Real-Time Color TreeBASIS Feature Matching on a Limited-Resource Hardware System

Garrett Sean Hartman
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd
Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/4002

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Real-Time Color TreeBASIS Feature Matching on a Limited-Resource Hardware System

Garrett S. Hartman

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Doran K. Wilde, Chair
James K. Archibald
Dah-Jye Lee

Department of Electrical and Computer Engineering
Brigham Young University
October 2013

Copyright © 2013 Garrett S. Hartman
All Rights Reserved
ABSTRACT

Real-Time Color TreeBASIS Feature Matching on a Limited-Resource Hardware System

Garrett S. Hartman
Department of Electrical and Computer Engineering, BYU
Master of Science

This research has been conducted in order to create a robust, light-weight feature detecting and matching algorithm that builds upon the foundation set by the TreeBASIS algorithm. The goal is to create a color-based version of the TreeBASIS algorithm that uses less hardware resources than the original, is more accurate in its matching capabilities, can successfully be deployed on a resource-limited FPGA platform, and can process in real time.

This thesis first presents the newly designed hardware tri-channel FAST Feature Detector that finds features in color. Next the TreeBASIS algorithm is analyzed to discover what improvements can be made in order to reduce its resource usage sufficiently to be able to run on the Xilinx Virtex-4 FX60 while processing color features. At the same time, a software version of the Color TreeBASIS algorithm is compared to the original algorithm and is found to have a 93.3% accuracy on a test set of aerial images, surpassing the accuracy of TreeBASIS by nearly 12%. Then the hardware is meticulously reviewed to discover even more optimizations that allow the Color TreeBASIS algorithm to easily fit onto the Virtex-4 FX60. Next a new application for the matching algorithm, object detection, is introduced as well as the hardware needed to support it. Finally the algorithm is tested on the FPGA system for object detection and is able to successfully identify objects at 60 FPS.

Color TreeBASIS proves itself to be more accurate than the TreeBASIS algorithm in the aerial images tests, it ends up using less memory and logic resources than its predecessor, even though it processes three times as much data, it is successfully deployed on a resource-limited FPGA system, and it shows accurate results in real-time object identification, generating an accurate homography 20 to 45% of the time while processing matches at a rate of 60 FPS.

Keywords: feature detection, feature matching, low-resource, limited-resource, FPGA, computer vision, TreeBASIS, color, hardware
ACKNOWLEDGMENTS

I would like to thank all of those that have helped me on this arduous journey towards the completion of my thesis. The road was long and rough, and though the end seemed prohibitively distant at times, we made it.

I first want to thank my wife, Gabriela, for all of the support and motivation she has given me. Although I started this adventure without her, it is because she entered my life that I was able to get things truly moving. Thank you, my love.

Next I would like to thank my parents, John and Debbie, for always believing in me and being supportive and for letting me bounce ideas off of them, even if the ideas made no sense. Thanks for being awesome parents!

Dr. Wilde. If it weren’t for you, who knows what would have happened! Thank you for pointing me in the right direction for my research and for being supportive even when progress was slow. I also greatly appreciate our extra-long discussion sessions. Many great ideas were formed and later executed because of them. Once again, thanks.

Thanks are in order for Spencer Fowers, the creator of the TreeBASIS algorithm. Your help getting me familiar with the Helios platform and your TreeBASIS algorithm made my research possible. Thank you.

I would like to thank Dr. Archibald and Dr. Lee for their patience and willingness to help move along the process of publishing and presenting this thesis. It wasn’t the ideal situation, but I am forever grateful for your helpfulness and flexibility. Many thanks!

I am grateful for my best friend, Brian, who frequently took interest in my research and gave me a chance to discuss my successes and failures. Thanks, bro.

Lastly, I would like to thank my mother-in-law, Iracema, for all of her support, her prayers, her fasts, and for giving me the best wife in the whole universe. I am in your debt.
For any of those not mentioned here, and all of those who are, I want to thank you for all of your help and your prayers. I have learned so much through this experience and your loving support has made this research worthwhile. Thank you.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
</tbody>
</table>

## Chapter 1 Introduction

1

## Chapter 2 Features by Accelerated Segment Test (FAST)

5

2.1 Hardware FAST

2.1.1 Start-up Stage

2.1.2 Corner Identification and Scoring

2.1.3 Non-maximal Corner Suppression

2.1.4 Clean-up Stage

2.2 Adding Dominant Rotation Orientation to FAST

14

## Chapter 3 Color TreeBASIS Feature Matching

15

3.1 Offline Processing

3.1.1 BASIS Dictionary Creation

3.1.2 BASIS Tree Creation

3.1.3 Evaluating the Generated Color TreeBASIS Dictionaries and Trees

3.2 Online Processing

3.2.1 FRI Binary Quantization

3.2.2 Descriptor Assignment

3.2.3 Descriptor Matching

3.3 Software Results

3.4 Future Work

34

## Chapter 4 Color TreeBASIS Hardware

36

4.1 Helios Platform and Xilinx Embedded Project

4.2 Image Preprocessing

4.3 Feature Detector

4.4 Binary Quantizing Module

4.5 BASIS Tree Processor

4.5.1 Fully Pipelining the BASIS Tree Processor

4.6 Library Matcher

4.7 Miscellaneous Hardware

4.8 Hardware Color TreeBASIS Resource Usage

4.9 Software Processing

4.10 Results

4.11 Future Work

69

## Chapter 5 Conclusions

77

5.1 Contributions

78
LIST OF TABLES

4.1 Pixel Movement Through Region Kernel ............................................ 53
4.2 FRI Movement Through the Tree Processor ........................................ 61
4.3 Total FPGA Utilization ................................................................. 67
4.4 Individual Hardware Module Resource Utilization ............................. 68
4.5 FPU Operation Latencies ............................................................... 68
## LIST OF FIGURES

1.1 SIFT Feature Detection and Matching ........................................ 3
2.1 FAST Feature Invariance .................................................. 6
2.2 7×7 Pixel FAST Kernel on Image ........................................ 7
2.3 FAST Kernel Buffer ..................................................... 9
2.4 Detail of FAST Kernel .................................................. 11
2.5 Conditions to Accept a Corner ........................................ 12
2.6 Block Diagram of FAST Hardware ...................................... 13
3.1 Processed BASIS Dictionaries .......................................... 19
3.2 Distribution of Y Channel Descriptors with Training Set .......... 25
3.3 Distribution of Y Channel Descriptors with Nature Video ........ 27
3.4 Distribution of Y Channel Descriptors with Car Driving Video .... 28
3.5 FRIs Before and After Being Binary Quantized .................... 29
3.6 Example Image from Idaho Test Set ................................... 34
4.1 Data Flow of the Color TreeBASIS Hardware Algorithm .......... 36
4.2 Helios Board and Daughter Board .................................. 39
4.3 Camera and Full System ................................................ 40
4.4 Color TreeBASIS Hardware Preprocessing Block Diagram ....... 41
4.5 Bayer Filter Pattern .................................................... 42
4.6 Before and After Demosaicing ........................................ 43
4.7 Before and After Color Correction ................................... 44
4.8 YCbCr Image Split into its Components .............................. 45
4.9 Before and After Gaussian Filtering .................................. 46
4.10 Display of Detected Features .......................................... 47
4.11 Pixel Region Kernel Sliding Over Frame ............................. 49
4.12 Region Kernel Accumulator ........................................... 50
4.13 Window Kernel Accumulator .......................................... 55
4.14 Block Diagram of the Binary Quantizing Module .................. 56
4.15 Color BASIS Tree Processor Loop ................................. 59
4.16 Block Diagram of the BASIS Tree Processor ...................... 62
4.17 Block Diagram of the Library Matcher .............................. 65
4.18 Results of Hardware Color TreeBASIS Using the ColorChecker .... 70
4.19 Conditional Scale Invariance with Right-Angle Features ........ 71
4.20 Results of Hardware Color TreeBASIS Using Book Covers ...... 73
4.21 MicroZed FPGA Board ................................................ 74
CHAPTER 1. INTRODUCTION

With the ever increasing presence of mobile electronics, the world grows more accustomed to the applications of computer vision. From facial recognition on one’s laptop [1]–[3], to the motion controlled video games in their living room [4]–[7], to the augmented reality applications on their smartphone [8], [9], to their car that automatically parallel parks [10], [11], computer and robotic vision are flooding into the lives of the masses. Although many of the applications serve as entertainment or convenience, other applications can be used to protect [12] and save lives [13], [14].

That being said, most applications of computer and robotic vision today are being designed in a software environment with the intention of running on a computer, a tablet, or a smartphone. While they are all perfectly good platforms, each with their own strengths, many potential applications for a vision system require the processing power of a modern desktop but also need to be portable and of low power consumption. Since a smartphone and a tablet might not always be sufficient in terms of processing power, the alternative is a custom hardware design.

Some applications of a custom vision hardware design might include an autonomous unmanned micro aerial vehicle, an autonomous ground-based security patrol vehicle, an autonomous micro ground-based search and rescue vehicle used to search through rubble, or perhaps a quality control monitor for a high-throughput factory that frequently changes the product on its lines. Though such custom hardware designs could be produced as application-specific integrated circuits (ASICs), the flexibility of a Field Programmable Gate Array (FPGA), essentially a programmable canvas for custom hardware circuits, allows the designer of the application to make modifications to the hardware and upload the changes to the hardware according to their present needs.

These applications also share a common set of vision based tasks: to see, identify, and react. One possible way for these applications to achieve their goal of “seeing” is to use a feature detector. The goal of the feature detector is to extract “interesting” or “unique” information from a frame
for future processing. These “seen” features can then be identified via a process called “matching” where the detected features are compared against a set of already processed features to see how similar they are. High similarities and low similarities are both good information from which the vision applications can react. For instance, a low match correspondence for the quality control monitor might mean that the product on the line is bad and needs to be discarded. For the patrol vehicle a low correspondence might indicate that everything is clear, but a high correspondence might indicate a threat and the need to sound an alarm.

Though the “react” portion of each application is different, they all share the same basic principles for defining the “seeing” and “identifying” portions of their respective algorithms.

For “seeing” the interest points, one of many feature detection algorithms exist with differing levels of accuracy and efficiency. These detectors also vary in their ability to detect the same features in different translations, rotations, scales, and view points (translational, rotational, scale and affine invariance) [15]. Although a feature detector that is robust, repeatable, and invariant on all fronts is desirable, most fully invariant detectors are too computationally intense to implement on a limited-resource FPGA-based system, requiring too many logic cells and too much memory [16]–[22].

“Identification” occurs via two main steps: description and matching. Some feature detectors also assign a description, essentially a unique identifier, upon detection [23]–[26], others find just the point of the feature with no description [27]–[30]. If the detector does not provide a descriptor, a description algorithm must be created [31]–[33]. Upon creation of the descriptor, that feature point now has a unique identity that can be used for matching according to one of a number of potential distance calculations (i.e. Euclidean, Manhattan, Hamming [34], etc.) [31]. Two features with identical or nearly identical descriptors are considered matches. Certain combinations of matches will then allow the various applications to “react” according to their programming.

In order to keep the amount of memory and logic needed for feature detection to a minimum, the translational and rotationally invariant Features by Accelerated Segment Test (FAST) feature detector [29], [30] has been chosen for this research. FAST quickly, and repeatably, finds features which are sharp angles (corners) in an image at a throughput of one feature per clock cycle upon filling the pipeline.
Figure 1.1: SIFT feature detection and matching. The top image shows detected feature points. The bottom image shows matches of features between two different frames. Images courtesy of Lukas Mach (en.wikipedia.org and http://labts.troja.mff.cuni.cz/ machl5bm/sift/).

Upon detection, these features are passed to the Color Tree BAis Sparse-coding Inspired Similarity (Color TreeBASIS) Feature Matcher, based off of TreeBASIS [31], for description and matching. Color TreeBASIS efficiently converts \(30 \times 30\) regions of color input pixels into 43-bit descriptors, a unique identifier for that region, and then compares the new descriptor with previously computed 43-bit descriptors to determine if any are a match. Matches are then sent to software for further processing (see Chapter 4.9).

This thesis will discuss in depth the steps necessary to create the Color FAST Feature Detector and the Color TreeBASIS Feature Matcher and then how to convert those two algorithms
into hardware. Chapter 2 discusses the FAST Feature Detector and its hardware implementation. Chapter 3 gives a look at the Color TreeBASIS algorithm, how it was derived from the TreeBASIS algorithm [31], and how it compares to its parent algorithm in software simulation. Chapter 4 explains in details the design of the Color TreeBASIS Feature Matcher in hardware and also presents the results of using the algorithm in a real FPGA system.
CHAPTER 2. FEATURES BY ACCELERATED SEGMENT TEST (FAST)

The feature detector that has been chosen for the vision system is Features by Accelerated Segment Test (FAST) [29], [30], an optimization of the Smallest Unvalue Segment Assimilating Nucleus (SUSAN) corner detector [15], [28]. FAST is translationally and rotationally invariant, meaning that a feature that is found in an image could be detected at any x-y coordinate position and at any rotation that is in the plane parallel to the camera. FAST is not, however, scale invariant nor affine invariant, meaning that changes in object size and variations of camera point-of-view or object rotations not parallel to the camera can cause that feature to not be detected [15]. Note that although FAST is rotationally invariant, it does not include any feature description, thus it doesn’t provide a dominant rotation in order to match two identical features at different rotations (see Chapter 2.2).

No special treatment of an input image is necessary for FAST to function. However, a noisy image will cause FAST to detect an excess of features that are weak and highly variable. It is a good idea to run filtering on the image before passing it on to the FAST feature detector. A noise reducing filter will remove features caused by electromagnetic variations in the sensor and a blurring filter will remove weak corner candidates such that only strong corners remain for matching purposes.

The FAST detector takes the pixels on the perimeter of a pseudo-circle at a given radius from a center pixel and compares said pixels to the center. If there are enough consecutive pixels at a high contrast to the center pixel, the center pixel is marked as a corner. High contrast is found under two conditions: when the perimeter pixel has a sufficiently larger value than the center pixel (brighter) and when the perimeter pixel has a sufficiently smaller value than the center pixel (darker). The contrast is “sufficient” when it is greater than or equal to the threshold chosen by the user.
Figure 2.1: Feature invariance with FAST. The top-left image shows a feature, circled in red. The top-right image shows the same feature highlighted after the object has been rotated along the plane of the camera and translated to another x-y coordinate position. The bottom-left image shows the object scaled down to the point that the feature is no longer detected. The bottom-right image shows a viewpoint change where the object is rotated on the horizontal plane that is perpendicular to the camera. This rotation causes the feature to go undetected.

