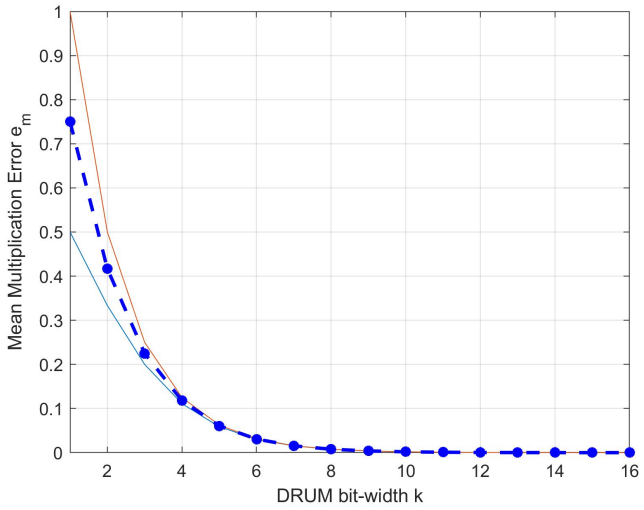


Figure 3.3: Mean Multiplication Error for different values of k

The trend shows an increase in e_m for bit-widths lower than 8, thereby hinting at output quality degradation. This observation combined with the equations in Table 3.2, imply that error introduced due to convolution (using k_1 bit-width DRUM) strongly impacts later stage pointwise multiplications (using k_2 bit-width DRUM). As we go across the HCD pipeline, the transition from convolution to multiplication occurs twice whereas the transition from multiplication to convolution occurs only once.

Therefore, a high ΔC due to lower k_1 will create an error compounding effect. On the other hand, lower k_2 only produces an additive effect. Aggressively targeting convolutions can therefore become counter-productive as bit-width k_1 is drastically reduced. This called for moderation in bit-width reduction and cemented our intuition of keeping k_1 higher than k_2 to prevent undesirable quality loss, the validity of which is confirmed in the experimental results section.

The proposed approximation strategy is applied and results with varying accuracy profile across different k_1, k_2 combinations are obtained. It is observed that subjecting the inaccurate R to a pre-determined empirical threshold (typically a large positive value) causes large number of false corner candidates to pass on to the NMS stage, where pseudo corners and corner clusters manifest. This threshold, which normally varies from image to image, is found to be inadequate for varying accuracy results for the same image. Thus, requiring a revised approach to its selection.

3.3.1 Modified Response Function and Threshold Selection

When the response function obtained from approximated HCD setups was treated with a single threshold (roughly estimated via trial and error), it resulted in large number of false corners as seen in the figures of Table 3.3.

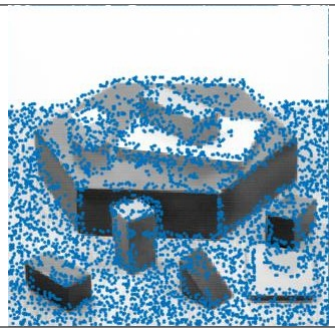
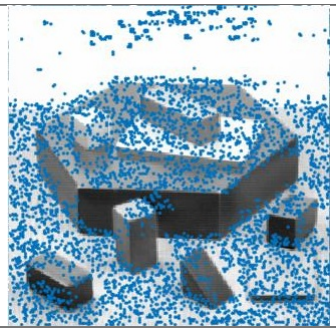
HCD(15,15)	HCD(8,8)
	

Table 3.3: Corners with arbitrarily large threshold

This further becomes evident from the visualization of the response function values in Fig. 3.4.

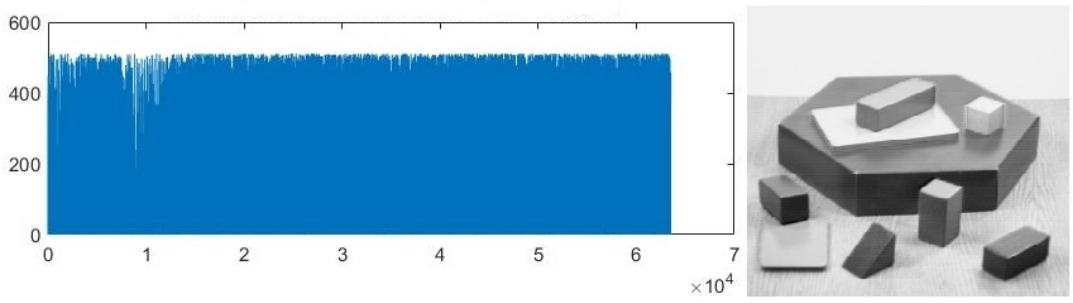


Figure 3.4: Original Response Function of Objects Image from HCD(8,8)

In the current work we propose a method based on further manipulation of the response function R. We defer the comparison of R with the threshold and perform a stencil computation instead. The R is subjected to weighted manipulation of its neighborhood values using a 3×3 stencil centered on the current value. The corner values are given a weight -1 and the immediate neighbors are given the weight 1. This operation changes the R from equation (2.4) to the modified response function R_M in equation (3.1).

C_i	N_i	C_i
N_i	R_i	N_i
C_i	N_i	C_i

Figure 3.5: Pixel neighbours and corners

$$R_M = R_i + \sum_{x,y \in N_i} R(x,y) - \sum_{x',y' \in C_i} R(x',y') \tag{3.1}$$

where, R is the original response function value for a given pixel, $R(x,y)$ are the response function values for pixels at neighboring locations and $R(x',y')$ are response function values of corner pixels. Fig. 3.6 illustrates the change from original R to R_M.

