ONBOARD VIDEO STABILIZATION FOR UNMANNED AIR VEHICLES

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Nicholas S. Cross

June 2011

COMMITTEE MEMBERSHIP

TITLE:	Onboard Video Stabilization for Unmanned Air Vehicles
AUTHOR:	Nicholas S. Cross
DATE SUBMITTED:	June 2011
COMMITTEE CHAIR:	John Seng, Ph.D.
COMMITTEE MEMBER:	Clark Savage Turner, Ph.D.
COMMITTEE MEMBER:	Aaron Keen, Ph.D.

Abstract

Onboard Video Stabilization for Unmanned Air Vehicles

Nicholas S. Cross

Unmanned Air Vehicles (UAVs) enable the observation of hazardous areas without endangering a pilot. Observational capabilities are provided by on-board video cameras and images are relayed to remote operators for analysis. However, vibration and wind cause video camera mounts to move and can introduce unintended motion that makes video analysis more difficult. Video stabilization is a process that attempts to remove unwanted movement from a video input to provide a clearer picture.

This thesis presents an onboard video stabilization solution that removes high-frequency jitter, displays output at 20 frames per second (FPS), and runs on a Blackfin embedded processor. Any video stabilization algorithm will have to contend with the limited space, weight, and power available for embedded systems hardware on a UAV. This thesis demonstrates how architecture-specific optimizations improve algorithm performance on embedded systems and allow an algorithm that was designed with more powerful computing systems in mind to perform on a system that is limited in both size and resources. These optimizations reduce the total clock cycles per frame by 157 million to 30 million, which yields a frame rate increase from 3.2 to 20 FPS.

Acknowledgements

I want to thank a few people for aiding me with completing this thesis. Dr. Seng for his support throughout the thesis. Without his guidance I would never have finished. David Johansen for developing and documenting a video stabilization algorithm. I'd like to thank him for spending time to personally discuss the algorithm with me. Randy Merkel for helping with ideas about optimizations and the continuous support throughout the years. George Kadziolka for his Blackfin programming guidance. I'd like to thank AeroMech Engineering, Inc. for providing me with hardware and a development license. I thank my family for their support and love. Last, but not least, I'd like to thank Laurie for her endless support, patience, and love throughout the entire thesis.

Contents

Contents vi

List of Tables viii

List of Figures x

1 Introduction 1

1.1 Motivation . . . 1

1.2 UAV Challenges . . . 2

1.3 Proposed Solution . . . 3

1.4 Overview of Chapters . . . 4

2 Background 5

2.1 Video Stabilization . . . 5

2.1.1 Motion Estimation . . . 5

2.2 Blackfin . . . 8

3 Related Work 9

3.1 Stabilization Methods . . . 9

3.2 Embedded Processor Comparisons . . . 11

3.3 Blackfin Optimizations . . . 12

4 Solution 14

4.1 Base Stabilization Algorithm . . . 14

4.1.1 Feature Selection . . . 15

4.1.2 Feature Tracking . . . 17

4.1.3 Frame Motion Estimation . . . 20

4.1.4 Frame Movement . . . 21
4.2 Blackfin Optimized Algorithm . . . 23
4.2.1 Algorithm Overview . . . 23
4.2.2 Optimizing Memory . . . 24
4.2.3 Deinterlacing and Asynchronous Frame Capture . . . 30
4.2.4 Feature Selection . . . 34
4.2.5 Feature Matching . . . 41
4.2.6 Motion Estimation . . . 46
4.2.7 Intended Motion Calculation . . . 47
4.2.8 Offset And Display The Image . . . 48
5 Results **50**
5.1 Stabilization Results . . . 50
5.1.1 Binary Corner Detection . . . 50
5.1.2 Parabolic Fit Camera . . . 51
5.1.3 Stabilization Offsets . . . 57
5.2 Optimization Results . . . 58
5.2.1 Source Videos . . . 58
5.2.2 Searching Middle Pixels . . . 58
5.2.3 Grayscale Conversion . . . 59
5.2.4 Memory Movement . . . 63
5.2.5 Memory Placement . . . 65
5.2.6 Sum of Absolute Differences . . . 67
5.2.7 Deinterlacing . . . 70
5.2.8 Loop Unrolling . . . 72
5.2.9 Branch Prediction . . . 73
5.2.10 Transfer Next Frame in Background . . . 74
6 Conclusions and Future Work **79**
6.1 Conclusions . . . 79
6.2 Future Work . . . 81
Bibliography **92**

List of Tables

4.1 L3 Memory by Sub Bank 25
4.2 L3 Sub Bank 0 and 1 Memory Placement 27
4.3 L3 Sub Bank 2 and 3 Memory Placement 28
4.4 L2 Memory Allocation 29
4.5 `SAA` Macro 44