Suppose there is a circle of radius $r = 3$, a minimum number of consecutively contrasting pixels $n = 9$, and a pixel contrast threshold $t = 20$. This means that of the 16 pixels at $r = 3$ from the center pixel, at least 9 consecutive pixels must be “brighter” by a value of at least 20 or at least 9 consecutive pixels must by “darker” by a value of at least 20 for the center to be marked as a corner.

In the example of Figure 2.2, the center pixel is the corner of a black square on a white background. The value of the white pixels is 255. The value of the black pixels is 0. The center pixel, highlighted in yellow, is a black pixel, as are the five perimeter pixels highlighted in green. The 11 perimeter pixels highlighted in red are white pixels, which have a high contrast to the center pixel (a value difference of 255). Therefore, this point would be marked as a corner because there are 11 consecutive high-contrast pixels on the circle around it.

Once the corner is detected, it goes through a scoring process that is similar to corner identification. In fact, the region of interest once again undergoes the process of comparing the
Figure 2.2: 7×7 pixel feature kernel on an image feature. The center pixel is highlighted in yellow. Red pixels signify high contrast to the center pixel. Green pixels signify low contrast.

perimeter pixels to the center pixels. However, the threshold used for comparing the center pixel with the perimeter pixels becomes variable during this process and the comparison process is repeated until it converges on a score value for the pixel. The threshold for this process starts out as an average, $b$, between a minimum value, $b_{\text{min}}$, and a maximum value, $b_{\text{max}}$, namely the original threshold and the pixel maximum value (255 for 8-bit pixels). This mean threshold, $b$, is then used to determine if the center point is a corner with the new threshold (either darker or brighter). If the pixel matches the corner criteria (i.e. 9 consecutive high-contrast pixels on the perimeter) the previous $b_{\text{max}}$ remains the same and $b_{\text{min}}$ is given the value of the mean, $b$. If the pixel is classified as not a corner, the previous $b_{\text{min}}$ remains the same and the $b_{\text{max}}$ is given the value of the mean, $b$.

This process repeats, much like a binary search, until the $b_{\text{min}}$ and $b_{\text{max}}$ converge, at which point the score is given by the value of $b_{\text{min}}$.

With a score given, non-maximal corner suppression is performed on the identified corners by looking at the local neighborhood around each corner and removing all but the corners with the highest score value. More precisely, the immediate 8 neighboring pixels around the point are compared with the corner's score and if any of these 8 pixels has a higher corner score, the center
pixel is marked as not a feature. This reduces the total number of features found and also leaves only the strongest features, allowing one to give the algorithm a lower threshold for finding corners and still get useful results.

2.1 Hardware FAST

FAST was originally developed with sequential processing speed in mind, containing an optimized decision tree with a quick exiting time upon the failure of the conditions necessary for a pixel to be considered a feature [29], [30]. This decision tree lets the software implementation of FAST start processing the next possible feature more quickly upon failure while slowing down slightly the extraction of a valid feature. However, in hardware, all pixels in the pseudo-circle can be compared to the center pixel in parallel, thus making the “accelerated segment test” portion of the feature detector unnecessary. In fact, the feature detector is pipelined and thus can process a new feature every clock cycle, making the throughput of the hardware FAST much higher than its software counterpart.

There are two main stages to the hardware implementation of the feature detector: corner identification and scoring and non-maximal corner suppression. There is also a start-up stage and a clean-up stage included to interface the pixel stream to the feature detector.

2.1.1 Start-up Stage

In the start-up stage, pixels are streamed into a memory that holds 6 rows of 640 pixels plus an extra 6 pixels; enough pixels to allow access to a 7×7 pixel region for processing. Until the buffer has been filled, no features can be extracted. Also, because the pixel in the middle is the one that gets processed, pixels are streamed out from the buffer when it is half full. This gives the output pixels the proper delay to line up with the corresponding features which have been identified.

2.1.2 Corner Identification and Scoring

Once the buffer has been filled, a set of 17 pixels (marked red, green, and yellow in Figure 2.2) is sent to the corner identifier every cycle until the last pixel has entered the buffer. These
pixels include the center point of the $7 \times 7$ region (the yellow pixel) plus the 16 pixels at a radius of 3 pixels from the center (the red and green pixels).

Figure 2.3: Graphical representation of the buffer used to store the pixels in a $7 \times 7$ region for processing. The rows are actually 640 pixels long. Notice that the buffer does not contain a 7th pixel on the 7th row. This is because the pixel that is being fed into the buffer is already available and does not need to be stored before being used.

To be identified as a corner, 9 consecutive perimeter pixels at a radius, $r = 3$, must be at a contrast larger than the threshold, $t$, to the center pixel. Furthermore, the pixel value for all 9 consecutive pixels must either be a threshold greater or a threshold less than the center pixel value (i.e. the contrasts must be in the same direction, either darker or lighter, for all 9 pixels). With these constraints there are 32 possible combinations of perimeter pixels that will identify the pixel as a corner.

First, all 16 pixels are compared to the center pixel twice; once to see if the pixel is larger than the center by the threshold and once to see if the pixel is smaller than the center by the threshold. Two comparison values are made, the first being the value of the center pixel plus the threshold, the second being the value of the center pixel minus the threshold. To avoid overflow and underflow, the comparison values are computed using signed integers with enough range to cover the values of $-t$ through $255 + t$ where $t = \text{threshold}$. With the comparison values computed, all 16 pixels are compared to both comparison values simultaneously, generating two 16-bit vectors. One vector is labeled “brighter,” with a value ‘1’ meaning that the pixel was a threshold greater than the center pixel value. The other vector is labeled “darker,” with a value ‘1’ meaning that the pixel was a threshold less than the center pixel value.
Once the two contrast vectors have been computed, the two vectors are split into 16 vectors of 9 bits and are then all tested to see if at least one of the vectors contains all ‘1’s. These split vectors and their comparisons are as follows:

vector(8,7,6,5,4,3,2,1,0) = “111111111”
vector(9,8,7,6,5,4,3,2,1) = “111111111”
vector(10,9,8,7,6,5,4,3,2) = “111111111”
vector(11,10,9,8,7,6,5,4,3) = “111111111”
vector(12,11,10,9,8,7,6,5,4) = “111111111”
vector(13,12,11,10,9,8,7,6,5) = “111111111”
vector(14,13,12,11,10,9,8,7,6) = “111111111”
vector(15,14,13,12,11,10,9,8,7) = “111111111”
vector(0,15,14,13,12,11,10,9,8) = “111111111”
vector(1,0,15,14,13,12,11,10,9) = “111111111”
vector(2,1,0,15,14,13,12,11,10) = “111111111”
vector(3,2,1,0,15,14,13,12,11) = “111111111”
vector(4,3,2,1,0,15,14,13,12) = “111111111”
vector(5,4,3,2,1,0,15,14,13) = “111111111”
vector(6,5,4,3,2,1,0,15,14) = “111111111”
vector(7,6,5,4,3,2,1,0,15) = “111111111”.

If either the “brighter” vector or the “darker” vector contains 9 consecutive bits equal to one, as tested above, the pixel is identified as a corner and is then passed to the scorer.

The scorer uses almost the exact same logic as the corner identifier. In fact, the only difference is that the threshold changes at each iteration of the scorer. As in the software algorithm, the goal of the scorer is to converge on a score value to pass to the non-maximal corner suppression hardware. However, an undetermined loop in a hardware pipeline is not feasible. Therefore, to limit resource usage and pipeline delay the number of iterations to determine a corner score is fixed at 7. The number of iterations can be easily changed due to parameterized HDL, but 7 iterations is accurate enough for the purpose of isolating the strongest corner points.
The first iteration of the scorer uses a fixed value for its threshold which is the average between the pixel maximum (255) and the corner detection threshold. The corner detection threshold used in hardware varies between channels, but it is approximately 12 for each channel. Therefore, the first scorer threshold is about 133. If the first iteration matches the criteria for a corner, the next iteration will be given a $b_{\text{min}}$ value of 133 and a $b_{\text{max}}$ of 255, creating a scorer threshold of 194. If the first iteration does not match the criteria for a corner, $b_{\text{min}}$ is assigned 12 and $b_{\text{max}}$ is assigned 133, creating a scorer threshold of 72. This process repeats 4 more times to come to a final score, which is assigned the value of $b_{\text{min}}$ of the last iteration.

The possible number of corner scores is dependent on the number of scoring stages. With 7 stages there are $2^7 = 128$ possible score values. If one wishes a higher score resolution to better filter out weak corners, extra stages are needed. The maximum number of stages is 8, since the range is $t$ to 255 and $2^8 = 256$. Adding more stages, of course, requires additional hardware resources.

Once computed, all scores, including the score of 0 for non-corners, are passed on to a 2-row-plus-2-pixel buffer to be used for corner filtering.
2.1.3 Non-maximal Corner Suppression

Upon filling the score buffer, a $3 \times 3$ window of corner scores is available to filter out weak features. In order to pass through the filter and be marked as a feature the center pixel of the $3 \times 3$ window must have a greater score than the pixels around it. More specifically, as seen in Figure 2.5, the center pixel must be greater than the 3 pixels above it and the pixel immediately to its left as well as have a score equal to or greater than the pixel to its right and below it. As seen in the equalities below, it is possible to get groups of points with the same score, in which case all are kept.

$$
\begin{array}{ccc}
C > (1,1) & C > (1,2) & C > (1,3) \\
C > (2,1) & (2,2) & C \geq (2,3) \\
C \geq (3,1) & C \geq (3,2) & C \geq (3,3)
\end{array}
$$

Figure 2.5: Conditions that must be met for a corner to not be suppressed. Note that $C$ is the value of the center pixel score.

The result of the corner suppression is that only the strongest of corners are output as features, decreasing the total number of features output from the detector and increasing the quality of the output features. In doing so, one can decrease the threshold for finding corners without making the results too noisy.

2.1.4 Clean-up Stage

Due to the filling of the buffers needed for processing the corners and the need to output the pixels alongside their feature identification, care must be taken to properly offset the streaming out of pixels. As mentioned before, pixels begin streaming out when the first buffer is half full.
This, however, means that the other half of the buffer must be streamed out at the end of processing a frame. A flushing process is in place that outputs the additional 640\times 3 + 3 pixels that are not yet streamed out upon the completion of the feature detection for the frame. Doing this flushing ensures that all 640\times 480 pixels that are input to the FAST detector are also streamed out from the detector.

Figure 2.6: Block diagram of FAST hardware.
2.2 Adding Dominant Rotation Orientation to FAST

To be able to match rotated features a rotationally invariant descriptor must be used. As of this moment, Color TreeBASIS is only translationally invariant. However, should a rotationally invariant version be designed, or another description algorithm be used, a dominant rotation orientation must be given to the description algorithm from FAST.

The method of assigning an orientation rotation, as proposed in this thesis, is to assign the middle point of the continuous segment of high-contrast pixels as the orientation. For example, for the feature shown in Figure 2.4, the middle of the continuous segment of high-contrast pixels (shown in red) is pixel 14. With the 16-pixel FAST, as used in this research, there are 16 possible orientations, representing 16 different rotations of 22.5°. Higher precision can be achieved by using a larger radius for FAST.

In certain circumstances the high-contrast segment is an even number, so the middle pixel must be chosen from either the first or the second of the two middle pixels or the sub-pixel point in-between the two (which allows for 32 different rotations instead of 16). Also, on occasion the segment will be all 16 pixels (a complete circle). For these two circumstances, a consistent behavior must be defined, although what that behavior is of less importance than is its consistency.

By supplying the dominant rotation orientation, the feature description and matching algorithm is given the ability to de-rotate the input features in order to make the matching of identical but rotated features possible [31].
CHAPTER 3. COLOR TREEBASIS FEATURE MATCHING

With features extracted from a frame, one may wish to determine if said features represent a known target or discover the homography between the current and a previous frame. Spencer Fowers proposed and designed the single-channel TreeBASIS feature matcher with the goal of finding the homography between consecutive aerial images [31]. With the homography computed, it is possible to relate its data to the real world, allowing for applications such as panoramic photography via photo-stitching [35], [36], object pose estimation [37]–[39], or even unmanned vehicle navigation [40]–[42].

The first aspect of TreeBASIS consists of a number of offline pre-processing steps to generate a library of elements used to describe features called Basis Dictionary Images (BDIs). These images are then used in a second offline step as comparison pivot points to generate a special binary tree called a BASIS tree. This BASIS tree is then used online as the method to generate feature descriptors. These descriptors are then compared with the descriptors of past images. If a pair of identical descriptors is found between the two images, the features are marked as matches and are sent to a list of matches from which a homography can be computed.

TreeBASIS showed great accuracy and consistency between consecutive frames of video from an aerial vehicle [31]. This set of test images allowed for the assumptions that the scaling would not drastically change between frames nor would the rotation since the aerial vehicle was flying in a level path at a fixed velocity. However, many potential platforms that could see benefit from using the TreeBASIS matching algorithm cannot hold the same assumptions about scaling, rotation, and viewpoint changes. For example, a ground based vehicle will encounter large changes of scaling on a per-frame basis, drastic viewpoint changes, as well as motion blur when cornering. Furthermore, frame to frame matching might not be necessary in other applications. One such platform is a fixed camera system used for object classification/detection. Such a system would
require a library of features corresponding to a known object and also be prepared to detect the object at a variety of \( x, y, \) and \( z \) rotations as well as at different scales.

The main goal of the Color TreeBASIS is the same as the goal of the original TreeBASIS, to implement an accurate and compact feature matching algorithm that can consistently find matches between a currently captured image and a stored list of features and do so in an efficient manner such that the algorithm can be utilized by an embedded system in real time. However, the platform for which Color TreeBASIS is being targeted is a stationary camera with the objective of detecting known objects.

In order to more accurately detect the target objects using TreeBASIS, all pixel information from the image (i.e. grayscale plus color information) are used. Adding color information to the TreeBASIS feature matcher increases the accuracy of the matching algorithm by making the set of unique descriptors larger, thus making false positives less likely. Also, by using all three visual channels from the camera, extra precision can be achieved while using smaller decision trees on a per channel basis.

This chapter will present the Color TreeBASIS algorithm. Section 3.1 describes in depth the offline preparations needed to make the Basis Dictionaries and Basis Binary Trees. Section 3.2 explains the online portion of the Color TreeBASIS algorithm.