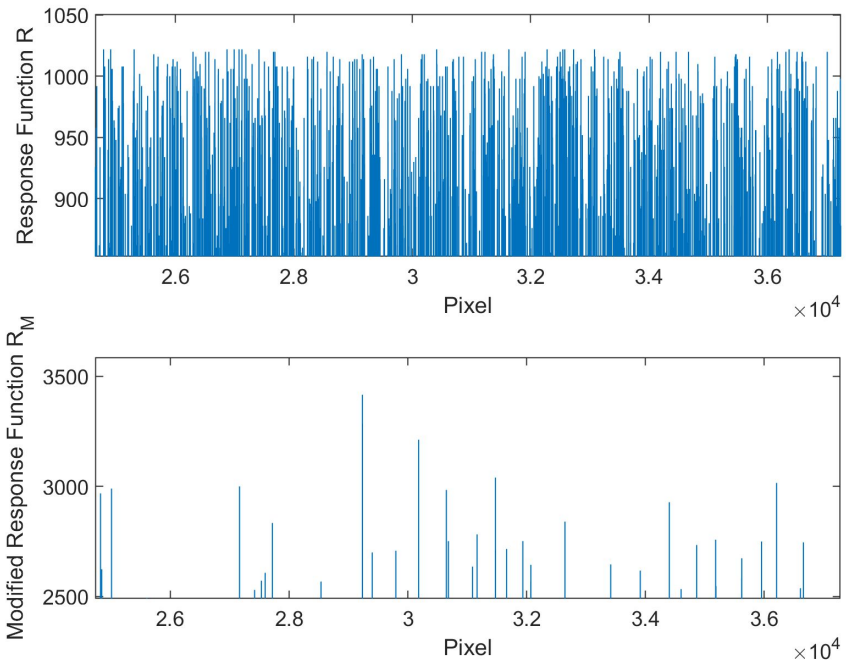


Figure 3.6: Comparison between the R and R_M for Block image from HCD(8,8)

Where earlier the use of a coarsely determined threshold did not differentiate weak corners from strong noisy points, the proposed kernel operation makes a marked difference in their R_M values making it much easier to nail down an optimum threshold value. Since this operation is performed using an accurate multiplier, the error ΔR_M computes to equation (3.2) which doesn't add any error on top of what has already propagated to the R values used in equation (3.1).

$$\Delta R_M = \sqrt{\sum_i (\Delta R_i)^2} \tag{3.2}$$

The modified DAG with R_M update can be seen in the Fig. 3.7.

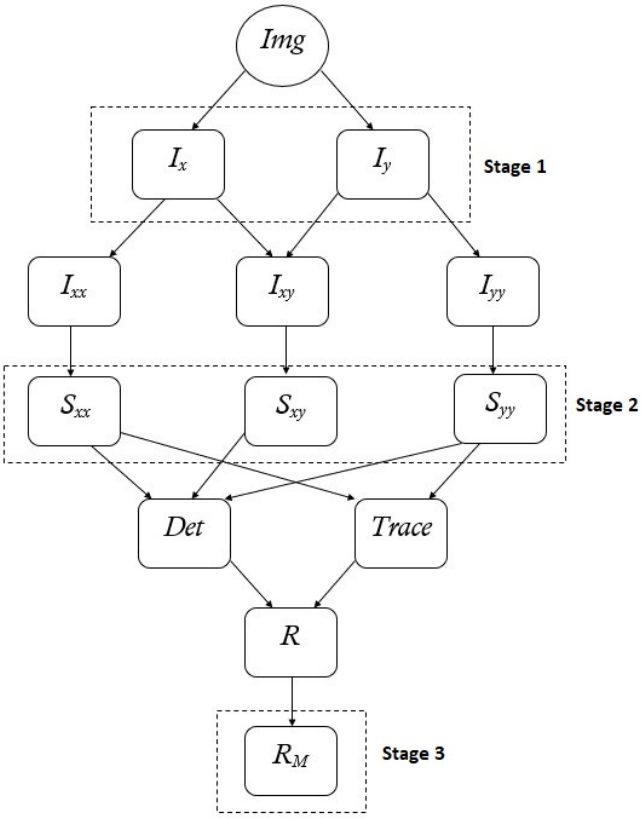


Figure 3.7: DAG representation of modified HCD algorithm

There are clear advantages to our approach. Since the R is specific for a given image, the proposed approach is able to adapt to different images. Due to effective thresholding of R_M values, the number of corner clusters to be processed by non-maximum suppression is also drastically reduced.

3.4 Experimental Results and Comparative Analysis

In this section, we will provide experimental results and performance evaluations for the proposed HCD implementation. We use fixed-point data type representation for the data produced and consumed at various stages of the pipeline, fixing the integral and fractional bit-widths to 10 and 6 respectively. The inbuilt multiplier function available in the BSV environment is used for the baseline implementation of original HCD algorithm. We examine the impact of changing the k_1, k_2 on the output of the proposed implementation. We also evaluate the design in terms of resource, power and speed of operation. All characteristics are reported post Place and Route.

3.4.1 Comparative works

For comparison purposes, we have selected two HCD implementations on the Avnet Zedboard device explored in [4] and [13].

[4] explores the Harris algorithm, eliminating the repeated corner features in a local area using a binary feature window. Once a feature candidate is found, it then propagates in a binary feature window which disregards any candidates that come later. Since the classical non-maximum suppression method requires a window of Harris response scores, utilizing a binary window largely reduces the hardware resources required.

[13], on the other hand, focuses on reducing the logic resource consumption of HCD algorithm by performing fixed-point computations on optimal bit-width arguments. Additionally, the Harris response of feature candidates is set to 255 and those that do not qualify are set to 0. Unlike the traditional non-maximum suppression using a matrix of Harris response value, using 255 and 0 reduces logical resource required and does not affect the performance of the algorithm. This implementation also allows setting multiple thresholds in order to control the corner output.