5.1 Frames Per Second of Algorithm With Different Search Sizes . . . 59
5.2 Converting a Field from Color to Grayscale in Millions of Cycles . 61
5.3 Comparing `Memcpy` vs MDMA L3 to L1 transfers in Millions of Cycles 63
5.4 Optimizing memory placement 65
5.5 Comparing Single Execution of Different SAD Algorithms in Cycles 67
5.6 Comparing Different SAD Algorithms in Million of Cycles 68
5.7 SAD vs Pyramid Template Matching in Million of Cycles 69
5.8 Algorithm Runtime With or Without Deinterlacing enabled in Millions of Cycles 70
5.9 Loop vs Unrolling a Loop in Millions of Cycles 72
5.10 Branch Prediction 74
5.11 Wait time Between Frames in Millions of Cycles 74
5.12 Algorithm Clock Cycles with Transferring Next Frame in Background vs Not Transferring 76
5.13 Algorithm and Frame Capture Time in Millions of Cycles 77

6.1 How Much Each Optimization Reduced Clock Cycles per Frame in Millions of Clock Cycles 80

List of Figures

2.1 Example of Feature Matching 7

4.1 Base Algorithm Flowchart 14
4.2 Pyramidal Template Matching 19
4.3 Blackfin Adapted Algorithm Flowchart 23
4.4 NTSC (4:2:2) Frame 26
4.5 Interlaced Frame 31
4.6 Deinterlaced Frame 32
4.7 Asynchronous Frame Capture Part 1 32
4.8 Asynchronous Frame Capture Part 2 33
4.9 Generating the Binary Image 34
4.10 Using the Binary Image to Find Corners 38
4.11 Leapfrogging `SAA` and `SAAR` Calls 45

5.1 Grayscale image of one field 51
5.2 Smoothed image of one field 51
5.3 Binary image of one field 52
5.4 Vertical Vibration - Intended Vs Unintended Motion. 1 Before Frames, 1 After Frames 53
5.5 Vertical Vibration - Intended Vs Unintended Motion. 8 Before Frames, 1 After Frames 53
5.6 Horizontal Vibration - Intended Vs Unintended Motion. 4 Before Frames, 31 After Frames 54

5.7 Vertical Vibration - Intended Vs Unintended Motion. 4 Before Frames, 31 After Frames 54
5.8 Vertical Vibration - Intended Vs Unintended Motion. 1 Before Frames, 31 After Frames 55
5.9 Vertical Vibration - Intended Vs Unintended Motion. 8 Before Frames, 31 After Frames 56
5.10 Vertical Vibration - Intended Vs Unintended Motion. 4 Before Frames, 1 After Frames 56
5.11 Stabilization Offset Per Frame 57
5.12 Grayscale Conversion with or without MDMA Enabled 62
5.13 L3 to L1 conversion with or without MDMA Enabled 64
5.14 Full frame processing with different memory mapping 66
5.15 Trend lines in random memory mapping 66
5.16 Comparing SAD Algorithms 68
5.17 Comparing SAD vs Pyramid Template Matching 70
5.18 Frame Processing With vs Without Deinterlacing Enabled 71
5.19 Loop vs Unrolling a Loop 73
5.20 Wait Time Between Frames in Millions of Cycles 75
5.21 Algorithm Clock Cycles with Transferring Next Frame in Background vs Not Transferring 76
5.22 Algorithm and Frame Capture Time in Millions of Cycles 78

Chapter 1

Introduction

1.1 Motivation

Unmanned Air Vehicles (UAVs) are an increasingly important part of our lives. They are useful for surveillance during military exercises [41], firefighting [31], weather monitoring [33], police work [39], and border patrol [41]. The main purpose of UAVs is to provide video surveillance. Since UAVs are unsteady platforms, video stabilization is a critical asset for providing high quality video.

Video processing is a growing field. In the past the only video processing was used for machine vision. These days nearly every video device has video processing. These algorithms may include background or foreground replacement for webcams, video stabilization for home video cameras, movement detection for security cameras, and many others. With the popularity and speed of embedded processors and DSPs these are all possible on our everyday devices like cell phones, laptops, and cameras.

New video processing hardware and algorithms also affect how UAVs utilize

video processing. Video stabilization, target tracking, mosaicing, fusion, and other image enhancement algorithms are available to UAV operators onboard or on the ground. This thesis provides an onboard video stabilization solution to help provide quality intelligence to UAV operators in safety critical situations.

1.2 UAV Challenges

Unmanned Air Vehicle (UAV) video typically has a lot of problems associated with it. Unexpected and quick movement of the platform causes the video camera to move in unwanted ways. UAVs typically have smaller and lighter gimbals to control their video cameras [25]. These gimbals provide mechanical stabilization, but since they are lighter and made for UAVs, they typically only stabilize up to 100 μradians [25]. This means that even with mechanical stabilization, engine vibration shakes the video frames and distracts operators.

In the past, video stabilization and other video processing are done on the ground. Ground based systems have the advantage of processing the video with fast CPUs and memory systems. The radio frequency (RF) link adds complications to ground based processing. The RF link adds noise to the image. Noise makes features harder to track because noise adds occlusion and hides features from being matched from frame to frame. If video processing is providing outer loop control to the UAV system, RF latency becomes a problem because typically, RF adds about 200ms of latency to a transmission trip time.