### 3.1 Offline Processing

Before being able to run the feature matcher on input data, the Binary Dictionaries and the BASIS trees for each of the three data channels must be created. In order to prepare these elements, some groundwork must be established.

First, the pixel representation chosen for Color TreeBASIS is YCbCr, which is a special way of coding the RGB color space [43]–[45]. The luma (\( Y \)) channel contains a grayscale version of the original image while the chroma blue (\( C_b \)) and chroma red (\( C_r \)) channels contain the information for the color (\( C_b \) contains green and blue data and \( C_r \) contains green and red data). By using these three channels, emphasis can be given to \( Y \), since it contains the most relevant data for feature detection and matching, and the two chroma channels can be used to improve upon the precision of the feature descriptor.
Second, Feature Region Image (FRI) properties must be defined. An FRI is simply an image surrounding a feature point found from a feature detector. FRIs are not limited to a specific size, but $30 \times 30$ pixels is the size chosen for Color TreeBASIS following the previous research on TreeBASIS [31]. It was found that $30 \times 30$ pixel FRIs maximize the uniqueness of each FRI while keeping the amount of computation to a reasonable level since a larger FRI allows for increased descriptiveness of a feature point while a smaller FRI allows for a smaller memory footprint and lower computation costs.

Third, let BASIS Dictionary Images (BDIs) be defined as special FRIs which represent the fundamental patterns found in images. BDIs are the same size as FRIs ($30 \times 30$ pixels for Color TreeBASIS) and are used in the process of generating descriptors from FRIs.

### 3.1.1 BASIS Dictionary Creation

With the groundwork established, a dictionary of BDIs must be created for each channel. A dictionary size of 64 is chosen for each channel since dictionaries beyond size 64 only provide about a 1% improvement in matching accuracy while significantly increasing the memory required for the dictionary [31]. To generate the dictionaries one must first capture a large set of FRIs by running a feature detector over a number of training images. The training image set for the Color TreeBASIS dictionaries was taken from the embedded system for which the algorithm was designed and consists of images from a variety of buildings on the Brigham Young University campus. Only one frame every second or two was used for training so as to not bias the dictionary to any one specific pattern. Once a set of around 25000 or more FRIs is created, the set must be processed with the K-SVD [46] algorithm in order to generate the dictionary. The K-SVD algorithm takes the training images and performs singular value decomposition in order to create a dictionary for sparse representation; in other words, it determines a dictionary with the best representations of the fundamental shapes present in the training images. Once created, the dictionary contains the 64 patterns which best represent a basis for all of the FRIs used in training (i.e. the best set of patterns that could be used to reconstruct the original training images).

Following dictionary construction (Figure 3.1), the dictionaries undergo binary quantization (Algorithm 1) in order to reduce the memory footprint of the dictionaries as well as to simplify the logic needed to compare FRIs with the BDIs. Each $30 \times 30$ pixel BDI is passed to a function
which splits the BDI into 100 3×3 pixel regions. The average pixel value of the 30×30 region is computed and compared to the average pixel value of each 3×3 region. Each 3×3 region that has an average greater than the total average is assigned a value of 1. Each 3×3 region that has an average less than or equal to the total average is assigned a value of 0.

**Algorithm 1** Binary Quantization Algorithm. *F* is the FRI. Note that BDIs are just a special type of FRI. In the case of Color TreeBASIS, *F* is 30×30 pixels and there are a total of 100 3×3 pixel regions in *F*. QuantizedFRI is the result of the Binary Quantization algorithm.

1: procedure BINARYQUANTIZATION(*F*)
2:     FRIPixelSum ← 0
3: for all Pixel ∈ *F* do
4:     FRIPixelSum ← FRIPixelSum + Pixel
5: end for
6: FRIPixelAvg ← FRIPixelSum ÷ PixelsPerFRI
7: RegionIndex ← 0
8: for all Region ∈ *F* do
9:     RegionPixelSum ← 0
10: for all Pixel ∈ Region do
11:     RegionPixelSum ← RegionPixelSum + Pixel
12: end for
13: RegionPixelAvg ← RegionPixelSum ÷ PixelsPerRegion
14: if RegionPixelAvg > FRIPixelAvg then
15:     QuantizedFRI[RegionIndex] ← ‘1’
16: else
17:     QuantizedFRI[RegionIndex] ← ‘0’
18: end if
19: RegionIndex ← RegionIndex + 1
20: end for
21: end procedure

3.1.2 BASIS Tree Creation

With the dictionaries generated, the BASIS trees must be formed. In the original TreeBASIS algorithm, BASIS trees are formed by partitioning a training set of binary-quantized FRIs with the binary-quantized BDIs until fully partitioned (see Algorithm 2). Starting with the set of all FRIs, a BASIS tree node is created by comparing each FRI to every BDI using the Hamming similarity as the measure of similarity. The Hamming similarity, which is the opposite of the Hamming
Figure 3.1: The processed Basis Dictionaries. The original 30×30 BDIs are on the left, the binary-quantized 10×10 BDIs are on the right. The top row is the Y dictionary, the middle row is the Cb dictionary, and the bottom row is the Cr dictionary. The binary-quantized BDIs are just stored as 100-bit Boolean strings in memory.
distance \[y\], counts the number of bits that are similar between the two inputs, so a large value equals a high correspondence and a small value equals a low correspondence. For example, the Hamming similarity of the binary numbers 0b00000000 and 0b01001001 is 5 since there are 5 bits of value '0' in the second number and all bits are '0' in the first number. Each FRI with a similarity of less than or equal to 50 (half of the size of an FRI/BDI) is put into the left FRI set for the corresponding BDI and each FRI with a similarity greater than 50 is put into the right FRI set for the corresponding BDI. Having partitioned the FRI set with each BDI, the BDI which most evenly partitioned the FRIs is chosen as the most effectively descriptive BDI (EDBDI) for the BASIS tree node, the EDBDI is removed from dictionary set, and the node creation process starts again on the left set of FRIs and on the right set of FRIs. This process is continued until the FRI training set is completely partitioned.

The resultant BASIS tree contains approximately \(|F|\) nodes, where \(F\) is the input set of FRIs. It only contains approximately \(|F|\) nodes because there are situations in which a BDI is unable to partition a set of FRIs and as such, no more nodes are created for that subset of FRIs. The minimum achievable depth for the tree is \(D = \lceil \log_2 |N| \rceil\) and the maximum depth is \(D = |N|\), where \(|N|\) is the number of nodes in the tree and \(|N| \leq |F| - 1\). Although the minimum and maximum tree depths are possible, the depth tends to be closer to the minimum depth, which is an important factor for the success of the TreeBASIS algorithm. Descriptor length and the number of comparisons to arrive at that descriptor are equivalent to the depth of the tree, so a minimum depth is preferred for the sake of memory requirements and minimizing the amount of processing per descriptor.

In the original implementation of TreeBASIS, the optimum BASIS tree was generated using 50000 FRIs for training with a BASIS dictionary of 64 BDIs. The resulting tree was 17 deep, with each node containing a 6-bit pointer to its corresponding EDBDI, a 17-bit pointer to its left child, a 17-bit pointer to its right child, and a 1-bit flag indicating if the node is a leaf or not, a total of 41 bits. The memory needed for this tree was approximately 250 kB: \((50000 - 1) \times (6 + 17 + 17 + 1)\). This tree provided good accuracy in matching and a reasonable memory footprint, but there are a couple of inefficiencies in the tree building methodology.

First, the tree created is not fully packed. With a depth of 17, a fully balanced binary tree has \(2^{17} = 65,536\) leaf nodes, meaning that only about 76% of the possible descriptors represented
with 17 bits are being used. Second, most of the tree consists of memory pointers to other locations in the tree. Ideally the tree would be full, using 100% of the possible descriptors represented by the number of bits equal to the tree depth minus one, and would only hold the BDI address and leaf flag.

**Algorithm 2** Create BASIS Tree Node Algorithm. $B$ is the binary-quantized BASIS dictionary. $F$ is the binary-quantized training set of FRIs. $n$ is the index, or memory address, of the BASIS tree node.

1: **procedure** `CREATEBASISTREE_NODE`($B, F, n$)
2:    `NodeIndex` ← $n$
3:    `MinDifference` ← |$F$|
4:    $F_L$ ← $\emptyset$
5:    $F_R$ ← $\emptyset$
6:    **for all** $\beta$ ∈ $B$ **do**
7:        $L$ ← $\emptyset$
8:        $R$ ← $\emptyset$
9:        **for all** $f$ ∈ $F$ **do**
10:            if Hamming_Similarity($f, \beta$) ≤ $|\beta| ÷ 2$ **then**
11:                $L$ ← $L$ ∪ $f$
12:            **else**
13:                $R$ ← $R$ ∪ $f$
14:        **end if**
15:    **end for**
16:    `Difference` ← ||$L$| − |$R$||
17:    **if** `Difference` < `MinDifference` **then**
18:        `MinDifference` ← `Difference`
19:        `EDBDI` ← $\beta$
20:        $F_L$ ← $L$
21:        $F_R$ ← $R$
22:    **end if**
23: **end for**
24: **if** |$F_L$| < 2 **or** |$F_R$| < 2 **then**
25:    `Leaf` ← 1
26: **else**
27:    `Leaf` ← 0
28:    `LeftChild` ← `CREATEBASISTREE_NODE`( $B \setminus BDI, F_L, 2n + 1$)
29:    `RightChild` ← `CREATEBASISTREE_NODE`( $B \setminus BDI, F_R, 2n + 2$)
30: **end if**
31: **end procedure**
In Color TreeBASIS, the BASIS trees are created with the goal of removing these two inefficiencies (see Algorithm 3). First, trees are formed by constraining the depth to a fixed value and using a training set that exceeds the size of the tree at that fixed depth. For example, there are 32,768 leaf nodes in a full tree with a depth of 16. If a tree of this depth is desired, a training set much larger (an order of magnitude or more) than 32,768 FRIs should be used in order to fill the tree. Not all of the FRIs used for training will have a unique descriptor, but most of the possible descriptors at size 16-bits will be used. Second, since BASIS trees are binary search trees, their memory access patterns are uniform and need not require memory pointers.

Recall the following two lines of the pseudo-code for creating a basis tree:

\[
\text{LeftChild} \leftarrow \text{CreateBASISTreeNode}(B, BDI, FL, 2^n + 1)
\]
\[
\text{RightChild} \leftarrow \text{CreateBASISTreeNode}(B, BDI, FR, 2^n + 1)
\]

The key to the memory accesses are the expressions \(2^n + 1\) and \(2^n + 2\). From any given node, one must multiply that node’s address by two and then add one to reach its left child or add two to reach its right child. Considering this and the fact that a traversal of the tree always starts at the same node (the root) with an address of 0, no pointers need to be stored in the BASIS tree.

Combining these two improvements results in BASIS trees with nearly 100% efficiency in their descriptor representations (i.e. nearly all descriptors that could be represented by a given number of bits are represented) and a memory size fixed to the tree depth. In one test, a training set of 300,000 FRIs and a desired tree depth of 16 created a BASIS tree with a descriptor representation efficiency of 99.94% (32749 out of a possible 32,768 leaf nodes). The memory required by the tree is computed as \((\log_2 |B| + 1) \times (2^\text{depth} - 1)\) where \(|B|\) is the size of the BASIS dictionary. In the case that \(|B| = 64\) and the depth = 17, the memory required for the BASIS tree is \((6 + 1) \times (217 - 1) \approx 115\) kB; a 54% decrease in memory used compared to the original BASIS tree as well as a 31% increase in the number of unique descriptors being represented by the tree, assuming all possible descriptors are represented.

For Color TreeBASIS, three BASIS trees must be created; one for each YCbCr channel. As was previously stated, the emphasis is on the grayscale information found in the Y channel. This means that the Y channel BASIS tree will be larger than the BASIS trees for the other two
Algorithm 3 Updated Create BASIS Tree Node Algorithm. Note that it differs only in the inclusion of depth \((d)\) for the leaf calculation. Also, although the node index is still shown, the tree node is merely stored at that location in memory and no pointers exist to the node indices.

```plaintext
1: procedure CREATEBASISTREENODE\((B, F, n, d)\)
2:   NodeIndex \(\leftarrow n\)
3:   MinDifference \(\leftarrow |F|\)
4:   \(F_L \leftarrow \emptyset\)
5:   \(F_R \leftarrow \emptyset\)
6:   for all \(\beta \in B\) do
7:     \(L \leftarrow \emptyset\)
8:     \(R \leftarrow \emptyset\)
9:     for all \(f \in F\) do
10:        if \(\text{Hamming\_Similarity}(f, \beta) \leq |\beta| \div 2\) then
11:           \(L \leftarrow L \cup f\)
12:         else
13:           \(R \leftarrow R \cup f\)
14:       end if
15:     end for
16:     Difference \(\leftarrow ||L| - |R||\)
17:     if Difference < MinDifference then
18:         MinDifference \(\leftarrow\) Difference
19:         EDBDI \(\leftarrow \beta\)
20:         \(F_L \leftarrow L\)
21:         \(F_R \leftarrow R\)
22:     end if
23: end for
24: if \(|F_L| < 2\) or \(|F_R| < 2\) or \(d = \text{MaxDepth}\) then
25:   Leaf \(\leftarrow 1\)
26: else
27:   Leaf \(\leftarrow 0\)
28:   LeftChild \(\leftarrow\) CREATEBASISTREENODE\((B \setminus \text{BDI}, F_L, 2n + 1)\)
29:   RightChild \(\leftarrow\) CREATEBASISTREENODE\((B \setminus \text{BDI}, F_R, 2n + 2)\)
30: end if
31: end procedure
```

channels, giving it a greater variety of descriptors than the other channels. Also, because there are three channels of descriptors in Color TreeBASIS instead of just one, the size of the individual channel descriptors do not need to be as large as the size of the descriptor in the original TreeBASIS algorithm. This means that the overall descriptiveness of a Color TreeBASIS descriptor is much larger while using the same or less memory for the BASIS trees. To demonstrate this, consider the 17-bit BASIS tree which has about 65000 possible descriptors and uses about 115 kB of memory.
Now, consider a collection of three BASIS trees, the first being 16-bit, containing about 32000 possible descriptors and using around 57 kB of memory, and the other two being 15-bit, containing about 16000 descriptors each and each using about 29 kB of memory. In the end, the Color TreeBASIS trees use the same amount of memory as the single channel TreeBASIS, around 115 kB, but together create descriptors of 43 bits (15-bit + 14-bit + 14-bit), bringing the total possible number of unique descriptors to around $2^{43} \approx 8.8$ trillion.