The challenge of eliminating the repeated corners in a local pixel space, as discussed in [4], is akin to the challenge faced in approximate HCD setup to refine the corner candidates using modified response function. Making this modification a part of the pipeline ensures that our algorithm doesn't need any extra delay time. Additionally, our algorithm focuses on finding the optimal DRUM bit-width for operands to meet the required accuracy, in contrast to the bit-width optimization of arguments discussed in [13].

3.4.2 Resource Utilization

The baseline and proposed HCD architectures were implemented using Bluespec SystemVerilog (BSV) and synthesized on Zynq-7000 and Xilinx Virtex-7 FPGA devices using Xilinx Vivado 2018.3 ISE software. Table 3.4 presents resource costs for different HCD setups on the chosen devices. To enhance the speed of our implementation, we used pipelined line buffers alongside the pipelined DRUM instances and the associated routing logic. This contributed to increased Look-Up Table (LUT) and Flip Flop (FF) utilization compared to the baseline. However, in contrast to the baseline and the approaches in [4] and [13] for 640x480 images, our approximate design exhibits nil BRAM and DSP usage and achieves a better throughput even at reduced accuracy as seen in the next section.

Device	k_1, k_2	$LUTs$	FF	$BRAM$	DSP
Viretx-7 xc7v585ttfg1157-2	8,8	18.220	20.151	0	0
	8,15	26.965	20.187	0	0
	10,10	24.910	19.090	0	0
	15,8	27.463	19.476	0	0
	15,15	36.105	19.514	0	0
	Baseline	5.078	2.165	0.755	4.127
Zynq-7000 xc7z045ffv900-2	8,8	34.612	22.576	0	0
	8,15	41.877	23.351	0	0
	10,10	37.961	23.318	0	0
	15,8	43.740	23.444	0	0
	15,15	50.355	23.508	0	0
	Baseline	8.462	3.607	1.101	5.778
xc7z020-clg484-1	[4]*	17.82	3.88	7.246	0
	[4]**	1.836	0.622	7.246	50.00
	[13]	33.00	2.20	53.57	25.00

*without DSP. **with DSP.

Table 3.4: Percentage Resource Utilization for HCD on Virtex-7 and Zynq-7000

3.4.3 Power and Timing Analysis

To emulate real design behavior in a software environment, synthesis and implementation was further qualified by performing simulation in the Xilinx Vivado tool. The SAIF (Switching Activity Interchange Format) from the simulator was back-annotated into the Xilinx power analysis and optimization tools for the power measurements and estimations. Table 3.5 below captures the power, maximum speed and throughput figures for different HCD setups synthesized and implemented on Virtex-7 and Zynq-7000 boards.

k_1,k_2	Virtex-7			Zynq-7000		
	$Power$	F_{max}	$Throughput$	$Power$	F_{max}	$Throughput$
8,8	1.344	211	46	1.094	209	46
8,15	1.608	203	42	1.511	205	45
10,10	1.430	209	44	1.132	206	45
15,8	1.802	204	42	1.418	203	44
15,15	2.145	200	41	1.966	200	43
Baseline	0.376	125	27	0.333	125	29
Comparative works	[4]*					30
	[4]**					44
	[13]					47

Table 3.5: Power(W), $F_{max}(MHz)$ and Throughput ($Megapixels/sec$) for HCD on Virtex-7 and ZYNQ-7000

As evident from the data, our proposed solution offers a significant speed boost compared to the baseline implementation. Additionally, Fig. 3.8 shows that our approximate architecture rivals the throughput performance from [4] and [13].

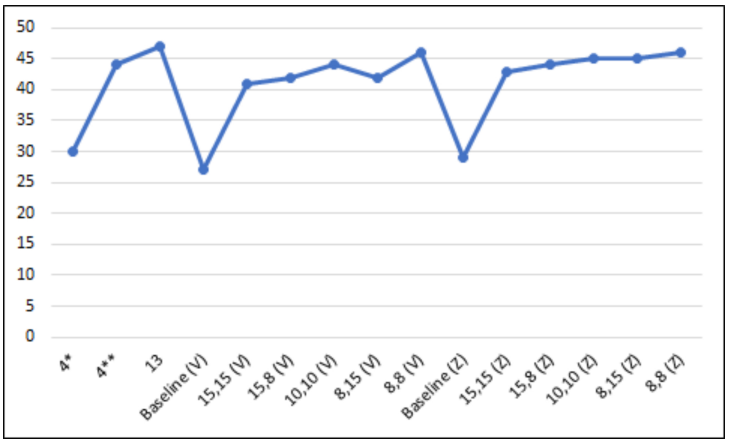


Figure 3.8: Throughput for HCD architectures on Virtex-7 and Zynq-7000

3.4.4 Output Quality Analysis

Evaluation of the proposed architecture is of paramount importance as the multiplier bit-width reduction affects the corner detection capability. The performance of different implementations on the selected grayscale test images is shown in Fig. 3.9, where the output exported form the BSV testbench has been plotted using MATLAB's image processing toolbox. The MATLAB (R2021a version) results indicate that the corner detection is increasingly impacted by the inaccuracy introduced.