Putting the video processing onboard the UAV alleviates these two problems. The video is captured by the video processing board without any additional RF noise. This provides a clear picture to aid the video processing algorithm. If the algorithm is providing outer loop control, then there is no latency between the

video processing module and the receiver of the control messages.

Switching to an onboard embedded solution seems like an obvious solution for handling UAV video processing. It does come with its own disadvantages. The processors available for integrating onto a UAV are limited. Processor speed, power requirements, memory speed, size, and weight are all additional problems created by using an embedded processor on the UAV.

A Blackfin processor meets these requirements. It is a fixed point DSP that has a core clock speed of 600 MHz. The low power requirement is met because it is a DSP that does not natively support floating points operations. This means that it does not need to draw the extra power required for extra floating point related chips. The modules are small and the weight is negligible. The chipset allows parallel instructions, circular addressing, and register reversal that speeds up algorithms [15]. These specifications are not enough to run a stabilization algorithm that was designed with desktop computing power in mind. Section 1.3 explores what this thesis does to solve this problem.

1.3 Proposed Solution

This thesis modifies an algorithm that is not designed with hardware limitations in mind and optimizes it for a Blackfin processor's architecture. These optimizations includes strategic memory placement, architecture specific peripheral use, builtin in macros that reduce clock cycles for mathematical functions, and other adjustments that consider the architecture and platform.

These optimizations must improve the performance of the algorithm without affecting the stabilization performance. The stabilization results show that the

algorithm stabilizes vibration and unintended motion even with the optimizations enabled.

1.4 Overview of Chapters

Chapter 1 introduces the topic matter and the thesis. Chapter 2 defines some terms and background information for the reader. Chapter 3 describes other solutions for video stabilization and other implementations of Blackfin specific optimizations and algorithms. Chapter 4 showcases the solution to this problem. Chapter 5 goes through the stabilization and optimization results. Chapter 6 discusses the conclusions and future work from this thesis.

Chapter 2

Background

2.1 Video Stabilization

Digital video stabilization is the process of removing unwanted movement from a video stream. Digital video stabilization is different than mechanical and optical stabilization. Mechanical stabilization physically dampens out vibration or unintended movement with gyroscopes. Optical stabilization is when sensors detect mechanical movement and shift the lens slightly. Both of these methods stabilize the image before it is converted to digital. Digital video stabilization modifies each input frame to maintain a steady image after it is converted to digital. A transformation matrix is calculated via different motion estimation methods and is used to move the image.

2.1.1 Motion Estimation

Motion estimation is required to determine how much offset there is between two images. There are two types of motion estimation: indirect and direct. Indi-

rect [40] and direct [18] are compared and contrasted in two works by Microsoft Research Labs. The next section describes the differences.

Direct motion estimation algorithms utilize every pixel to calculate motion information [18]. These algorithms use "measurable image quantities", like brightness, at each pixel in the image to acquire information [18]. Direct methods are useful because they give high sub-pixel accuracy and handle outliers well [18]. Direct methods compare the spatial derivatives of each image [18]. They are then compared using an affine model [18]. The simplest forms of direct motion estimation work when there is only a small movement between frames because the first assumption is that the displacement is minimal [18].

Since video tends to have greater than one pixel difference between frames, an iterative approach is often taken to calculate motion in steps [18]. A course-to-fine iterative estimation, or a pyramidal algorithm, transforms the original image into a course image using a Gaussian blur [10]. These course images are used to compare via a direct method. The frame is then warped by the value calculated between the course images [10]. A finer version of the course image is compared next. This repeats five times and it is able to get the total movement between the frames because it can calculate small jumps at each step [10]. This improves the maximum distance for a match up to $10 - 15\%$ [18].

An example of a direct method is optical flow. Optical flow commonly uses the Lucas-Kanade approach [29]. Optical flow uses the direct method approach of looking at each pixel in the frame, but it uses least squares to determine the velocity between each frame with respect to time [29].

Indirect motion estimation algorithms use features to calculate motion information [40]. They rely on finding good correspondence between strong features

[40]. Harris Corner Detection is a common feature detection algorithm [40]. This uses corners to find locations on the image that have a high chance of also being in the next frame [21]. Indirect methods need filters because each feature has a unique vector calculation [40]. RANSAC is a commonly used method to filter out outliers [40]. Unlike direct methods, indirect methods handle different lighting conditions well because they are not brightness based [40]. Also indirect has individual errors per feature that are averaged and filtered out [40]. This helps prevent global error offsets that can happen using a direct method [40].