### 3.1.3 Evaluating the Generated Color TreeBASIS Dictionaries and Trees

Good Color TreeBASIS dictionaries should cause the tree creation process to output a tree that is shallow and full. Dictionaries were generated from the training images taken from the Helios embedded system [47], [48]. The dictionary generated for the Y channel was used to partition a set of 200,000 unique FRIs and created a BASIS tree for Y containing 32749 out of a possible 32768 leaf nodes, or 99.94% of all possible descriptors represented by a tree of depth 16. The dictionary generated for the Cb channel was used to partition a set of 150,000 unique FRIs and created a BASIS tree for Cb containing 16384 out of a possible 16384 leaf nodes, or 100% of all possible descriptors represented by a tree of depth 15. The dictionary generated for the Cr channel was used to partition a set of 105,000 unique FRIs and created a BASIS tree for Y containing 16384 out of a possible 16384 leaf nodes, or 100% of all possible descriptors represented by a tree of depth 15. These 3 BASIS trees together allow for a total of $32749 \times 16384 \times 16384 \approx 8.79$ trillion unique descriptors.

In order to test the validity of the BASIS trees, a number of test scenes were converted into sets of unique FRIs and the FRIs were pushed through the tree to see how well distributed the descriptors were. This test indicates whether or not a BASIS tree is good by showing that all or most possible descriptors are found in the set of test images and that each descriptor is found about the same number of times as the other descriptors. If there is a heavy bias towards only a few descriptors, the tree will not be able to provide descriptors unique enough for matching purposes.

The first test set is the FRI set used to build the dictionaries. Since the number of FRIs used to build the tree far exceeds the number of leaf nodes in the tree, it is expected that at least a few descriptors will receive a greater number of hits than the others. Figure 3.2a shows the number of hits for each descriptor in the Y channel tree, numbered 0 through 32767. There are approximately
200,000 FRIs in the test set, so a perfectly balanced distribution would be about 6 hits on each descriptor. The training set shows a good distribution, with 89% of all descriptors being hit within a standard deviation of the average hit frequency of 6. Only a very few number of descriptors are hit significantly more than the ideal average.

Figure 3.2: Distribution of Y channel descriptors hit by the 200,000 FRIs used to form the BASIS tree. A) shows the hits at each descriptor from 0 to 32767. B) shows the quantity of descriptors with total hits from 0 to 45. C) shows the same data as B, just zoomed in to show the number of descriptors with total hits 13 through 45.

The second test set is from a video of some foliage and flowers. This footage was not taken with the camera on the Helios embedded system. Also, the images are of subjects in nature
which were not included in the FRI set used to build the dictionary. Figure 3.3a shows the findings
from the test. There are approximately 460,000 unique FRIs in the test set, so a perfectly balanced
distribution would be about 14 hits on each descriptor. The training set shows a good distribution,
with 91% of all descriptors being hit within a standard deviation of the average hit frequency of
14. The anomalies in this set are more notable, with one descriptor getting hit about 800 times and
another getting hit around 1000. Even so, these two anomalies only account for 0.4% of all of the
FRIs processed. This test indicates that the BASIS tree is good.

The third test set is from a video of a car driving through a city. The distribution of descriptor
hits also indicates a good dictionary, with 94% of all descriptors being hit within a standard
deviation of the average hit frequency of 14.

The results of the distribution tests for the Cb and Cr channels demonstrated similar re-
sults to those of the Y channel, indicating that the dictionaries and BASIS trees were well formed,
especially for the images taken in the environment where the embedded system will be capturing
images and running the TreeBASIS algorithm. The tests from other cameras in differing envi-
ronments showed a little more deviation, along with more anomalous descriptors with high hit
counts, but still showed good results, demonstrating the robustness of these BASIS trees and the
TreeBASIS descriptors in general.

3.2 Online Processing

The online portion of the Color TreeBASIS algorithm is accomplished in three main steps. The first is the binary quantization of the input features. The second is the formation of descriptors
by feeding the binary-quantized FRIs into a BASIS tree comparator which compares the FRIs to
EDBDIs from the root through a path to a leaf node. The third is the descriptor matcher which
compares the newly computed descriptors with previously processed descriptors from either the
previous frame or from a library of pre-defined objects.

3.2.1 FRI Binary Quantization

Upon the completion of feature detection, the detected features, along with the surrounding
pixels comprising the FRI, are forwarded to the binary quantization algorithm. This algorithm is
Figure 3.3: Distribution of Y channel descriptors hit by the 460,000 FRIs from some footage of nature. A) shows the hits at each descriptor from 0 to 32767. B) shows the quantity of descriptors with total hits from 0 to 74. C) shows the quantity of descriptors with total hits from 75 to 964.

the same as shown in Algorithm 1 which was used for the binary quantization of the BDIs and FRIs necessary for the formation of the BASIS trees. This algorithm splits the FRI into a number of square regions, computes the average pixel value of each region, then compares each region average with the average of the whole FRI. Each region that has an average greater than the FRI average is assigned a binary value of ‘1’. Each region that has an average less than or equal to the FRI average is assigned a binary value of ‘0’.

27
Figure 3.4: Distribution of Y channel descriptors hit by the 470,000 FRIs from some footage of a car driving through a city. A) shows the hits at each descriptor from 0 to 32767. B) shows the quantity of descriptors with total hits from 0 to 74. C) shows the quantity of descriptors with total hits from 75 to 964.

Upon completion of the binary quantization algorithm, the three channel color FRI has been converted into three binary strings of length equal to the number of regions in the FRI. In the implementation of Color TreeBASIS chosen for testing, the FRIs are $30 \times 30$ pixels and are split into 100 $3 \times 3$ pixel regions. Since the images are in color, the FRIs are quantized in each individual channel and the output is three 100-bit binary strings representing each channel. This
binary quantization reduces the data that will be used in the rest of the matching algorithm from $900 \times 24 = 21600$ bits to just $3 \times 100 = 300$ bits. Although one might be concerned that the data for the FRI is reduced by 98.6%, it is important to note that the number of representable FRIs with 300 bits is still $2^{300} \approx 2.03 \times 10^{90}$, which is more than enough to accurately differentiate between all of the FRIs in an image.

Figure 3.5: Examples of some FRIs being binary quantized.

3.2.2 Descriptor Assignment

With the FRIs simplified to binary strings, each binary quantized FRI is sent to the BASIS descriptor assignment algorithm. Since there are three distinct channels in each FRI and because there is a different BASIS tree and BASIS dictionary for each channel, the descriptor assignment process for a Color TreeBASIS feature requires separate processing on each channel. Even so, the process of generating a descriptor is the same for each channel and is done in parallel.
To generate a descriptor for a single channel, a single channel binary quantized FRI is input to a tree processor which then compares the FRI with the EDBDIIs of the various nodes stored in the BASIS tree for that channel. The first comparison is between the FRI and the EDBDI of the root node. If the FRI and EDBDI are similar (i.e. a Hamming similarity of greater than \( |FRI| \div 2 \)), the value of the least significant bit of the descriptor is assigned a ‘1’ and the next node used for comparison is the right child of the root. If the FRI and EDBDI are not similar, the value of the least significant bit of the descriptor is assigned a ‘0’ and the next node used for comparison is the left child of the root. The same process is repeated until the current tree node being used for comparison is a leaf node, at which point no comparison takes place and the descriptor is considered generated.

For experimental purposes, the maximum depth of the Y channel tree is 16 and the maximum depth of the Cb channel and Cr channel trees are 15, thus creating a 15-bit descriptor and two 14-bit descriptors. The descriptor generation is complete when the FRI has been processed by all three tree processors and a 43-bit, three channel descriptor has been output. Each of the three channels’ descriptors are generated in parallel, so the algorithm is limited only by how long it takes to compute the longest descriptor which is the Y channel descriptor. The tree processor algorithm pseudo-code is displayed below (Algorithm 4).

### 3.2.3 Descriptor Matching

The final part of the Color TreeBASIS algorithm is the matching of the newly generated descriptors with the previously processed features. In the testing of the original TreeBASIS algorithm, the set of previously computed descriptors is the set of descriptors captured from the previously processed frame in a video stream. The test images were taken from a plane flying at a steady altitude and velocity in a straight line with stable lighting conditions. Due to the constancy of these images, TreeBASIS could use a non-rotationally invariant and non-scale invariant feature detector (FAST) as well as have a simple equals-to matching condition, meaning that two descriptors are considered a match only if they have identical descriptors.

The environment in which Color TreeBASIS is to be used is indoors with objects that can change scale, rotation, and position rapidly and whose lighting can change drastically between frames. These differences make the task of matching more difficult due to the fact that the feature
Algorithm 4 Generate Descriptor Algorithm. FRI is the binary quantized FRI for which the descriptor is formed. Descriptor is the output of the function.

1: procedure GENERATE_DESCRIPTOR(FRI)
2:    Descriptor ← 0
3:    CurrentDepth ← 0
4:    TreeAddr ← 0
5:    TreeNode ← BASIStree[0]
6:    BDI ← BASISdictionary[TreeNode.BDIaddr]
7:    while TreeNode.IsLeaf = FALSE do
8:        if Hamming_Similarity(FRI, BDI) ≤ |FRI| ÷ 2 then
9:            Descriptor[CurrentDepth] ← ‘0’
10:           TreeAddr ← 2 × TreeAddr + 1
11:        else
12:            Descriptor[CurrentDepth] ← ‘1’
13:           TreeAddr ← 2 × TreeAddr + 2
14:        end if
15:        CurrentDepth ← CurrentDepth + 1
16:        TreeNode ← BASIStree[TreeAddr]
17:        BDI ← BASISdictionary[TreeNode.BDIaddr]
18:    end while
19:    DescriptorValid ← TRUE
20: end procedure

detector and matcher can only tolerate a little bit of rotation and scaling before matching becomes impossible. Also, lighting differences due to shadows and reflections are much more prevalent in the laboratory than in aerial images. These factors have led to a rule set of conditional matching for Color TreeBASIS.

For two descriptors to be considered a match they must pass one of a number of conditions which demonstrate acceptable similarity. For instance, if the Y channel (grayscale) of the two descriptors is identical, the two color channels (Cb and Cr) must match above a threshold. This condition can be even more fine-tuned by only considering the two descriptors a match if either Cb or Cr is a strong match and the other is at least a partial match. A particular set of rules has been chosen via experimentation and are:

Descriptor A and Descriptor B match if:

1. (a) the first 14 bits of the Y channel descriptors are identical

and
(b) i. the first 12 bits of the Cb channel descriptors are identical and the first 5 bits of the Cr channel descriptors are identical

or

ii. the first 12 bits of the Cr channel descriptors are identical and the first 5 bits of the Cb channel descriptors are identical

or

iii. the first 8 bits of the Cr channel descriptors are identical and the first 8 bits of the Cb channel descriptors are identical

OR

2. (a) the first 11 bits of the Y channel descriptors are identical

and

(b) i. the first 12 bits of the Cb channel descriptors are identical and the first 8 bits of the Cr channel descriptors are identical

or

ii. the first 12 bits of the Cr channel descriptors are identical and the first 8 bits of the Cb channel descriptors are identical

OR

3. (a) the first 10 bits of the Y channel descriptors are identical

or

(b) the first 12 bits of the Cb channel descriptors are identical and the first 12 bits of the Cr channel descriptor are identical

If any of these three sets of conditions are met, the feature points are marked as matches. These rules make Color TreeBASIS more robust for matching under less than ideal conditions. With extra lenience built in, features are matched with slightly larger rotations and scalings,
matches are still found with variations in lighting, and libraries of objects for matching can be made without fearing that the conditions will change too much to ever find the object outside of the location where the training images were taken. Results using the new method of feature matching will be reviewed at the end of the chapter about the Color TreeBASIS hardware. Before then, a quick test to directly compare Color TreeBASIS to the single channel TreeBASIS is performed.

3.3 Software Results

In order to compare the Color TreeBASIS algorithm to the single channel TreeBASIS algorithm, the Color TreeBASIS feature matcher was run on 60 images of the Idaho test set [31], per the test constraints on TreeBASIS [31]. Under the testing conditions, a set of matches was considered good if the homography generated from it had diagonal element $H_{1,1}$ and $H_{2,2}$ close to a value of one. Specifically, the two diagonal elements had to be between 0.7 and 1.3. The descriptor matching technique used for Color TreeBASIS in this test was a simpler form of the tests in Section 3.2.3. The compared descriptors required a match of the first 14 bits of the Y channel and then a combined total of 10 matching bits for the Cb and Cr channels to be considered a match. These constraints led to a valid homography on 56 of the 60 test images, thus demonstrating an accuracy of 93.3% as compared to an accuracy of 79.6% found for TreeBASIS [31]. Since the BASIS tree of the original TreeBASIS algorithm is different from the Y channel tree created for Color TreeBASIS, grayscale accuracy using just the Y channel was also tested, demonstrating an accuracy of $49/60 = 81.7\%$, very close to the 79.6% accuracy found in the original experiment.

This test demonstrates the benefit of adding color information to the TreeBASIS algorithm. It is apparent that the color information contains a bit more noise than the grayscale, causing poor results if one tries to match the features by identical descriptors on all channels. This is perhaps due to limitations of the YCbCr color representation and the fact that green is split over two channels, or it might be that color hue and saturation might vary more greatly between images than does the grayscale component of the images. Nevertheless, by using the grayscale as the main source of data and then using the color information to support the findings of the grayscale feature matching, increased accuracy is achieved.
3.4 Future Work

As mentioned in Section 2.2, a potential improvement to the FAST algorithm is the addition of rotationally invariant feature descriptors. Assuming that FAST (or whichever feature detector being used) supplies a dominant rotation orientation, the feature description algorithm should be able to rotate each input FRI such that the dominant orientation is fixed for all FRIs at 0 (or whichever arbitrary position that is chosen). The rotation of the descriptors is possible via two methods.

The first method is to do the rotation on the input FRI before quantization. When rotating, some pixels will be “drawn” off screen while some will be left blank. Therefore cropping is necessary. With an FRI of $30 \times 30$ pixels, the valid region after rotating becomes $22 \times 22$ pixels. This means that the FRI must either be passed in as a larger area in order to crop down to $30 \times 30$, the FRI must be upscaled from the cropped image back to the full size after rotation, or the new cropped size must be accepted for quantization.