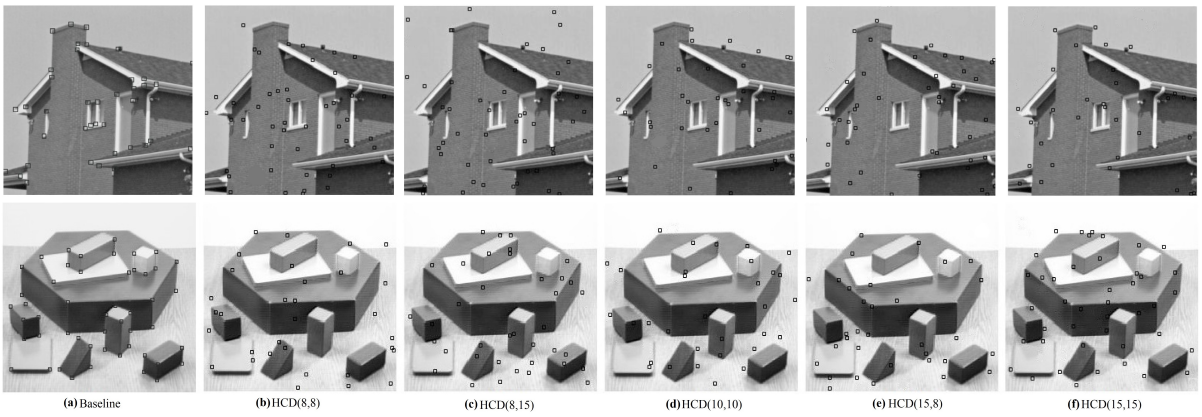


Figure 3.9: Corner Detection for House and Block image for baseline and approximate HCD implementations

The squares in the figure indicate corner features detected by the baseline and our proposed implementation of varying accuracy. It can be observed that the approximate result does not follow the baseline result with high fidelity. In fact, as predicted in Section III, with decreasing k_1, k_2 increase in false corners is observed. Also, we see more degradation in the quality of corners detected for lower k_1 setup. This can be confirmed by comparing results with complementary k_1, k_2 setting. With highly inaccurate convolutions accompanied by later stage error compounding, HCD(8,15) result in Fig. 3.9c reports higher count of false corners compared to HCD(15,8) result in Fig. 3.9e. Therefore, it can be rightfully expected that the performance will only get worse below bit-width 8. To verify this, we have adopted the Accuracy (ACU) criteria from [15] to quantify the observed performance variation across implementations.

$$ACU = 100 \times \frac{\frac{N_a}{N_o} + \frac{N_a}{N_g}}{2} \tag{3.3}$$

where N_o is the total number of corners detected by an HCD setup, N_g is the total number of corners in the ground truth and N_a is the total number of matched corners. Depending on the application requirement, lower than permissible ACU may deem the proposed architecture unsuitable below its corresponding bit-width. For the purpose of evaluating the ACU figure, only strong corners were considered as ground truth. Higher accuracy metric requires that matched corners should be detected as close as possible to these ground truth corners. To this end, Euclidean distance between them was computed and thresholded. If this distance crossed the threshold, the detected corner was not counted. Fig. 3.10 presents the ACU figure for HCD setups where both k_1 and k_2 were set to the same value.

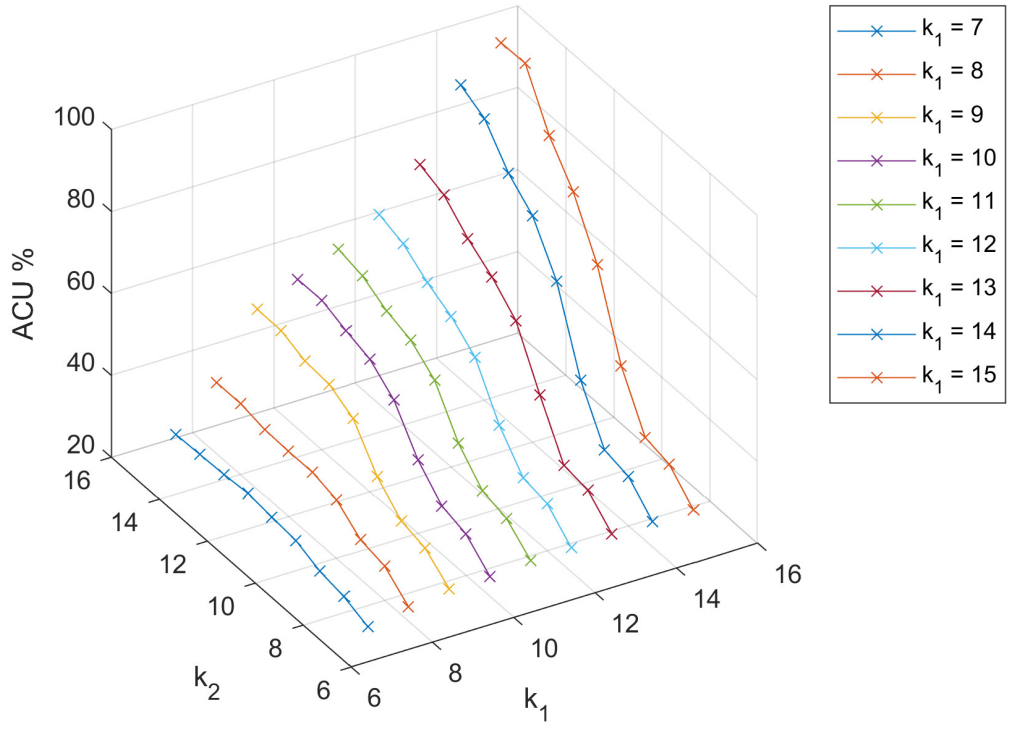


Figure 3.10: Accuracy for implemented HCD(k_1,k_2), from $k_1 = 7$ plot line on the left to $k_1 = 15$ plot line on the right

Looking at the different ACU plot lines, one can discern the bit-width range required to meet the needs of an application using our proposed design. The low ACU measure quantifying the increase in false corner detection, gives an idea about the expected degradation in performance at lower accuracy. For HCD setups with DRUM bit-widths less than 9, the unusual distribution of corner points is found to be of little use for feature identification.