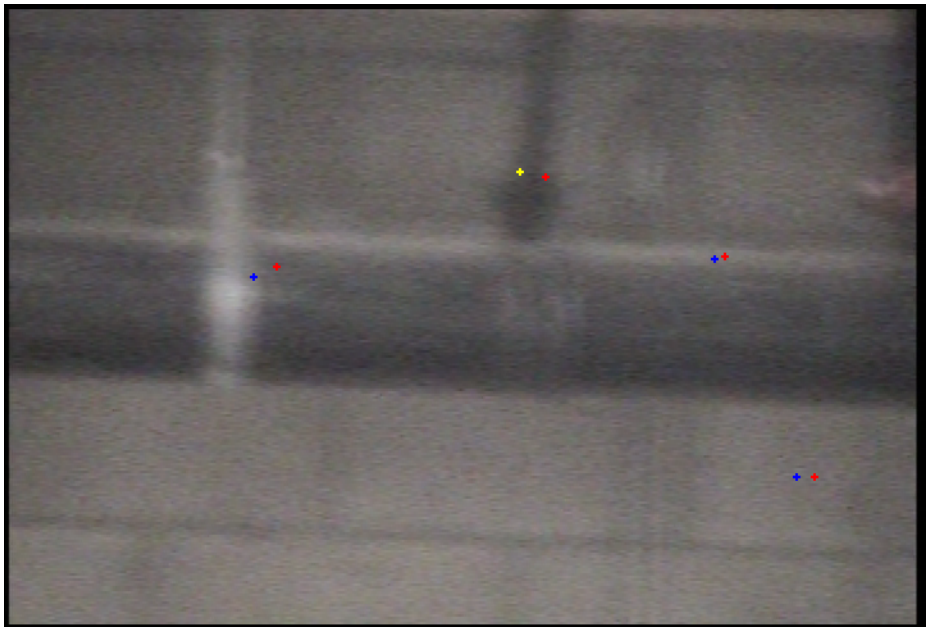

Figure 2.1: Example of Feature Matching

Figure 2.1 shows four features that have found a match between frames. The red dots are original locations and the colored dots are their new location on the current frame. The one outlier needs to be filtered.

Both methods use comparison metrics like Sum of Absolute Differences, Sum of Squared Differences, and Mean Squared Error [18]. Although these algorithms

output different results, they all output values that are comparisons between sets of values. The calculated differences help determine where the best match is.

2.2 Blackfin

The Blackfin® processor is a Digital Signal Processor (DSP) that is made by Analog Devices®. The Blackfin 561 has two 600 Mhz processors [5]. It has two 16-bit ALUs which are capable of executing 1 to 4 instructions per cycle [5]. There are two Parallel Peripheral Interfaces (PPI), a SPI port, SPORT and other interfaces [5]. Video is read in through a decoder [4] that uses one of the PPI ports [5]. Video is output through an encoder [3] that uses the other PPI port [5]. This is advantageous because the 561 processor can write and read video frames at the same time [5]. The 561 is unique because other Blackfin DSPs do not have two PPI ports so they have to alternate input and output.

The Blackfin 561 is a fixed point DSP [5]. That means it does not natively support floating point operations. Integer and fixed point calculations are faster on a fixed point DSP, but floating point calculations require function calls and are much slower [7].

The Blackfin has three different levels of memory. These range from large and slow to small and quick. These are discussed in section 4.2.2.

Chapter 3

Related Work

3.1 Stabilization Methods

Developing video stabilization algorithms for Unmanned Air Vehicle (UAV) video presents unique challenges. The background is moving, Radio Frequency (RF) noise can be a problem, and there are independently moving objects in the screen. Stabilization algorithms are an important part of video processing. This section reviews the state of the art.

Digital stabilization algorithms require object tracking and motion estimation algorithms to calculate the difference between frames. Some algorithms assume a static background. Sun et al. [20] assume a background is static and use a modified version of background subtraction. Background subtraction makes color and contrast maps out of the background and foreground and compares them to remove the background [20]. Background subtraction works well for removing most of the background, but it misses some areas and leaves markings along the edge of the foreground because those are high contrast parts [20]. Sun et al.

[20] propose a method called Background Cut. This method adapts the standard background subtraction and removes the additional contrast issues by subtracting the contrast of the background image from the contrast of the original image [20]. Yu et al. [42] also use a static background and background subtraction to generate object data, but minimize energy at each pixel compared to its neighbors to detect what is foreground and background. This combined with a color gradient allows them to track the foreground and background separately. These algorithms are made for surveillance cameras, sporting events, and webcam software.

Stabilization with direct methods of motion estimation are common. One method to stabilize video is to find the displacement between two frames. Kumar et al. [25] present a solution that does not use features. It uses an adaption of optical flow that is presented in [9]. Optical flow algorithms, like the Lucas-Kanade algorithm [29], use the difference in brightness between two frames to estimate motion. The adapted version presented in [9] uses an iterative course-to-fine method to estimate the difference between each frame. It then uses the captured frames and "stitches" them together to make a mosaic [25]. This creates a stabilized image by increasing the field of view and painting the field of view with a moving camera. This keeps all objects in the same exact location.

Feature based stabilization methods are also popular. Dave Johansen uses features to stabilize UAV video with Harris Corner detector and an affine model [21]. Another approach is to use "ego-motion compensation" combined with an affine model [12] [22] to determine the difference between two frames by isolating the independent movement and filtering it out from the global movement estimation. The independent movement is determined using a procedure called Adaptive Particle Filtering [22] which helps cull outliers.