The second method rotates the $10 \times 10$ descriptor post-quantization. This encounters the same cropping issues as the previous method. However, since the descriptor only contains 100 bits of data opposed to the 21,600 bits in the $30 \times 30$ FRI, rotation and scaling can cause more loss or corruption of the information. Furthermore, if the corruption is too great, two identical features of
different rotations can be normalized into quantized FRIs that generate very different descriptors; not ideal for feature matching.

Though the first method offers better consistency of quantized FRIs, due to the fact that the quantization is determined on the FRI post-rotation, it also requires more computational effort. The second method is preferred in terms of computational simplicity for implementing in hardware since the rotation is computed on only 100 bits and because the streaming optimizations of the binary quantizing module can remain intact (see Chapter 4). In fact, the optimizations made to fit the binary quantizing module into the hardware system become obsolete if the FRIs are rotated before they are quantized since the region computations will be different for each rotation and cannot be saved for future use (see Chapter 4). It is possible that rotating post-quantization is accurate enough for Color TreeBASIS, in which case a hardware implementation of rotation can be added to rotate the binary-quantized FRI before computing its descriptor.
CHAPTER 4. COLOR TREEBASIS HARDWARE

For systems that require real-time feature matching, specialized hardware is the best option. A custom hardware platform more fully exploits parallelisms from within an algorithm as well as allowing the designer to more fully control the flow of data making on-the-fly processing more efficient. The Color TreeBASIS algorithm has been placed into such a custom hardware system, built off of the foundation established by the TreeBASIS hardware design [31].

Figure 4.1: Data flow of the Color TreeBASIS hardware algorithm.
The Color TreeBASIS hardware module works in parallel with the data streamed from a camera, processing the pixels while the image is still being captured. As the pixels are fed in, they undergo preprocessing, some being specific to the camera being used in the embedded system (i.e. Bayer demosaicing [49], [50] and color correction [51], [52]) and others specific to the algorithm (i.e. RGB to YCbCr conversion [43], [44], [53] and Gaussian filtering [10]). As the pixels are streamed out of the preprocessing units, they are passed to the FAST Feature Detector. The input pixels as well as detected features are then fed to the Binary Quantizing Module. All 30×30 pixel regions are binary quantized, but only the FRIs (the 30×30 pixel regions corresponding to a feature) have their quantized values forwarded to the BASIS Tree Processor. As binary quantized FRIs are passed to the BASIS Tree Processor, each takes turns accessing the BASIS trees and dictionaries in order to generate a descriptor. These descriptors are then passed to the Library Matcher which attempts to match the processed descriptors with the pre-computed descriptors of a library set. All matches are then forwarded to memory for further processing.

This chapter will explain in depth the complete hardware design for Color TreeBASIS as well as give a brief overview of the embedded system environment used in the Color TreeBASIS experimentation. Section 4.1 overviews the Helios board [47], [48], its peripherals, and the Xilinx Embedded Project which encapsulates the Color TreeBASIS algorithm. Section 4.2 overviews the system specific image preprocessing as well as the general image preprocessing needed for Color TreeBASIS. Section 4.3 briefly reviews the FAST Feature Detector hardware and establishes how it is used for three pixel channels. Section 4.4 describes the Binary Quantizing Module’s design. Section 4.5 describes the BASIS Tree Processor. Section 4.6 describes the Library Matcher. Section 4.7 overviews the additional hardware modules designed. Section 4.8 overviews the hardware resources used by Color TreeBASIS. Section 4.9 overviews the additional software processing necessary for finding homographies. Section 4.10 discusses the algorithmic results of the Color TreeBASIS hardware. Last, Chapter 4.11 discusses a few topics that could be pursued for the improvement of the Color TreeBASIS algorithm.

4.1 Helios Platform and Xilinx Embedded Project

The Helios Robotic Vision Platform [48] is an FPGA-based embedded system platform designed at Brigham Young University with the intended purpose of real time processing of vision
applications [4], [54]–[58]. The Helios board contains a Xilinx Virtex 4 FX60 FPGA plus a number of embedded peripherals, including a 32 MB SDRAM, a 1 MB SRAM, a 16 MB Flash memory, a Xilinx Platform Flash chip, and a USB 2.0 chip. The Helios board also provides a number of headers and ports for the connection of user designated peripherals, a JTAG port for FPGA programming, and a number of buttons, switches and LEDs [47]. The Xilinx Virtex 4 FX60 FPGA [59] contains 56,880 logic cells (4-input look up tables), 522 KB of block RAM, 128 DSPs, and 2 embedded PowerPC 405 processors. The FX60 can run at a clock frequency up to 500 MHz and the two embedded PowerPC processors can run at a clock frequency up to 450 MHz. However, certain design limitations reduce the clock frequency of the FPGA and PowerPC processors to 100 MHz in the Color TreeBASIS project. These limitations stem from timing constraints due to the routing of such a large project in the FPGA and the inclusion of a floating point coprocessor in the FPGA for the PowerPCs to use for software computations. For the application of the Color TreeBASIS algorithm a daughter board is used with the purpose of attaching a Micron MT9V024 camera [60] and an XBee wireless universal asynchronous receiver/transmitter (UART) [61]. The daughter board being used was originally designed for the Robot Racers Senior Project at BYU, thus it supports even more peripherals that are not being used for testing the Color TreeBASIS algorithm, namely: a gyroscope, a color LED tower, a second wireless UART, two digital pulse wave modulators (PWMs), a wheel encoder, a speaker, and a wired UART; most of which are controlled by an embedded processor.

A Xilinx Platform Studio (XPS) project has been created to define the interfacing of the Helios peripherals necessary for testing Color TreeBASIS with the Virtex 4 FPGA. The specific peripherals being used in this project include: Micron MT48V8M32LFF5-8 32 MB SDRAM, Cypress CY7C68014A-56LFXC USB 2.0 chip, Micron MT9V024 camera, and a Digi XBee Pro Wireless UART. The XPS project connects these peripherals to the hardware components in the FPGA, such as the embedded PowerPCs and the Color TreeBASIS hardware, via a processor local bus (PLB) in order to create a unified embedded environment to be controlled via a software project.

A Xilinx Software Development Kit (SDK) project has been developed to drive the XPS project peripherals, control the inter-peripheral communications, and to control memory allocation in the project. Onboard processing of algorithms is possible in this software environment, although
Figure 4.2: The Helios board and the daughter board.
that capability is not currently being exploited, as will be explained in section 6 of this chapter. Software controls the video capture (i.e., the control signals for when to start capturing and where to store the frames in memory) and also controls the output of frames and matches to PC via USB.

4.2 Image Preprocessing

Before the Color TreeBASIS algorithm can be performed on the incoming image data, the incoming pixels must undergo preprocessing. Two types of preprocessing are present in the embedded system: system-specific preprocessing and general preprocessing.

Every camera is unique and thus different cameras may require different image processing steps in order to achieve usable pixel data. Whereas some camera systems might accomplish this...
within their own hardware (i.e. a proprietary image processor embedded on the camera board), the MT9V024 does not do all of its image processing onboard and thus requires external processing.

The sensor array of the MT9V024 is designed to capture images using a Bayer filter [49]. Therefore, the camera really captures the images in grayscale, with each pixel representing either a red, a green, or a blue pixel, and the 8-bit pixel data must be converted into RGB via demosaicing [50]. To accomplish this, the pixels are streamed into a buffer 4 line plus 4 pixel buffer which outputs a $5 \times 5$ pixel window for processing. Demosaicing takes places by multiplying the $5 \times 5$ window by three kernels, one for each of the channels, which change depending on the position in the Bayer pattern, and then assigning the sums of the products to the pixel in the center of the window. The final product is a 24-bit RGB image.

The color of the images output from the demosaicing algorithm, however, is not very accurate (most likely due to the lighting conditions in the laboratory). The images retrieved from

Figure 4.4: Block diagram of the image preprocessing.
the camera appear unnaturally yellow and green and thus require color correction. As the pixels
arrive in the color correction module, they are multiplied by a unique $3 \times 3$ color correction matrix
that has been computed for the camera [62]. The result is a new RGB value that has better color
fidelity. With color correction in place the system-specific preprocessing is complete and general
preprocessing may proceed.

The first step of preprocessing specific to the Color TreeBASIS algorithm is a conversion
of the pixels from RGB to YCbCr. The process is similar to color correction in that the RGB pixels
are multiplied by a conversion matrix to form YCbCr pixels. This matrix operation is

$$
\begin{bmatrix}
Y \\
Cb \\
Cr
\end{bmatrix} = \begin{bmatrix}
0 \\
128 \\
128
\end{bmatrix} + \begin{bmatrix}
0.299 & 0.587 & 0.114 \\
-0.169 & -0.331 & -0.500 \\
0.500 & -0.419 & -0.081
\end{bmatrix} \cdot \begin{bmatrix}
R \\
G \\
B
\end{bmatrix}.
$$

This conversion accepts full-range R/G/B inputs (0-255) and outputs full range Y/Cb/Cr values
(0-255) [53]. Rounding is applied to the outputs in cases where the result is not an integer.

With the RGB pixels converted to YCbCr, the final preprocessing step of Gaussian filtering
is performed. The purpose of this module is to remove noise from the image that is present due
to electromagnetic interference and artifacts from demosaicing. First a two line plus two pixel
buffer is filled in order to compute on a $3 \times 3$ pixel window. Once the window has been filled, each
Figure 4.6: The raw image before demosaicing followed by the same image after demosaicing and color correction.

of the nine pixels is multiplied by a corresponding coefficient as suggested in the Gaussian filter kernel [63]. The chosen kernel is,

$$
\begin{bmatrix}
21 & 31 & 21 \\
31 & 48 & 31 \\
21 & 31 & 21 \\
\end{bmatrix}
$$

(4.2)
Figure 4.7: Captured image before and after color correction. The third image was taken with a digital camera for comparison and the fourth shows the expected color of the ColorChecker [52].

Note that these 9 kernel multiplications occur on each of the three channels, thus meaning that there are really 27 multiplications, not just 9. The multiplications are performed in DSP blocks and the outputs are summed in pairs, leaving 5 partial sums. Next the 5 partial sums of each channel are summed and the resultant values are normalized by selecting only the top 8 bits and throwing away the bottom 8 bits (in effect a divide by 256). The normalized values are passed on as the new values of YCbCr of the pixel in the center of the window.

The Gaussian filter causes the input image to be blurred, thus reducing noise and removing weak features that would have otherwise been detected in the image. It also adds a 1 pixel black border around the image where the kernel could not be used because some pixels were off of the image. After filtering the pixels are ready to be processed for feature detection.
4.3 Feature Detector

The algorithm for FAST feature detection is described in Chapter 2 and will not be reiterated here. However, there are some necessary notes that must be added to understand how the algorithm is being used in Color TreeBASIS.

The first note is that the FAST feature detector only works on one channel at a time, meaning that for Color TreeBASIS, three instances of the feature detector must be run on each pixel. This is done in parallel and as such, if any of the three channels is classified as a feature, the whole 3-channel pixel is classified as a pixel.

The second note is that although there are three instances of FAST feature detector processing each pixel, the only difference in the computation of the three is the threshold used for each
Figure 4.9: Detail of two images, the first without filtering and the second with a Gaussian filter. Notice that the second image is less grainy and the borders become less defined.
channel. That means that the number of non-maximal suppression stages is equivalent between the 3 feature detectors as well as image size and data width. Holding everything but the thresholds as equivalent simplifies the processing and ensures that the results of feature detection arrive at the same time for all channels.

![Image displaying detected features. Black means that there was no feature found and the color value of the feature displays its feature score (brighter color means a stronger feature).](image)

**Figure 4.10**: Image displaying detected features. Black means that there was no feature found and the color value of the feature displays its feature score (brighter color means a stronger feature).

### 4.4 Binary Quantizing Module

Of all of the hardware modules in the Color TreeBASIS design, the binary quantizing module is the most resource consuming due to the fact that it uses a 29 line plus 29 pixel buffer to perform arithmetic on a 30×30 pixel window. In fact, the original binary quantizing module, as designed for the single channel TreeBASIS algorithm [31], was so resource consuming that it could not be placed into a Virtex 4 FX60 FPGA. Much optimization was necessary in order to
reduce the resource usage of the module enough to get it to fit into the FPGA, especially since three-channel quantization was desired.

The initial implementation of the binary quantizing module in TreeBASIS has a number of inefficiencies that do not matter for simulation but make implementation difficult. The first is the method in which the region sums and the window sum are computed. In the brute force approach chosen for the quantizing module, each of the 100 $3 \times 3$ regions have their 9 pixels summed to generate the region sums and then all 100 region sums are added together to form the window sum. This amounts to 900 additions every clock cycle, which not only uses a lot of resources but also significantly increases the clock period due to carry chain delays, effectively making this design unusable. To make matters worse, the window sum and the region sums also have their values right shifted to form the window and pixel averages and then are used for the quantizing comparisons in the same clock cycle as the additions. Furthermore, the right shifts used in place of division are not exact because a right shift can only be used for divisions of a power of 2, thus the window average is actually being divided by 1024, not 900, and the region averages are only being divided by 8 instead of 9. Although the result of quantization is still a 100-bit binary string for each channel, the results are not the same as the algorithm pseudo-code would suggest.

For the algorithm to work efficiently in hardware and be equivalent to the pseudo-code a number of changes must be implemented. First, logic optimizations must be introduced to reduce the number of additions necessary to compute the window and region sums. Second, the computation of the region and window averages must be corrected so that the comparisons between the two generates binary quantized values for each $3 \times 3$ region that are equivalent to the results generated by the pseudo-code algorithm. Third, pipelining must be introduced such that the clock frequency can be maximized. Making these improvements generates a much more efficient module that is able to binary quantize 3 channels in parallel and still leave room on the Virtex 4 FX60 for other hardware components.

To optimize the logic of the window and region sums one must take advantage of the streaming nature of the system. It is important to remember that as the summation kernel “slides” over the frame, it is really just moving over one pixel column each time, thus the summed data remains the same inside of the kernel except for the pixel values entering the kernel from the right and the pixel values leaving the kernel from the left. This means that if an accumulation value is
stored for each kernel, only the entering pixels and the exiting pixels must be added and subtracted from the kernel.