Chapter 4

Approximation Methodology for Unsharp Masking

In this section, we aim to approximate USM algorithm at the hardware level by dynamically changing the precision of intermediate multiplier inputs using the DRUM multiplier. We used fixed-point data type representation for the data produced and consumed at various stages of the pipeline, fixing the integral and fractional bit-widths to 10 and 6 respectively.

4.1 Implementation of Pipelined USM

The USM algorithm can be visualized as a filter that amplifies the high-frequency components of an input signal. The algorithm comprises of four pipelined stages and was tested on images with the resolution of 1024 x 1024. The first two stages perform a Gaussian blur of the original input image along the x and the y directions subsequently. The kernel for blurring the original image across x-axis and y-axis directions can be seen below.

$$\begin{bmatrix} 0.04 & 0.04 & 0.04 \\ 0.04 & 0.04 & 0.04 \\ 0.04 & 0.04 & 0.04 \end{bmatrix}$$

Figure 4.1: Kernel used for blurring along both X and Y axes

Image pixels from the test bench are sent to the main USM module. Inside the module a line buffer stores the values of the image pixels. The size of the line buffer is set to 3*1024. This makes sure that the blur operation with the kernel is stalled until atleast three rows of pixel data has been sent to the module. Once the required data is available in the line buffer, it is fetched from the line buffer in tiles of size 3*3 and convolved with the blur kernel. Once the blurred data I_{blur} starts coming, it is then used to compute I_{diff} and I_{sharp}. The comparison of the I_{diff} data with the threshold decides which pixels would be substituted with their sharpened counterpart from I_{sharp}. The processed data is then sent to the output stream.

4.2 Proposed Approximation Strategy for USM

The DAG from Fig. 2.3 made use of both convolution and pointwise operations. The multiplications in these computations are independently targeted for approximation by using DRUM of different bit-widths : k_1 for convolutions and k_2 for pointwise multiplications. Table 4.1 shows the computations at each stage of the DAG and the operations involved.

Operands	**Operation**	**Stage Output**
I, K_{gauss}	Convolution	I_x
I_x, K_{gauss}	Convolution	I_y
I, W, I_y	Pointwise Multiplication and Subtraction	I_{diff}

Table 4.1: Operations involved at each stage of USM

To obtain the sharpened pixel for a pixel P requires close to 20 multiplication operations, 19 of which are for the convolution stages alone. This again heavily skews the balance of inaccuracies resulting from the two categorical operations towards the convolutions.

4.3 Operational Error Propagation

Analyzing the different operations involved in the USM algorithm, we derived the equations for error in the sharpened output. The blurring operation was found to require convolution operations and the subsequent stage of computing the weighted sum of the resultant blur was found to require pointwise multiplications. In Table 4.2, errors resulting from inaccurate convolution (C) and pointwise multiplication (M) are used to derive the error propagated to the final USM stage depending on I_{diff}-threshold comparison. Thus, the USM setup also uses two DRUM instances with bit-widths k_1 and k_2.

Operation	**Operands**	**Error**
C	I, K_{gauss}	$\Delta C = \sum_{K_g} \lvert K_i \rvert . \Delta I_i + \sum_{K_g} e_m$
C $\rightarrow$ **C**	I_x, K_{gauss}	$\Delta C' = \sum_{K_g} \Delta C + \sum_{K_g} e_m$
M	I, W	$\Delta M = \lvert W \rvert . \Delta I + e_m$
D	I, I_y	$\Delta D = \sqrt{(\Delta M)^2 + (\Delta C')^2}$

Table 4.2: Error Propagation due to Approximated Operations in USM

To classify a pixel P for sharpening effect requires close to 20 multiplication operations, 19 of which are for the convolution stages alone. This again heavily skews the balance of inaccuracies resulting from the two categorical operations towards the convolutions. As we go across the USM pipeline, the

transition from convolution to multiplication makes it evident that high values of ΔC and $\Delta C'$ due to lower k_1 create an error compounding effect at the I_{diff} computation stage. This creates the possibility of pixel mis-classification which can lead to overshoot artefacts or noise amplification. Once again we arrived at the same conclusion, that of keeping k_1 higher than k_2 in order to prevent undesirable quality loss in the sharpening effect, the validity of which is confirmed in the experimental results section.

This approximation strategy was applied and results with varying accuracy profile across different k_1, k_2 combinations were obtained. Just like HCD, USM output quality was found to depend on the threshold value used for pixel classification. The approximations introduced due to varying DRUM bit-widths make the threshold choice even more critical. The effect of different threshold values becomes obvious from the "sharpened" images shared in the Fig. 4.2.

Figure 4.2: Sharpened outputs from USM(15,15) with different threshold values (A $<$ B $<$ C). From left to right : Original, Sharp(A), Sharp(B), Sharp(C)

From these images it becomes clear that the threshold in USM cannot be arbitrarily chosen. An extremely low or high value of threshold results in undesirable image deterioration. Therefore, threshold is probably the most important control to get right. Recalling the equation 2.8, it becomes imperative to find the optimum threshold value. In simple terms, the purpose of the threshold is to dictate how different two pixels need to be for sharpening to be applied. It is rather pointless to sharpen each pixel in the image. This calls for threshold adjustment where the user aims to find the balance between detail and noise. A threshold low enough to sharpen the important details in the image, but high enough not to sharpen any unwanted noise, somewhat like Sharp(B) from Fig. 4.2.

4.3.1 Adaptive Threshold Selection

In search for the optimum threshold, the approximate USM setups were subjected to a range of threshold values. To quantify the visual perception of resulting sharpening effect, Structural Similarity Index (SSIM) metric for each of the "sharpened" outputs was computed with respect to the original input image. This metric assesses the degradation of structural information in the sharpened image in comparison with the original. This information is of interest to us as we wish to avoid degraded outputs with "sharpened" noise elements.