Lee et al. [26] combine feature based and direct by using an optical flow like

approach but using features to track instead of brightness. Their solution finds features by calculating a Laplacian image and then searching through the pixels to find the local minimums and maximums [28]. It then makes trajectories out of pairs of points that share edges [26]. It matches using the same properties described in [28], but it combines it with an outlier detection method.

The outputs generated by these algorithms are different. Some digital stabilization methods mask the vibration instead of correcting it. Sun et al. [20] applies a Gaussian blur with a variance of 2 if the translation is less than 4 pixels. Some stabilization algorithms account for rotation and scale. Many solutions warp the image to attempt to maximize the screen use [25] [26]. Dave Johansen also accounts for rotation and scale using an affine model [21].

3.2 Embedded Processor Comparisons

There are a few different approaches to embedded video processing. One approach is to use two Digital Signal Processors (DSPs). Another is using one core for the DSP and one for the Microcontroller Unit (MCU). The last is to use a FPGA in combination with a MCU. The setup logically separates the video processing core from the core that handles I/O or runs an operating system.

The pure DSP solution investigated in this comparison is the Blackfin 561 processor. A mixed solution is the Texas Instrument's DaVinci chip with an arm processor. An Altera FPGA is another solution for video processing [2]. All three methods process video successfully and have their own advantages and disadvantages.

Ease of programming is an important category to consider. The Blackfin and

DaVinci DSPs are easier to program than a FPGA [11]. Unlike the DaVinci, the Blackfin uses the same DSP core and instruction set for its DSP and "MCU". The ARM processor uses a different instruction set than the DaVinci processor so that combination requires two different instruction sets to program compared to single set for the Blackfin DSP.

FPGAs are very strong because they are highly parallelizable compared to the Blackfin and DaVinci chip [2]. That is advantageous because a lot of video processing algorithms are easily parallelized. Performance is important for video processing. In 2007, the Blackfin outperformed the ARM i.MX processor in every category [14]. This includes MPEG encoding and decoding [14]. FPGAs outperform the Blackfin if the algorithm is in fixed point [11].

3.3 Blackfin Optimizations

The Blackfin processor has been used before for video processing applications. It is a powerful DSP, but it requires optimizations to process video effectively.

Memory placement is an important approach to optimizing video processing on the Blackfin. Memory location has been shown to influence algorithm run time by utilizing different sub banks [27] [16] [30]. Getz uses different memory banks to increase the speed of a dot product on a Blackfin [16].

A few solutions optimize their algorithms with memory movement. Li et al. [27] demonstrate that the Blackfin processor can be used for a video network surveillance system. Li et al. [27] use caching and DMA to achieve an increase in speed by 50%. The Blackfin is also used to handle safety critical sensors for cars. Katz discusses how Blackfin video processing is used to warn the driver if

the car is departing its lane and to aid in collision avoidance [24]. Katz credits the Blackfin's ability to process these algorithms because it can transfer memory in the background using MDMA [24].

To get video to properly execute, Ajay et al. present a solution that requires two Blackfin processors in conjunction to process video [1]. They ping pong DMA by working on one frame while capturing the second frame [1]. The Blackfin can also capture active video only [24]. This saves precious clock cycles every frame [24]. Sanghai et al. suggest also incorporating MDMA into an algorithm if using caching to avoid page misses [37]. It is also suggested to make sure that traffic control is on. That means for every MDMA or DMA channel, do not alternate the direction of memory transfer in respect to memory banks. If possible, transfer all the data that needs to go in one direction, then move all of the information that needs to go the other or have different channels for different directions. This matters more on a BF533 because it only has one PPI. The 561 reads and writes to two different PPIs at the same time to capture and display video frames at the same time [37]. Sanghai et al. [37] also recommend to use multiple DMA and MDMA controllers and spread out between them.

Processor architecture is also used to improve performance. Li et al. [27] used built in instructions, zero overhead looping to achieve 30% increase in speed. Getz warns that hand writing assembly can make things slower if the architecture is not taken into consideration [16]. The Blackfin is also a fixed point DSP. Utilizing fixed point math greatly improves algorithms that require extended precision [8].

Chapter 4

Solution

4.1 Base Stabilization Algorithm

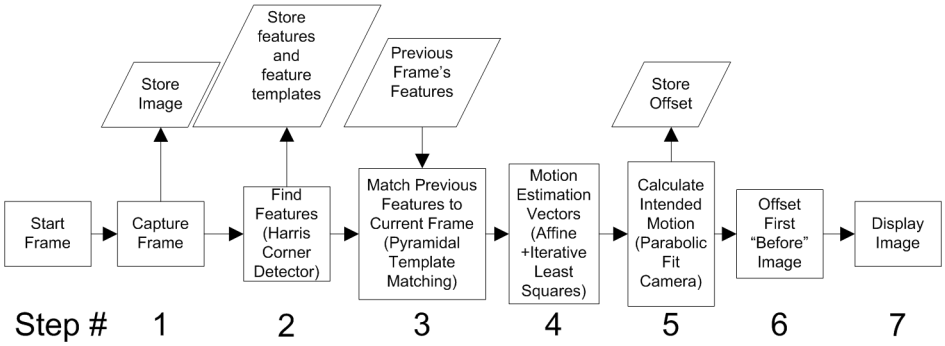

Figure 4.1: Base Algorithm Flowchart

The algorithm in David Johansen's master thesis [21], "Video Stabilization and Target Localization Using Feature Tracking With Small UAV Video" inspired the solution presented in this thesis. The following sections describe the original algorithm. Section 4.2 describes how the algorithm is optimized for embedded processing.