Figure 4.11: 3×3 region kernel passing over an image region. The white line marks the position of the kernel. The red squares are pixels that are exiting the kernel, the green squares are pixels entering the kernel, the blue squares are pixels that are still in kernel, and the gray squares are the pixels that have not yet entered the kernel.

In the naive approach for a 3×3 kernel, there are 4 additions and 3 subtractions, a slight improvement over 9 additions, and for a 30×30 kernel there are 31 additions and 30 subtractions, a significant improvement over 900 additions. The optimization can be taken one step further at the cost of slightly higher memory usage by storing the sum of the input pixels and then using the stored sum when the pixels exit the kernel instead of computing the sum again. This reduces the number of operations for a 3×3 kernel from 7 to 5 (4 additions and 1 subtraction) and reduces the number of operations for a 30×30 kernel from 61 to 32 (31 additions and 1 subtraction). The extra registers created to store the input sum to later be used as an output (3 registers for a 3×3 kernel and 30 registers for a 30×30 kernel) is still a net reduction in terms of total percentage of FPGA fabric used.

To reduce the number of LUTs used, DSPs are used as accumulators for the region sums. To even further the optimization, the three channels of a single region are accumulated in one DSP. First the 3 input pixels to the kernel are summed on each channel and the result is registered (necessary pipelining in order to improve the clock frequency). Next the input sums are concatenated into a 36-bit vector and are passed to the DSP as the additive input. At the same time the exiting pixel
sums are inverted and are also input as an additive input. The inversion plus the constant carry in of ‘1’ ensure that the exiting pixel sums are actually subtracted from the accumulated value. The DSP takes these values and adds them to the current accumulated value and then stores the new accumulated value in an internal register. On the next clock cycle the region sum is available on the output.

Doing the region sum of three channels in one DSP is possible because the maximum value of a 3×3 pixel region is $255 \times 9 = 2295$, which is a 12-bit number but is much smaller

Figure 4.12: The $3 \times 3$ pixel region kernel accumulator. The three adders have registered outputs.
than 4095, the maximum value of a 12-bit unsigned number. Even if the accumulated sum added the new input before subtracting the exiting pixels, the value only reaches 3060, never reaching a 13-bit value. Also, the subtraction of the unsigned exiting pixel sum via inversion plus the carry in of 1 works because of a number of conditions. First, the value being subtracted from the region is never greater than the current sum of the region. Second, the maximum value to be subtracted is 765, a 10-bit value, and the inversion always causes the 11th and 12th bits become one, which, coupled with the fact that the current sum must contain at least the value that is being subtracted, always has the effect of adding 1 to the 12-bit value packed above it. This means that the first 12-bit subtrahend is formed via an inversion plus the carry in, forming a true 2’s-complement negative number. The second 12-bit subtrahend is also inverted and receives an equivalent carry in which comes from the overflow of the addition and subtraction of the first 12-bits of all operands. The same happens for the third 12-bit subtrahend. The following example demonstrates the functionality of the region accumulator DSP:

\[
\text{Accum} = (\text{Accum} - \text{RegionOut}) + \text{RegionIn},
\]

\[
Y : \text{Accum} = 320(0x140), \text{RegionIn} = 700(0x2BC), \text{RegionOut} = 180(0x0B4),
\]

\[
Cb : \text{Accum} = 2295(0x8F), \text{RegionIn} = 765(0x2FD), \text{RegionOut} = 765(0x2FD),
\]

\[
Cr : \text{Accum} = 1034(0x40A), \text{RegionIn} = 333(0x14D), \text{RegionOut} = 0(0x000),
\]

\[
\text{ExpectedAccumY} = (329 - 180) + 700 = 840(0x348),
\]

\[
\text{ExpectedAccumCb} = (2295 - 765) + 765 = 2295(0x8F7),
\]

\[
\text{ExpectedAccumCr} = (1034 - 0) + 333 = 1367(0x557),
\]

\[
\text{AccumDSP} = 0x140 \text{ 8F7 40A},
\]

\[
\text{RegionInDSP} = 0x2BC \text{ 2FD 14D},
\]

\[
\text{RegionOutDSP} = \sim (0x0B4 \text{ 2FD 000}) = 0xF4B \text{ D02 FFF},
\]

\[
\text{AccumOut} = 0x140 \text{ 8F7 40A} + 0x2BC \text{ 2FD 14D} + 0xF4B \text{ D02 FFF} + 0x1
\]

\[
= 0x348 \text{ 8F7 557} \text{ (plus a carry out which is discarded)}. \]

The window sum is also computed using DSPs, but since there is only one window kernel, and since the result is an 18-bit vector, three DSPs are used (one for each of the three channels). However, this brings up an important question: how many region kernels are needed? With a quick
inspection it might seem that there need to be 100 region kernels, one for each region. Fortunately
this is not the case. Once again the streaming nature of the system is exploited such that the region
sum only needs to be determined for the far right column of regions. As the window kernel slides
from left to right over the image, pixel data enters the 10 region kernels in the far right column
of the window. The 10 region sums are computed and then are forwarded to the next column of
regions. Table 4.1 demonstrates how the region sums are registered and passed to the regions to
the left. Notice that the columns of region 1 are identical to the columns of region 0 from three
time-steps before. That is, if the sum of region 0 is 128 at time \( T_1 \), the sum of region 1 at time
\( T = T_1 + 3 = T_4 \) is also 128. Furthermore, the sum of region 2 is identical to the sum of region
0 from six time-steps before. This indicates that once a region sum has been computed, it merely
needs to be time delayed by three clock cycles before being used as the region sum for the next
region to the left, and then three clock cycles again to go to the next neighbor, etc. This can be
written as a pseudo-code operation of,

\[
\text{Region}(x) \leftarrow \text{delay}(\text{RegionSum}0, 3x),
\]

where delay is a function that registers the first input for the number of clock cycles indicated by
the second input and \( x \) is the region column number, from 0 to 9. As can be seen, region 1 receives
the sum of region 0 delayed by \( 3 \times 1 = 3 \) clock cycles, region 2 receives the sum of region 0 delayed
by \( 3 \times 2 = 6 \) clock cycles, and region 9 receives the sum of region 0 delayed by \( 3 \times 9 = 27 \) clock
cycles.

As previously mentioned, the mean value computations for both the regions and the win-
dow in the original binary quantizing module are just approximations. To guarantee that the results
of the module are identical to the pseudo-code algorithm a different approach is needed. Since
division is not an efficient operation, an alternate set of operators is desired. The key to finding an
alternate operation is to realize what the mean pixel value of the regions and the full window.

By summing all of the pixels and dividing by the number of pixels in the kernel, the mean,
or average, is found. It also can be interpreted as the value of the sum per pixel of the kernel,
meaning that it represents an equal distribution of the kernel sum over all pixels in the kernel.
Through this definition it becomes clear that the kernel sum can also be defined as the total value
Table 4.1: Movement of pixels in region kernel as the window kernel moves from left to right. Each column represents a column from the corresponding region kernel. As such the values can be from 0 to 765. The grayed cells are just to highlight the movement of the values through the different region kernels.

<table>
<thead>
<tr>
<th></th>
<th>Region 2</th>
<th>Region 1</th>
<th>Region 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>T1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 128</td>
</tr>
<tr>
<td>T2</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 128 350</td>
</tr>
<tr>
<td>T3</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>128 350 212</td>
</tr>
<tr>
<td>T4</td>
<td>0 0 0</td>
<td>0 0 128</td>
<td>350 212 702</td>
</tr>
<tr>
<td>T5</td>
<td>0 0 0</td>
<td>0 128 350</td>
<td>212 702 0</td>
</tr>
<tr>
<td>T6</td>
<td>0 0 128</td>
<td>350 212</td>
<td>702 0 5</td>
</tr>
<tr>
<td>T7</td>
<td>0 128 350</td>
<td>212 702</td>
<td>0 5 0</td>
</tr>
<tr>
<td>T8</td>
<td>128 350 212</td>
<td>702 0 5</td>
<td>0 433 117</td>
</tr>
<tr>
<td>T9</td>
<td>350 212 702</td>
<td>0 5 0</td>
<td>433 117 369</td>
</tr>
</tbody>
</table>

per \( x \) pixels, where \( x \) is the number of pixels in the kernel. As such, the weight of the region kernels is 9, or in other words the sum is the pixel value over 9 pixels. The window kernel sum is defined similarly as the pixel value over 900 pixels. From this it can be concluded that all that is needed to do a comparison of the region sums with the window sum is to put the two values into equal weights, also called normalization. One such normalization is the method present in the pseudo-code which puts the two values at weight 1 by finding the mean. Since two divisions are not desired, another weight for normalization must be found.

The most obvious comparison weight is 900, the weight of the window sum kernel. All that must be done is a multiplication of the region sums by 100. This can be accomplished with three bit-shifts and two additions,

\[
\text{RegionSumWeight}_{900} \leftarrow (\text{RegionSum} \ll 6) + (\text{RegionSum} \ll 5) + (\text{RegionSum} \ll 2),
\]

which is equivalent to,

\[
\text{RegionSumWeight}_{900} \leftarrow (\text{RegionSum} \times 64) + (\text{RegionSum} \times 32) + (\text{RegionSum} \times 4).
\]
This solution is valid and allows accurate comparison of the region sums to the window sum. However, this shift and add function shows that there is room for improvement. Note that the bottom two bits of RegionSumWeight\(_{900}\) are always zero due to the fact that the smallest value is the region shifted left by two. With this fact the weight of the sums can be set to \(9 \times (100 \div 4) = 225\). The equations for the optimized region sums and the window sum are,

\[
\text{RegionSumWeight}_{900} \leftarrow (\text{RegionSum} \ll 4) + (\text{RegionSum} \ll 3) + \text{RegionSum},
\]

and

\[
\text{WindowSumWeight}_{225} \leftarrow \text{WindowSum} \gg 2.
\]

For Color TreeBASIS, the window and region sums weight is the value over 225 pixels and the weighted region sums can either be computed using the shift and add method or using a DSP in combinational node to multiply the sum by 25. The latter is the method presently used in the binary quantizing module due to the fact that it only uses 3 DSPs.

The final optimization necessary for the binary quantizing module to be complete is the addition of pipelining. The hardware algorithm as a whole will be summarized now, step by step, to demonstrate the pipelining and also to pull together all of the elements previously discussed into a complete system.

Stage one of the pipeline requires the pixels entering the 30\(\times\)30 pixel window kernel to be summed in groups of three to be used by the far right column of region sum kernels as well as to be used in the window sum calculation.

In stage two of the pipeline, three things happen. First, the input region sums (S0 through S9 in Figure 4.13) are passed to the input registers of the 10 region sum accumulators along with the ten output region sums. Second, the region sums are passed to the delay registers from which the output region sums are taken (this is the delay buffer that stores the input values until they are needed as output values). Third, the ten region sums are summed into the window kernel input partial sums (S10, S11, and S12 of Figure 4.13).

The third stage sums the 3 partial sums of each channel to form the complete window kernel sum inputs and then passes the input sums to the input registers of the window sum accumulators.
The output sums are also retrieved from the delay buffers and passed to the input registers of the window sum accumulators.

The fourth stage is just a bubble, or no-op, stage to allow the outputs of the accumulators to be registered for use in the next stage. It is also important to note that the region sum accumulators have an extra layer of input registers such that the region sums are output at the same time as the window sum.

The fifth, and final, stage takes care of a number of tasks. First it multiplies the region sums by 25 and right shifts the window sum by 2 (dividing the value by 4). Second it passes the weighted region sums to the delay registers to be used by the neighboring regions in the future. Third it retrieves the 90 delayed weighted region sums and compares them and the 10 new region sums to the weighted window sum. All regions greater than the window sum receive a binary value of ‘1’ and all less than or equal to the window sum receive a binary value of ‘0’. Although the multiplication is in stage five as a combinational operator, it could have also been placed in stage four with a pipeline register to hold its result for use in stage five.

Figure 4.13: The stages of generating the window kernel sum for a single channel.

Every center pixel for which the 30×30 window passes receives a 300-bit binary quantized string (100 bits for each channel), aside from the borders where the window is partially off of the image which receive an invalid quantized string. One new quantized string is output every clock cycle once the input pixel buffers are filled. Although almost every pixel window is given a binary
quantized value, only the pixels which were detected as features (FRIs) have their binary quantized strings passed to the BASIS tree processor. This is handled by another delay buffer which receives an input signal indicating whether the pixel is a feature or not and then delays the output until the binary quantized string attributed to that pixel leaves the binary quantizing module.

Figure 4.14: Block diagram of the binary quantizing module.
4.5 BASIS Tree Processor

The BASIS Tree Processor is the first hardware module in the Color TreeBASIS hardware system that is not fully pipelined. The main reason for the tree processor not being fully pipelined is due to resource sharing to reduce the resource usage of the module. Furthermore, as will later be demonstrated, the module is able to construct more than enough descriptors at a capture rate of 60 FPS, thus it does not need full pipelining. There are, however, a few improvements made to the BASIS tree processor, as designed for the TreeBASIS algorithm [31], to improve its throughput and the achievable clock frequency.

Before going over the improvements the rest of the tree processor module must be reviewed. The BASIS tree processor receives binary quantized features from a FIFO storing features. As the FRI is input to the module, it is compared to the BDIs in the BASIS tree from the root to a leaf node, building one bit of the TreeBASIS descriptor at each level of node in the BASIS tree. This process is driven by a state machine with 4 principal states: Idle, LoadingTreeNode, LoadingBDI, and Running. In the Idle state the module is waiting for a new FRI to be input. When one arrives in the module, the tree processor begins a loop of LoadingTreeNode, LoadingBDI, and Running that only ends when a leaf node is reached. In the first iteration the tree node that is loaded is the root (address = 0) and the corresponding BDI is loaded from the BASIS dictionary. Once the two cycles of loading are complete, the Hamming similarity is computed and used to determine what the next tree node address will be and if the descriptor bit will be ‘1’ or ‘0’ (see Chapter 3.2.2). This 3 cycle loop continues until the leaf is reached, which is a maximum 16 times if the tree processor is computing the Y descriptors, meaning that it takes \(1 + 16 \times 3 = 49\) cycles to compute one descriptor and return to idle. This, however, can be improved with pipelining and exploiting parallelism.