$Threshold$	$SSIM$	$Threshold$	$SSIM$
10	0.60	60	0.89
20	0.61	70	0.86
30	0.63	80	0.82
40	0.72	90	0.79
50	0.85	100	0.76

Table 4.3: SSIM variation with different thresholds for USM(10,10) setup

Threshold 10	**Threshold 20**	**Threshold 30**	**Threshold 40**	**Threshold 50**
				
Threshold 60	**Threshold 70**	**Threshold 80**	**Threshold 90**	**Threshold 100**
				

Table 4.4: Sharpened outputs from different threshold values

This SSIM data gives the user an idea about the permissible threshold range to be chosen for a given image. In case an erroneously large or small threshold is chosen and the SSIM of the sharpened output is found to be unacceptable, the threshold can be **adapted** to a more suitable value adhering to the required performance standards.

From the data in table 4.3 and its visual representation in table 4.4, it becomes obvious that a threshold in the range $50 < T < 70$ offers the best chance at getting an aesthetically pleasing effect from sharpening. The best thing about this approach is that it works well for USM setups with varying de-

grees of approximation. The USM pipeline making use of the SSIM check for adaptive threshold update, visualized as a directed acyclic graph (DAG) of computations, is illustrated in Fig. 4.3.

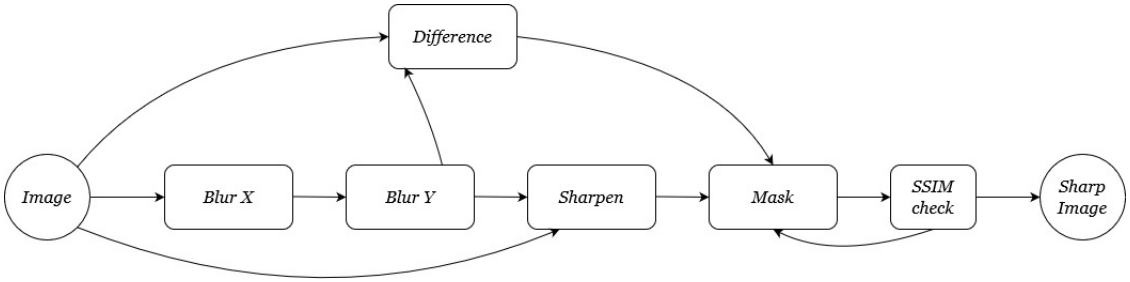


Figure 4.3: DAG representation of the Unsharp Mask (USM) algorithm for adaptive threshold selection

Seen in the figures below, is the varying degrees of sharpness produced by the USM setups (with different DRUM bit-widths) when subjected to an optimum threshold value.

Figure 4.4: Sharpened outputs for man image. From left to right : USM(15,15), USM(15,8), USM(10,10), USM(8,15), USM(8,8)

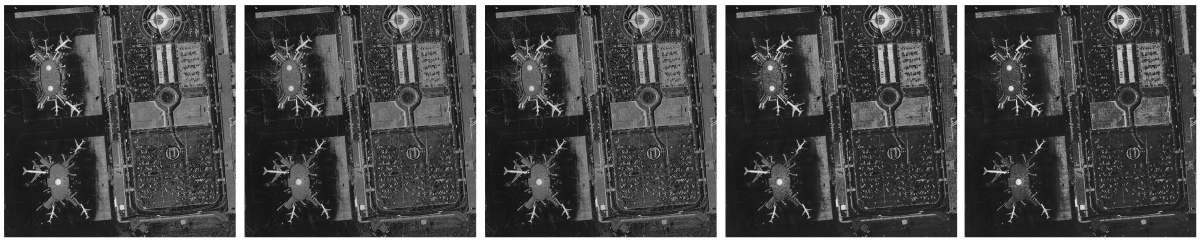


Figure 4.5: Sharpened outputs for airport image. From left to right : USM(15,15), USM(15,8), USM(10,10), USM(8,15), USM(8,8)

Where earlier the use of a coarsely determined threshold led to overshoot artefacts, the proposed SSIM based adaptive approach keeps these in check while adhering to output quality requirements. Fig. 4.4 and 4.5 shows the varying degrees of sharpness produced by the USM setups (with different DRUM bit-widths) when subjected to an optimum threshold value. There are clear advantages to our approach. The proposed approach is able to adapt to different images. Furthermore, due to effective thresholding the characterization of pixels for sharpening effect is more robust.

4.4 Experimental Results and Comparative Analysis

In this section, we will provide experimental results and performance evaluations for the proposed USM implementation. We use fixed-point data type representation for the data produced and consumed at various stages of the pipeline, fixing the integral and fractional bit-widths to 10 and 6 respectively. The inbuilt multiplier function available in the BSV environment is used for the baseline implementation of original USM algorithm. We examine the impact of changing the k_1, k_2 on the output of the proposed implementation. We also evaluate the design in terms of resource, power and speed of operation. All characteristics are reported post Place and Route.

4.4.1 Resource Utilization

Table 4.5 presents resource costs for the baseline and proposed USM architectures on the Zynq-7000 and Xilinx Virtex-7 FPGA devices. To enhance the speed of our implementation, we used pipelined line buffers alongside the pipelined DRUM instances and the associated routing logic. This contributed to slight increase in LUT and FF utilization compared to the baseline. However, in contrast to the baseline implementation, our approximate design exhibits nil DSP usage and achieves a better throughput even at reduced accuracy. Thus, striking a good balance between performance and resource utilization.