4.1.1 Feature Selection

This stabilization algorithm requires features to estimate motion between frames. This is the step 2 in 4.1. Since features are required to persist from frame to frame, it is a good idea to have a feature selection method that will choose parts of the image that will likely not change in appearance from frame to frame. An ideal feature is distinct and unique, is identifiable in the next frame, and provides new information to the stabilization process [21].

An identifiable and unique feature is a feature that will not create a false positive match with another feature in the next frame. This is important because many features, such as a feature on a line, will look identical throughout the entire line. This means that the algorithm may choose a different part of the line and provide invalid movement vectors. The vector outlier rejection algorithm prunes out fewer invalid movement vectors if there are a lot of false positives. Even worse, it may cull valid movement information if there are enough false positives.

The requirement where a feature must exist from frame to frame may seem obvious. The algorithm creates new features to track every frame and compares those features with just the next frame. This is done to minimize features disappearing due to occlusion and features leaving the field of view [21]. Noise creates features that may not exist in the next frame [21]. Choosing what noise reduction algorithm to use helps ensure that features shall exist from frame to frame.

Features must provide new information to the algorithm. This is important because a single portion of the image may influence the entire frame. This may not be a problem, but if the complex area on the image is an independently moving object, the stabilization will fail.

Corners are tracked for this algorithm because they are affected less by rota-

tion, noise, and are usually associated with objects of interest [34]. There are a few steps in determining features. The next sections describe how a feature detection algorithm assigns a rating to each potential feature and how each feature brings unique information to the algorithm.

Feature Rating Methods

Johansen presents a few different methods to select features from an image [21]. He compares Gradient Difference, Canny Edge Detection, Forstner Interest Operator, Harris Corner Detection, and Binary Corner Detection (BCD) [21]. Gradient Difference and Canny Edge Detection give high ratings to lines [21]. Since lines are not unique features, these features are not useful because they are likely to have a false positive match in the next frame [21]. Forstner Operator and Binary Corner Detector handle video noise well [21]. Harris Corner Detector identifies some false positives in video noise [21]. Forstner identifies too many false positives to be used effectively [21]. Binary Corner Detection does not differentiate between a strong and weak feature [21]. The rating method of Harris Corner Detector makes it a more desirable feature selection method than BCD for this algorithm because BCD does not have a rating system [21]. The algorithm presented in [21] includes a noise reduction step to account for Harris Corner Detector's false positives due to video noise.

Feature Filtering

Each feature must bring new information to the algorithm. Clusters of features put an emphasis on a certain portion of the image and may affect stabilization performance [21]. In [21], two feature separation filters are explained:

region based and minimum separation distance. Region based separates the image into different sections. The algorithm uses the best feature per region for its movement calculations and filters out the rest. Region based filtering still allows clustered features in the case where a region-dividing line goes through a complex, feature ridden part of the image [21]. A region based filter also requires a feature per region which forces weak features into the feature list [21]. This creates false positives and may affect the stabilization performance. A minimum separation filter forces distance between each feature. A minimum separation algorithm requires more computation time, but it prevents clustering of features [21].

4.1.2 Feature Tracking

Once the features are selected and recorded, the next step (Step 3 on 4.1) is to wait for the next frame. Once the next frame is available, the algorithm searches for the previous frame's features on the new frame. These matches are used to generate movement vectors that are combined to estimate motion.

Types of Feature Tracking

Johansen compares three feature matching algorithms: profile matching, template matching, and optical flow [21].

Both profile and template matching use a feature's surrounding area from the previous frame and compare it to the surrounding area in the new frame. These surrounding areas are boxes filled with intensity values. They both utilize comparison algorithms like Sum of Absolute Differences and Sum of Squared Differences to compare blocks [21]. The difference between the two is that profile

matching does an exhaustive search looking for a match between two blocks of memory whereas template matching looks for an approximation. Profile matching creates an identical sized window around each potential new location in the surrounding area and compares against the original feature window. It then compares the difference calculated against each other spot. The position with the least difference is the best match and is probably the feature's new location. Template matching is when a lower resolution of the previous window is compared to a low resolution window around potential new positions. It then increases resolution at each step until it finds an acceptable match [21].

Optical flow is defined in [9]. It is used to "estimate the motion that occurred at the pixel of interest and can be used as an estimate of feature motion [21]." Optical flow is not suited for UAV video due to the high number of features to track [21]. It only can measure a small distance as well [9]. Using advanced pyramidal iteration techniques, optical flow is more useful because it can calculate greater distances [18].