The original BASIS tree processor, just like the binary quantizing module, lacks the pipelining necessary to allow for a fast clock frequency. This is true because of the Hamming similarity calculation which first exclusive-NORs the input 100-bit binary quantized FRI with a 100-bit BDI and then counts all of the ‘1’s in the resultant string. This computation in the original module performs 1 exclusive-NOR and 100 additions, plus a comparison, in just one clock cycle. This had to be split up into multiple clock cycles in order to achieve a clock frequency of 100 MHz in the Virtex 4 FX60. The Hamming similarity calculation is currently performed in three pipeline stages.
The first divides the 100-bit binary vector into 20 groups of 5 binary digits and sums their values. The second stage divides the 20 sums of the previous stage into 4 groups of 5 sums and adds their values. The third stage adds the 4 sums to generate the complete Hamming similarity value and then compares this value with \(50 \left(\frac{|FRI|}{2}\right)\) in order to assign the next BASIS tree address and to assign the descriptor bit value. Although this pipelining seems to reduce the throughput of the module from 1 descriptor every 49 cycles to 1 descriptor every \(1 + 16 \times 5 = 8\) cycles, there is a way parallelize the processing such that the number of cycles per descriptor formed is reduced to about 18.

The key to exploiting the potential parallelism of the BASIS tree processor is to understand what limits the module from being completely pipelined. The limiting factor of the module is the sharing of the BASIS tree ROM and the BASIS dictionary ROM resources. The problem is that these ROMs are placed into memories that only have one read port, meaning that only one value can be read from each of the ROMs during a clock cycle. In a fully pipelined implementation, the ROMs would have enough ports (or would be replicated) such that the memory information could be accessed, when needed, by all FRIs at all depths of the tree (for the Y channel tree processor there would need to be 16 ports since the filled pipeline will have an FRI accessing the ROMs once for each of the 16 tree levels). Notice that with the pipelined loop (the two ROM loading cycles plus the three cycles for Hamming similarity/descriptor assignment) the design only accesses each ROM once every 5 clock cycles. Memory is only accessed at most 20% of the time, meaning that the introduction of staggered parallel computation paths will increase the maximum potential ROM usage to 100%.

The maximum number of features that can be processed at a given time in the tree processor using staggered parallel computation paths is 5, since each ROM can only be accessed once in a clock cycle and the pipeline loop is 5 cycles in length. To achieve this, an extra wait state is introduced (between Idle and the first LoadingBASISTree) that holds the incoming features until one of the five paths becomes available. Availability is defined as the existence of an empty compute path and no other paths accessing the BASIS tree ROM on the next clock cycle. In the case of maximum utilization, five FRIs are present in the feature FIFO and are passed to the BASIS tree processor one after the other. As they enter, they fill the 5 compute paths, but do not interfere with each other’s ROM accesses. Assuming the worst case on all paths for the Y channel, on the
Figure 4.15: Inner workings of the BASIS tree processor loop.
last cycle of the 16th loop the first descriptor is output, followed by the other 4 descriptors on the next 4 cycles. This means that at full utilization, 5 descriptors are computed in $2 + 16 \times 5 + 5 = 87$ cycles; one descriptor per $87/518$ cycles.

Since it takes approximately 18 cycles to process one descriptor, the maximum number of features processed by the BASIS tree processor at a given camera frame rate can be calculated. The Helios [11] system’s camera [x] can capture at either 30 or 60 FPS, thus two calculations shall be performed. The equation for solving the number of features, $n$, processed per frame at a frame rate $f$ and a clock period $T_c$ is

$$ n = \frac{f - 1}{18 \times T_c}. \quad (4.3) $$

Since the clock period is 10 ns, the result for 30 FPS is 185,185 features processed (60% of the total frame) and the result for 60 FPS is 92,592 features processed (30% of the total frame). The number of features that can be processed per frame is much larger than the actual amounts that the system will ever process (at most around 8,000 features for Color TreeBASIS). If the maximum allowed features to be processed of 2000 for the original TreeBASIS [31] is used, the maximum FPS is computed as,

$$ f = \frac{1}{n \times 18 \times T_c}, \quad (4.4) $$

$$ f = \frac{1}{2000 \times 18 \times 10^{-8}}, \quad (4.5) $$

$$ f \approx 2777 \text{FPS}. \quad (4.6) $$

For all intents and purposes, the BASIS tree processor performs at a high enough efficiency that even processing around the Color TreeBASIS maximum of 8,000 features per frame, it can still work with a camera capturing at 694 FPS. In its current state it can supply 92,592 3-channel descriptors (43-bits each) to the library matcher every frame. However, if for some reason it is desired to form descriptors for all pixels, full pipelining is required.
Table 4.2: FRI movement through the staggered parallel computation paths of the BASIS tree processor pipeline. I is the Idle state, N is the Next state which waits for the next available computation path, L1 is the LoadingTreeNode state, L2 is the LoadingBDI state, R1 is the first state of computing the Hamming similarity, R2 is the second state of computing the Hamming similarity, and R3 is the third state of computing the similarity and where the descriptor bit is assigned.
Figure 4.16: Block diagram of the BASIS Tree processor. The Tree Processor Core is diagrammed in Figure 4.15. Note the five Tree Processor Cores that all access the same ROMs.

4.5.1 Fully Pipelining the BASIS Tree Processor

The task of fully pipelining the BASIS tree processor requires distributing the BASIS tree ROMs and BASIS dictionaries. For the BASIS dictionaries this means that there will need to be as many copies of the dictionary as there are levels in the BASIS tree for that channel. For example, the Y channel requires 16 copies of the Y channel BASIS dictionary since it is 16 levels deep. The BASIS trees, however, are only accessed on a per level basis, meaning that each of the 16 accesses
to the Y channel BASIS tree per clock cycle is at a different level of the tree, from level 0 through 15.

Knowing this fact, the BASIS tree ROM can be split into 16 different ROMs of differing lengths. The levels closer to the root can be stored in registers since their sizes range from 1 7-bit entry at level 0 to 1024 7-bit entries at level 10. The other levels can be placed into separate block RAMs [64]. Although it might seem desirable to get rid of the BASIS dictionary ROMs altogether and just store the BDIs in the BASIS tree ROMs, the amount of memory used for the BASIS trees and dictionaries would increase significantly. For the Y channel the memory usage would increase to $101 \times 65535 = 6,619,035$ bits from $7 \times 65535 + 16 \times 64 \times 100 = 561,145$ bits. Also note that minimum extra memory for the fully pipelined version is $15 \times 64 \times 100 = 96,000$ bits. This, however, is somewhat misleading since each of the copies of the dictionaries will be placed in their own 18,432 bit block ram (about a 35% utilization of each block RAM) and many levels of the BASIS trees will be placed into underutilized block RAMs as well.

Once the ROMs have been distributed and copied, it is just a matter of fully unrolling the 5-cycle loop of the tree processor state machine into a linear pipeline.

### 4.6 Library Matcher

The library matcher is another module that is not fully pipelined in order to minimize resource usage. The resource usage of this module is also directly related to the number of objects that will be detected by the Color TreeBASIS algorithm as well as the number of descriptors being stored for each object. In order to be more portable, the library matcher is parameterized such that new library entries can easily be added and such that the size of the library entries can be modified.

The flow of data through the library matcher is fairly straightforward. First, 3-channel descriptors of 43-bits plus their corresponding (X,Y) coordinates are input to the matcher and stored in a FIFO. When the matcher is able to process a new descriptor, it grabs one from the FIFO and passes it to each library comparing module (there are four in the test hardware). In each library comparing module the descriptor is compared with the descriptors of the library entry ROM, using the comparison tests as outlined in Chapter 3.2.3, until either a match is found or there are no more descriptors in that library entry ROM. Arbitration logic makes sure that all matches that are found for an input descriptor are given the opportunity to be output from the library matcher. Matches,
m, are formed by concatenating the coordinates of the input descriptor, \(coord_{in}\), the coordinates of the library descriptor, \(coord_{lib}\), and the index of the library, \(ind\), in the following fashion:

\[
m = coord_{in} \parallel coord_{lib} \parallel ind.
\]

These matches are passed to another FIFO and then are eventually passed on to software for the computing of a homography.

There is an extra feature that can be used in the matcher which serves the purpose of improving the correlation of the matches. What this feature does is move the pointer of the first element available for matching in a library when a match is found. Since the matches in a library are stored in streaming order, this ensures that matches can only be found in the order they are present in the library. This also prevents library features from being matched multiple times. During experimentation this feature seemed to improve the accuracy of the computed homographies, perhaps by reducing the number of false matches.

The performance of this module in terms of how many descriptors can be processed in one frame is completely dependent on the maximum number of descriptors in a library entry. For testing purposes the max library size is 128 and there are 4 library entries. In a worst-case scenario, none of the descriptors of a frame would match the descriptors of the library entries. Therefore, the number of cycles to process one descriptor in the worst case is 1 cycle to retrieve the descriptor from the FIFO, 128 cycles to compare the descriptor to all of the descriptors in the library entry ROM, 4 cycles (one for each library entry) to output the match values, and 1 cycle to prepare to retrieve the next descriptor; a total of 134 cycles. Using the equation

\[
n = \frac{f^{-1}}{134 \times T_c}
\]

at 60 FPS and a clock period of 10 ns shows that the library matcher can process 12,437 descriptors in one frame, which is 55% more than the estimated maximum of features that will be processed per frame with Color TreeBASIS. A maximum FPS while processing 2000 descriptors is computed to be 373 FPS.

Although completely pipelining the library matcher is possible, it requires that the descriptors be stored in registers so that every single descriptor in the library entry can be accessed in the
same cycle. Upon doing this, all of the comparisons can be done in parallel in one cycle, but then some form of arbitration must be created in order to forward all of the matches out of the library matcher. In the end the library matcher will either use too many resources to be a feasible hardware module in a resource limited system or the number and size of the library entries will have to be reduced to the point that the module loses its effectiveness.

Figure 4.17: Block diagram of the library matcher. Note the four matcher and ROM blocks representing the matching of features to four different library entries.

4.7 Miscellaneous Hardware

There are a number of hardware modules that were mentioned in the previous sections that are too basic to warrant in depth explanation and analysis. These modules include:

- Accumulator
- Buffer for 3×3, 5×5, 7×7, and 30×30 kernel
• Combinational multiplier

• FIFO

• ROM

The accumulator and the combinational multiplier are merely wrappers around DSP primitive blocks [65], [66], with the first placing the DSP in the accumulate mode of \( P = P + (A : B + C) \) and the second setting the DSP as a combinational multiplier of \( P = A \times B \).

The various sized kernel buffers instantiate the minimum number of block RAM primitives [64] to store the kernel and takes care of the shifting of the pixels through the kernel. The buffers are parameterizable for data width and image width.

The FIFO and the ROM use the standard methods of inference for instantiating block RAMs for Xilinx synthesis [64]. The FIFO is first-word-fall-through and is parameterizable in terms of data width and FIFO depth. The ROM is actually inferred through the use of a RAM that writes its read data back to the read location due to a bug in the Xilinx synthesis flow that doesn’t allow block RAMs to be used for inferred ROMs. The ROM module is also parameterizable for data width and memory depth.

4.8 Hardware Color TreeBASIS Resource Usage

The complete hardware vision system, including the bus logic and image preprocessing maps to 23037 slice registers, 31002 4 input LUTs, 4468 LUTRAMs, 187 block RAMs, and 97 DSPs. The total Helios system, including an Auxiliary Processor Unit Floating-Point Unit (APU FPU), maps to 29483 slice registers, 38799 4 input LUTs, 4816 LUTRAMs, 227 block RAMs, and 101 DSPs. The total system device utilization can be seen in Table 4.3 and the individual module utilization can be seen in Table 4.4.

The resource utilization of the FAST feature detector can be significantly reduced by decreasing the number of stages of non-maximal suppression. In this synthesis the number of stages is 8, but can easily be reduced to 6 or 7 stages if less accurate suppression is acceptable.
Table 4.3: Total device utilization.

<table>
<thead>
<tr>
<th>Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Flip Flops</td>
<td>29,483</td>
<td>50,560</td>
<td>58%</td>
</tr>
<tr>
<td>DCM autocalibration logic</td>
<td>21</td>
<td>29,483</td>
<td>1%</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>37,519</td>
<td>50,560</td>
<td>74%</td>
</tr>
<tr>
<td>DCM autocalibration logic</td>
<td>12</td>
<td>37,519</td>
<td>1%</td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>24,188</td>
<td>25,280</td>
<td>95%</td>
</tr>
<tr>
<td>Number of Slices containing only related logic</td>
<td>24,188</td>
<td>24,188</td>
<td>100%</td>
</tr>
<tr>
<td>Number of Slices containing unrelated logic</td>
<td>0</td>
<td>24,188</td>
<td>0%</td>
</tr>
<tr>
<td>Total Number of 4 input LUTs</td>
<td>39,191</td>
<td>50,560</td>
<td>77%</td>
</tr>
<tr>
<td>Number used as logic</td>
<td>32,703</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as a route-thru</td>
<td>1,672</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used for Dual Port RAMs</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as Shift registers</td>
<td>4,734</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>255</td>
<td>352</td>
<td>72%</td>
</tr>
<tr>
<td>IOB Flip Flops</td>
<td>384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOB Dual-Data Rate Flops</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of BUFG/BUFGCTRLs</td>
<td>5</td>
<td>32</td>
<td>15%</td>
</tr>
<tr>
<td>Number used as BUFGs</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of FIFO16/RAMB16s</td>
<td>227</td>
<td>232</td>
<td>97%</td>
</tr>
<tr>
<td>Number used as RAMB16s</td>
<td>227</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of DSP48s</td>
<td>101</td>
<td>128</td>
<td>78%</td>
</tr>
<tr>
<td>Number of DCM_ADVs</td>
<td>3</td>
<td>12</td>
<td>25%</td>
</tr>
<tr>
<td>Number of PPC405_ADVs</td>
<td>2</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Number of JTAGPPCs</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Number of RPM macros</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Fanout of Non-Clock Nets</td>
<td>3.24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.9 Software Processing

The original idea for generating homographies using RANSAC [67] of the library image to the current frame was to do everything onboard the Virtex 4 FX60 using the embedded PowerPCs and the APU FPU that was added to the hardware design. Though the homography generation works onboard the FPGA, the embedded software system takes so long to operate on the floating point computations that the system could not process the homographies in real time. Just the Gaussian elimination used for finding a homography takes about 150,000 clock cycles to compute just the floating point operations. Add in all of the integer operations and the clock cycles increase to around 300,000, just for the Gaussian elimination. There are other floating point and integer
operations present in the RANSAC algorithm, plus the fact that RANSAC repeats the homography solver a number of times to find the “best” homography, running upwards of thousands of times in some cases. If the RANSAC algorithm causes the homography to be computed 100 times, the computation delay will be some value greater than 30,000,000 cycles, or 0.3s with a 100 MHz clock. Since the frame capture is 60 FPS, the amount of time available to process the frame is $1 \div 60 = 0.0167$ s. With these rates, the RANSAC algorithm would only complete once every 18 frames, or at a rate of 3.33 FPS or less. Furthermore, it is possible that the homography might be computed 1000 times or more, reducing the FPS even more to 0.333 FPS or less. This, coupled with the delays present from transferring the image data from FPGA to SDRAM then from SDRAM via FPGA to PC, made the performance of the onboard homography solver too slow for real time purposes.