Device	k_1, k_2	$LUTs$	FF	$BRAM$	DSP
Zynq-7000 xc7z045ffv900-2	8,8	7.301	3.341	0.642	0
	8,15	9.105	3.882	0.642	0
	10,10	8.712	3.782	0.642	0
	15,8	9.230	3.985	0.642	0
	15,15	10.133	4.085	0.642	0
	Baseline	1.555	0.798	0.642	1
Viretx-7 xc7v585ttfg1157-2	8,8	5.951	3.167	0.440	0
	8,15	7.102	3.177	0.440	0
	10,10	6.106	3.162	0.440	0
	15,8	7.132	3.189	0.440	0
	15,15	7.374	3.194	0.440	0
	Baseline	0.933	0.479	0.440	0.714

Table 4.5: Percentage Resource Utilization for USM on Virtex-7 and Zynq-7000

4.4.2 Power and Timing Analysis

Table 4.6 below captures the power and maximum speed figures for different USM setups synthesized and implemented on Virtex-7 and Zynq-7000 boards. The SAIF file from the simulator was back-annotated into the Xilinx power analysis and optimization tools for the power measurements and estimations.

k_1, k_2	Virtex-7			Zynq-7000		
	$Power$	F_{max}	$Throughput$	$Power$	F_{max}	$Throughput$
8,8	0.450	261	47	0.498	265	59
8,15	0.630	253	41	0.612	255	55
10,10	0.556	257	43	0.503	261	56
15,8	0.651	250	39	0.607	250	52
15,15	0.762	250	39	0.715	250	51
Baseline	0.279	125	27	0.235	125	36

Table 4.6: Power(W), $F_{max}(MHz)$ and Throughput ($Megapixels/sec$) for USM on Virtex-7 and ZYNQ-7000

4.4.3 Output Quality Analysis

In corner detection, we have a given number of ground truth corners for an image and approximate HCD setups are qualified using ACU metric on the basis of matched true corners. But in USM, we do not have such ground truth "sharpened" state against which we can qualify the approximate sharpened outputs. This in turn made us focus on the sharpening of wrong pixel candidates. Inaccuracy in computing a pixel value from the sharpen stage resulted in mis-classification during the final stage of the USM DAG when subjected to optimum threshold.

To quantify this, we relied on an error metric that is the percentage of pixels that were mis-classified. For accuracy estimation in USM setups, this mis-classification index (MI) was evaluated with respect to the output from golden setup using inbuilt multiplier function. All approximated USM outputs were computed using the optimal threshold.

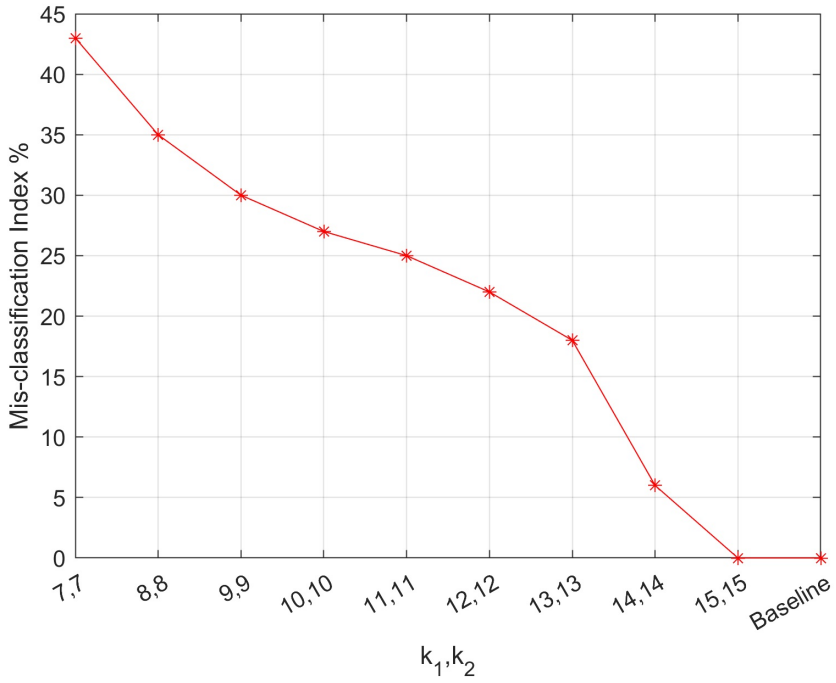


Figure 4.6: MI% for implemented USM(k_1,k_2)

As expected, the MI % turns out to be close to 0 for high fidelity outputs and increases gradually with the bit-width reduction. Permissible MI of 30-40% allows the user to reduce the bit-width of the DRUM to 8, thus reducing power and resource overhead while increasing the speed of operation.

Chapter 5

Conclusions

In this thesis, we have presented an approximated pipelined FPGA architecture to implement HCD and USM algorithms. The approximation has been realized using an approximate multiplier, with the aim to achieve desired performance levels from highly accurate to tolerably inaccurate.

The derived mathematical relationships between the pipeline operations and their statistical errors offered a robust analytical framework, independent of the application under consideration. As a result, this analysis can be applied to other algorithms such as Lucas Kanade and Gaussian Pyramid, whose equations 2.13 and 2.15 bear resemblance to the matrix computations for HCD (2.3).

The usage of DRUM offers easy scalability for higher input bit-widths. And the approximation strategy can be changed by using a different core multiplier inside DRUM or targeting a different operation (for instance, addition). The proposed approximate architecture utilized less than 50% of the resources of the Xilinx Virtex-7 and Zynq-7000 FPGA platforms. The usage of pipelined line buffers and pipelined DRUM instances contributed to increased LUT and FF utilization compared to the baseline implementation. However, our approximate design achieved good balance with the improved speed and throughput even at lower accuracy.