Johansen uses template matching because UAV video tends to have high amounts of noise and it performed better than optical flow [21]. It also is more accurate than profile matching [21].

Johansen describes a procedure called Pyramidal Template Matching that is outlined in section 4.1.2.

Pyramidal Template Matching

Johansen applies a pyramidal technique to improve performance of any of the methods featured in section 4.1.2. A pyramidal technique checks a course resolution to see if the new location has a possibility of being a match. If the

difference between the two windows is less than 1.25% different than the original location, a larger window is checked at that same location. This repeats five times and if each iteration passes, then the spot is a known match. If it fails at any level, it then rejects the potential new location earlier than later. A 1.25% maximum error rate rejects nearly all invalid features while rejecting less than 5% of the valid ones [21].

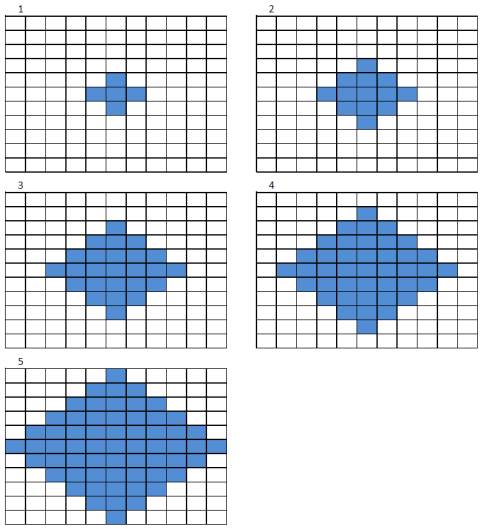

Figure 4.2: Pyramidal Template Matching

Figure 4.2 shows the small to large comparisons that makes up the pyramidal template matching. It first checks the middle four pixels, then iterates through four more comparisons. If the largest one has a difference less than 1.25%, it is a valid match.

After a feature finds its match, the `x` and `y` locations of both the previous and current frame are stored in an array. Once all of the features locations are captured they are inputs into the frame motion estimation algorithm discussed in section 4.1.3.

4.1.3 Frame Motion Estimation

The next step in the algorithm is to utilize the features' new and old locations to create movement vectors. This is step 4 in 4.1. A translation model and an affine model is described in [21]. He also discusses different outlier filters [21]. The next sections describe these different algorithms.

Models

The first method of estimating global motion is using a translation model. The translation model computes the `x` and `y` translation between each frame. For each direction, it takes the distance between each new and old location and sums them up. This means that it does not account for rotation.

When the filtering algorithms described in section 4.1.3 are applied, clustered features affect the translation model. If there are more features in a certain area or on a certain object than the rest of the frame, then averaging will give a higher priority to that area and the inliers are culled.

The second method is the affine model combined with least squares. This calculates rotation, scale, and translations for both axes [21]. The steps to setup and calculate the iterative least squares solution is described in [21].

Vector Filtering

Two vector filters, Iterative least squares and RANSAC, are described in [21]. Iterative least squares is the process of determining the least square solution and using the standard deviation to cull outliers [21]. RANSAC is a method that uses random samples from the selection of vectors to estimate the culling threshold

[21]. RANSAC requires a minimum amount of data points required per frame [21]. The vectors that fall within that culling threshold are used to estimate the full frame motion.

Noise affects each method differently. Non-zero mean noise is successfully filtered with RANSAC, but biases incorrectly when applying iterative least squares [21]. Zero mean noise is successfully filtered by iterative least squares but culls too many valid vectors in RANSAC [21].

Conclusion

Johansen does not use the translation model because it is not fit for UAV video and does not handle rotation [21]. He does not use RANSAC because it takes too long to operate and does not provide much advantage over iterative least squares [21]. Utilizing an affine model with iterative least squares provides the best combination of stabilization performance and runtime [21].

4.1.4 Frame Movement

The next step (number 5 in 4.1) is to move the frame by the calculated offset to stabilize the image. A Parabolic Fit Camera (PFC) algorithm is recommended over a PID loop and low pass filter [21]. This is because a PID loop must be calibrated for each type of expected video input and like a low pass filter, will create large undefined regions [21]. The PFC is described in section 4.1.4.

Parabolic Fit Camera

Stabilization algorithms can make UAV video worse than the original. This happens when `x` and `y` translations are moving quickly in the same direction for an extended amount of time. The video is jerky because it will stabilize then "reset" periodically. Stabilization is smoother if it considers intended motion. The Parabolic Fit Camera (PFC) is a filter that takes input data and smooths it to match intended motion. The idea is that each new data is fit to a parabolic shape defined by two parameters which changes how the algorithm responds to input. The parameters are the number of "before" frames and the number of "after" frames. The "before" frames are frames which have not been processed by the parabolic fit camera. Each frame enters a buffer of size "before". The algorithm buffers each frame because each frame includes motion information which helps smooth out incoming vibration. As the "before" number increases, the video frame delay increases, but the algorithm improves its motion correction. The delay is shorter with a lower "before" number. The reduction in information decreases motion estimation.