Table 4.5: FPU operation latencies [68], [69].

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycles Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>6</td>
</tr>
<tr>
<td>Multiply</td>
<td>5</td>
</tr>
<tr>
<td>Divide</td>
<td>17</td>
</tr>
<tr>
<td>Square Root</td>
<td>17</td>
</tr>
<tr>
<td>Convert</td>
<td>6</td>
</tr>
<tr>
<td>Fused Multiply-Add/Subtract</td>
<td>10</td>
</tr>
<tr>
<td>Move, Absolute Value, Negate</td>
<td>2</td>
</tr>
<tr>
<td>Compare</td>
<td>3</td>
</tr>
</tbody>
</table>
However, a PC with a clock frequency 32 times faster and a more efficient architecture for floating point and integer operations can compute RANSAC at least 32 times faster, meaning that it can complete the computation between a rate of approximately 11 and 110 FPS. Just communications delays caused from passing the image to memory and then out of the memory to the PC via USB already cause the FPS of the system to only go up to 14 FPS on the PC. Adding in the software for finding the best homography using RANSAC drops the frame rate down to 11 or 12 FPS at the worst, as was theorized above.

4.10 Results

For the first experiment a library of four images was made; each of the ColorChecker, but at different measured distances from the camera. The ColorChecker was used in order to demonstrate the usefulness of the color information for matching purposes since a grayscale image would cause many more false matches to occur. Each library entry has at most 128 descriptors. The FPGA is programmed and initialized and connected to the PC via USB in order to stream the images plus the match data to the PC for processing. Once on the PC, a homography is found using RANSAC and the homography is used to overlay a red rectangle over the border of the ColorChecker on the image.

A set of 400 images was captured over about a 30 second time frame, of which about 21% accurately identified the ColorChecker in the image. Although the communications delays mean that the number of frames per second which were accurately identified was only about 3, and assuming that this percentage scales with the amount of frames processed, the actual amount of accurately identified frames per second via the Color TreeBASIS matcher is likely around 12 FPS. This figure can likely be improved via an improved camera or at least improved camera processing that reduces the variation of exposure, gain, and white balance from frame to frame. The system would likely benefit from High Dynamic Range processing to balance out lighting extremes in the images, thus reducing the probability of washing out the image and also reducing the variation of the color values in the frames.

It is interesting to note that there is some lenience in the rotation and scaling of the object that is acceptable to still be able identify it, as can be seen in the images of Figure 4.18. Also, it is important to note that due to the ColorChecker being a grid, scale becomes invariant to a
Figure 4.18: Results of Hardware Color TreeBASIS for identifying the ColorChecker.
degree (basically as far as the $30 \times 30$ region contains the same information in the two scalings, as in Figure 4.19).

![Figure 4.19: Conditional scale invariance with right-angle Features. This shows two identical FRIs for the same image point at different scales. Even though the second FRI is of a smaller scale image, both FRIs are identical since the scaling is small.](image)

A second experiment was performed using four images of different book covers as the library (see Figure 4.20). Unlike the ColorChecker, these books contain much more feature information, ranging from 1100 to 2400 features. Therefore, a subset of 256 features of each image was used for the library entries. Unlike the regular grid of the ColorChecker used in the first experiment, each book (excluding the fourth) contains features that are highly sensitive to changes in scale, as would be expected for a feature found with FAST. Therefore, matching to these library entries is less tolerant of scale changes equating to an operating range of only 5 to 10 cm away from the original distance of the object from the camera when the library was generated.

A set of 200 output images was created for each library entry using the original rule set (see Chapter 3.2.3). For the first book, 20% of the output frames contained an accurate homography highlighting the book. Also, these homographies seemed a bit more precise with each highlighting box being an accurate fit over the book. The second book had a result of 30% of output frames containing an accurate homography. The third book saw a result of 45% of all frames containing an accurate homography. The fourth book, which contains very little color information and also a
highly repeated pattern, saw a poor performance of only 2.5% of the output frames containing an accurate homography.

Although changes in scale were not highly tolerated (only between 5 and 10 cm), affine and rotational transformations resulted in similar results to the first experiment, showing a decent amount of leeway in terms of the amount of object distortion caused by translations that are not scale-based. Also, the simplified rules set, as seen in Chapter 3.3, was tested and produced improved results for the first book (all other books achieved poorer performance with the simplified rules set). The results for the first book indicate that an accurate homography was assigned to the image about 72%, a vast improvement that possibly occurs due to the image containing many features of similar color tones which may have caused close, but not perfect, matches to be more acceptable, increasing the number of matches available for generating a homography.

4.11 Future Work

There are some modifications and improvements that can be made to the Color TreeBASIS hardware to make it a more robust, affordable, and easily deployed vision system. One issue with the system is that it is based off of the Helios platform [47], [48] (see Chapter 4.1) which is no longer being manufactured, has no customer support, and has a high price in terms of hardware costs. Since the platform is based off of the state-of-the-art technology of 2006, there are now alternatives that are much more cost effective and are commercial, adding support services from the manufacturer.

One potential platform for a Color TreeBASIS system is the MicroZed board from Avnet [70]. The MicroZed board comes in a few configurations, but the most appealing is the configuration that uses the Zynq 7020 FPGA [71] for approximately $300. The board contains header pins for user-designed daughter boards or a commercial Avnet I/O carrier card, USB and Ethernet headers, microSD card interface, 1 GB of DDR3 SDRAM, 128 Mb of QSPI Flash and the Xilinx FPGA. The FPGA contains 53,200 6-input LUTs, 106,400 flip flops, 560 KB of block RAM, and 220 DSPs. This platform has more memory, more connectors, more programmable logic, and is about $2000 less expensive than the Helios board. Furthermore, the Zynq 7020 FPGAs have two ARM processors that run up to a maximum of 800 MHz and have integrated floating point units,
Figure 4.20: Results of Hardware Color TreeBASIS for identifying four different book covers.
potentially allowing for the real-time processing of matches onboard as opposed to offloading the work to a PC.

If more logic or memory is required, a custom board based on the Zynq FPGA can be made at an accessible price point since the next level of Zynq FPGA, the 7030, has a price around $300 (much less than the $2000 Virtex 4 FX60). Even larger Zynq FPGAs are available with more than double the logic and memory of the 7030, though the price rises to $1000 or more and other limitations, such as power and cooling, might make it difficult to design a small portable platform around the FPGA.

Along with a new platform, a newer camera with better camera processing (either onboard the camera or user-controlled) is required to reduce the color and intensity variations of the captured frames over a wide range of environments and lighting conditions. By introducing high dynamic range (HDR) to the camera processor, colors and brightness should remain more constant in most environments and in most lighting conditions. Adaptive exposure, white balance, color correction, and noise reduction can make the image characteristics more uniform over all...
frames, potentially increasing the success rate of the matching algorithm. Future work will employ improved camera processing techniques to improve the quality of the frames captured for use in matching.

Another improvement that has been mentioned several times (see Chapter 2.2 and 3.4) is the modification of the feature detector and description algorithms to be rotationally invariant. Adding rotational invariance to Color TreeBASIS requires minimal changes to the overall algorithm since FAST is a rotationally invariant feature detector (see Chapter 2.2). The issue (as mentioned in 3.4) is that adding rotation of the FRIs pre-binary quantization makes the optimizations designed in Chapter 4.4 unusable. This is because the rotation of the FRIs makes the passing of data from one region to the next impossible since one pixel to the next might have a completely different rotation. In order to avoid this issue, rotation can be performed post quantization. However, the information contained in the 100-bit per channel 10×10 binary FRI is much less than the 7,200-bit per channel 30×30 8-bit FRI, meaning that the errors introduced because of rotation will be greater post-quantization. If these errors are too great, identical features, once rotated to a common orientation, might not generate matching descriptors. Future work will need to test the accuracy and repeatability of the descriptors when rotating post-quantization to determine if the hardware Color TreeBASIS will be able to support rotational invariance with all of the in place optimizations. Otherwise the binary quantizing module will have to be redesigned without the optimizations, requiring much more logic, in order to support pre-quantization rotation.

To improve the algorithm even more would be to make the detection and description scale invariant as well. Making Color TreeBASIS scale invariant is similar to making it rotationally invariant, only that it would require a completely different feature detector which is scale invariant [15], [23], [24], [26]. Creating a scale invariant hardware feature detector generally requires a lot of memory and a lot of logic and might not be able to run at fully-pipelined speeds. However, once generated, Color TreeBASIS would need to have scaling logic added to it, either before or after quantization. The scaling faces the same potential errors as rotation, thus pre-quantization is preferred in order to better preserve the data and keep the descriptors robust. Nevertheless, if scaling can be performed post-quantization, the Color TreeBASIS hardware can be kept intact and only the scaling (and rotation) logic must be added. Future work will test the possibility of
scaling post-quantization such that the rest of the Color TreeBASIS hardware can remain intact while making the feature descriptor and matcher more robust and feature-rich.

Another area of future work is the creation of a hardware method to calculate homographies using RANSAC. If this proves to be inefficient, either spatially or temporally, work can still be done to ensure that the homography calculation can be performed onboard the proposed hardware system using the embedded ARM A9 processors. Since the A9 processors have higher clock speeds than the current PPC405s, have integrated floating point units, and since the latency of floating point operations is less for each instruction, real-time RANSAC homography estimation onboard the MicroZed platform should be possible.

Finally, future work will include the exploration of practical applications of the Color TreeBASIS algorithm in a completely self-contained system. Applications may include navigation, tracking, localization, and object classification.
CHAPTER 5. CONCLUSIONS

The number and scope of vision applications in today’s world is ever increasing. Due to improvements in low-powered computing, many new and exciting real-time applications for computer vision applications, such as feature matching, are being explored in the realm of low-resource mobile computing. However, because of the rise in availability, affordability, processing power, and memory of mobile computers, such as tablets and smartphones, most recent research into resource limited computer vision has been in the realm of software. Although software is currently the more obvious choice for innovation in computer vision, it is important that one not forget the benefits and capabilities of a vision system made possible via hardware acceleration.

By moving algorithms that can be highly parallelized into hardware, true real-time processing can be achieved; even at high resolutions and/or high frame rates. The Color TreeBasis algorithm is one such perfect candidate for moving into hardware for improved processing speed. Also, being as resource efficient as it is, it can easily be placed into a reasonably priced FPGA development board (i.e. under $500) along with other hardware components needed to make a robotic vision system, such as an autonomous vehicle or an object detector/classifier.

Although the Color TreeBASIS tested used a library of objects for object detection (see Chapter 4.10), the matching module could quickly be replaced or augmented with a consecutive frame matcher for the purpose of computing optical flow for navigation. Furthermore, the addition of hardware to process matches with the purpose of forming homographies (see Chapter 4.11) or generating navigation information from image data could allow for increased precision automatons and even new, unimagined robotic vision applications.

The purpose of this thesis is to show that reliable, low-power, resource-limited, high-speed color feature matching is possible by using an FPGA based embedded hardware system. It is also a goal of this thesis to show that the hardware feature detector and matcher is accessible in terms of complexity and also in the affordability of platforms that can hold the vision system.
Chapter 2 presents the FAST feature detection algorithm, its hardware implementation (section 2.1), and proposes a method for assigning a dominant orientation to FAST features (section 2.2). Chapter 3 presents the Color TreeBASIS algorithm, its offline processing stages (section 3.1), its online processing stages (section 3.2), compares its results for frame-to-frame matching with single channel TreeBASIS (section 3.3), and proposes future work on Color TreeBASIS that includes rotationally invariant feature detection and description (section 3.4). Chapter 4 gives a brief overview of the complete Helios hardware platform (section 4.1), an in depth description of the Color TreeBASIS hardware (sections 4.2-4.7), an analysis of Color TreeBASIS’ resource usage (section 4.8), an overview of the software processing to estimate homographies (section 4.9), the results of the hardware Color TreeBASIS (section 4.10), and proposals of future work for Color TreeBASIS (section 4.11).

This thesis demonstrates the first real application of the hardware TreeBASIS algorithm while also improving the matching capabilities of the algorithm to include color images and also decreasing the amount of resources used as compared to its grayscale predecessor. It also demonstrates the first hardware implementation of the color FAST feature detector with non-maximal corner suppression. This work shows that a color-based feature detection and matching system is feasible and ideal for a low-resource reprogrammable hardware platform and that the results from such a system are reliable, robust, and available in true real-time at 60 frames per second. Using the Color TreeBASIS algorithm as a foundation, future work in object detection, target tracking, target following, and navigation in hardware could lead to a new generation of smaller, more efficient micro autonomous vehicles and more easily deployed (and affordable) security systems.

5.1 Contributions

- Fully-pipelined Color FAST Feature Detector in hardware with non-maximal corner suppression
- Proposed a method for finding a dominant orientation for FAST features
- Improved BASIS tree construction method
- Improved processing and resource efficiency of the TreeBASIS hardware modules
• Extension of the TreeBASIS algorithm and hardware to the Color TreeBASIS algorithm and hardware

• First implementation of TreeBASIS and Color TreeBASIS for use on a real FPGA embedded system

• First object detection system using Color TreeBASIS

• Real-time demonstration of success of Color TreeBASIS in the task of object detection
REFERENCES


80


[65] ———, LogiCORE IP DSP48 Macro v2.0 DS754, March 2011. 66

[66] ———, XtremeDSP for Virtex-4 FPGAs User Guide UG073 (v2.7), May 2008. 66


[68] Xilinx, APU Floating-Point Unit v3.1, March 2008. 68

[69] ———, PowerPC Processor with Floating Point Unit for Virtex-4 FX Devices XAPP547 (v1.0.1), Nov 2006. 68


[71] Xilinx, Zynq-7000 All Programmable SoC Overview DS-190 v1.5, September 2013. 72