With a system clock of 200MHz and 250MHz at their most accurate setting, the approximate HCD and USM designs offered increasing F_{max} with decreasing accuracy. Even at lower accuracy levels our HCD design outperformed [4] in terms of throughput and matched the performance of [13]. The performance was found to be sufficient for coarse object localization and tracking. When compared with analogous accurate implementations of HCD on other FPGA devices, the BRAM and DSP usage of our design was found to be the least. Similarly, the USM approximate design when compared with the base implementation, reported nil DSP usage. The design outperformed the baseline implementation in terms of speed and throughput with only slight increase in power.

Thus, the proposed HCD and USM designs with certain accuracy configurations achieved the best trade-off in accuracy, speed and power. Synthesis results show that the proposed implementation achieves over 60% increase in maximum frequency compared to the base implementation. The HCD and USM architectures were further qualified using application specific metrics such as the ACU and MI respectively. The analytical results showed that these metrics can be instrumental in determining the permissible bit-widths for the DRUM instances at different pipeline stages to achieve desired output quality.

Related Publications

Following paper has been accepted as a poster paper in the Student Research Track of IEEE Computer Society Annual Symposium on VLSI

- Shivani Maurya, Ziaul Choudhury, Suresh Purini. ”Accuracy Configurable FPGA Implementation of Harris Corner Detection” (ISVLSI, July 2022)

Bibliography

[1] A. B. Amara, E. Pissaloux, R. Grisel, and M. Atri. Zynq fpga based memory efficient and real-time harris corner detection algorithm implementation. In *2018 15th International Multi-Conference on Systems, Signals & Devices (SSD)*, pages 852–857. IEEE, 2018.

[2] M. F. Aydogdu, M. F. Demirci, and C. Kasnakoglu. Pipelining harris corner detection with a tiny fpga for a mobile robot. In *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 2177–2184. IEEE, 2013.

[3] M. A. Badamchizadeh and A. Aghagolzadeh. Comparative study of unsharp masking methods for image enhancement. In *Third International Conference on Image and Graphics (ICIG'04)*, pages 27–30. IEEE, 2004.

[4] T. L. Chao and K. H. Wong. An efficient fpga implementation of the harris corner feature detector. In *2015 14th IAPR International Conference on Machine Vision Applications (MVA)*, pages 89–93. IEEE, 2015.

[5] C. Harris, M. Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[6] S. Hashemi, R. I. Bahar, and S. Reda. Drum: A dynamic range unbiased multiplier for approximate applications. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 418–425. IEEE, 2015.

[7] X. He, S. Jiang, W. Lu, G. Yan, Y. Han, and X. Li. Exploiting the potential of computation reuse through approximate computing. *IEEE Transactions on Multi-Scale Computing Systems*, 3(3):152–165, 2016.

[8] A. Hernandez-Lopez, C. Torres-Huitzil, and J. J. Garcia-Hernandez. Fpga-based flexible hardware architecture for image interest point detection. *International Journal of Advanced Robotic Systems*, 12(7):93, 2015.

[9] J. Klippenstein and H. Zhang. Performance evaluation of visual slam using several feature extractors. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1574–1581. IEEE, 2009.

[10] S.-K. Lam, R. K. Bijarniya, and M. Wu. Lowering dynamic power in stream-based harris corner detection architecture. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 176–182. IEEE, 2017.

[11] X. Lanchi, G. Jingjing, and L. Zhihui. A novel unsharp mask sharpening method in preprocessing for face recognition. In *2015 Fifth International Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC)*, pages 378–381. IEEE, 2015.

[12] C.-L. Lin and C.-Y. Su. Modified unsharp masking detection using otsu thresholding and gray code. In *2016 IEEE International Conference on Industrial Technology (ICIT)*, pages 787–791. IEEE, 2016.

[13] S. Liu, C. Lyu, Y. Liu, W. Zhou, X. Jiang, P. Li, H. Chen, and Y. Li. Real-time implementation of harris corner detection system based on fpga. In *2017 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 339–343. IEEE, 2017.

[14] A. Mehta, S. Maurya, N. Sharief, B. M. Pranay, S. Jandhyala, and S. Purini. Accuracy-configurable approximate multiplier with error detection and correction. In *TENCON 2015-2015 IEEE Region 10 Conference*, pages 1–4. IEEE, 2015.

[15] F. Mokhtarian and F. Mohanna. Performance evaluation of corner detectors using consistency and accuracy measures. *Computer Vision and Image Understanding*, 102(1):81–94, 2006.

[16] A. M. Moubark, T. M. Carpenter, D. M. Cowell, S. Harput, and S. Freear. New improved unsharp masking methods compatible with ultrasound b-mode imaging. In *2017 IEEE International Ultrasonics Symposium (IUS)*, pages 1–4. IEEE, 2017.

[17] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback. A multi-resolution fpga-based architecture for real-time edge and corner detection. *IEEE Transactions on Computers*, 63(10):2376–2388, 2013.

[18] N. Ramakrishnan, M. Wu, S.-K. Lam, and T. Srikanthan. Automated thresholding for low-complexity corner detection. In *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 97–103. IEEE, 2014.

[19] E. Rosten, R. Porter, and T. Drummond. Faster and better: A machine learning approach to corner detection. *IEEE transactions on pattern analysis and machine intelligence*, 32(1):105–119, 2008.

[20] Z. Tang, Z. Ding, R. Zeng, Y. Wang, J. Wen, L. Bian, and C. Yang. Multi-threshold corner detection and region matching algorithm based on texture classification. *IEEE Access*, 7:128372–128383, 2019.

[21] T. Tuytelaars and K. Mikolajczyk. *Local invariant feature detectors: a survey*. Now Publishers Inc, 2008.

[22] W. Ye and K.-K. Ma. Blurriness-guided unsharp masking. *IEEE Transactions on Image Processing*, 27(9):4465–4477, 2018.