The "after" parameter is the number of frames which have already been displayed. These are for historical value and are used to predict the future from the past. A longer number predicts repetitive motion better, but makes abrupt movement less correctable. A lower number allows the algorithm to more quickly respond to new movement.

Once enough "before" and "after" frames are captured, every new frame uses the parabolic fit camera to calculate the intended location for that frame. This filter queues intended motion calculated and is used when it is at the end of the "before" frames. That means the frame is stored until then and is offset by the

difference between the calculated and intended motion. The output frame is then stabilized while considering the intended motion of the video.

4.2 Blackfin Optimized Algorithm

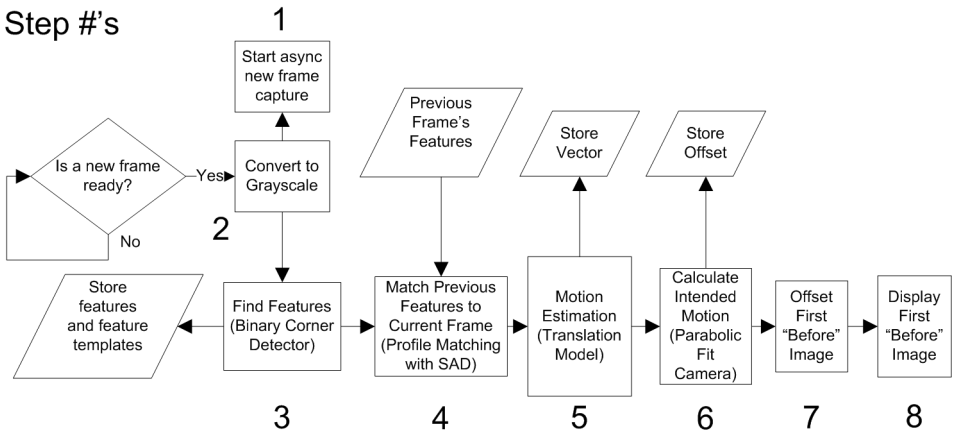

Figure 4.3: Blackfin Adapted Algorithm Flowchart

4.2.1 Algorithm Overview

This solution adapts David Johansen's algorithm described in section 4.1 for a Blackfin embedded DSP. Figures 4.3, 4.7, and 4.8 show the new algorithm. The multimedia template referenced in [36] is the template for the framework of this solution. Sanghai's solution setup the video decoder and encoder, input and output buffers, and processor setup code [36]. This section describes the differences between this new algorithm and its base algorithm depicted in figure 4.1. It also covers optimizations made to the algorithm to harness the advantages of the Blackfin DSP. Memory optimization is discussed in section 4.2.2.

The algorithm shown in figures 4.3, 4.7, and 4.8 is the Blackfin adapted algorithm. The algorithm has eight total steps. This thesis describes those eight steps through six major steps. Setting up the asynchronous capture of the next frame is the first step. This is discussed in section 4.2.3. The second step is to identify features and is described in section 4.2.4. The third step uses the features found in the previous frame to find matches to determine new locations in the current frame. This process is described in 4.2.5. The motion of the frame is then estimated by summation of the displacement vectors of each matched feature. This is detailed in section 4.2.6. This data is gathered and used to predict intended camera motion so the frame can be offset to remove unintended motion. This is estimated through a parabolic fit camera which is described in section 4.2.7. Section 4.2.8 describes the last step which is applying the found offset and displaying the image to the screen.

4.2.2 Optimizing Memory

Since memory is the bottleneck in the video processing pipeline, optimizing memory placement and movement is a very important part of Blackfin's ability to optimize execution [38]. The next sections describe how memory is utilized for this solution by strategically positioning chunks of memory and transferring memory utilizing the Blackfin core.

Memory Placement

The Blackfin 561 has three levels of memory: L1, L2, and L3. L1 is 100KB of internal memory for instruction data, and scratch pad memory [5]. L2 is 128KB of low latency memory [5]. L3 is 64MB of low speed memory which uses the system

clock (SCLK) to transfer data [5]. Memory placement is important because "the processor core and DMA controller can access different sub banks of memory in the same cycle, but when these resources attempt to access the same sub bank in the same cycle, one of the accesses must stall [7]."

Memory in L3 and L2 are accessible to both cores [5]. This solution uses L2 memory to transfer state information from the core processing the stabilization algorithm to the core processing IO. Each core has its own L1 memory that is not sharable [5].

L3 Memory

There is 64MB of L3 on a Blackfin 561 [5]. L3 is split into four 16MB banks [5]. The algorithm utilizes 16MB of the 64MBs of L3. Figure 4.1 shows how much space is used in each bank. The memory is split up between the different banks to increase algorithm performance.

Bank	Total Size in MB
SDRAM 0	2.70MB
SDRAM 1	2.21MB
SDRAM 2	2.91MB
SDRAM 3	8.83MB

Table 4.1: L3 Memory by Sub Bank

This processing board uses the National Television Standards Committee (NTSC) for input video. NTSC describes the frame height and width. NTSC is 525 rows and 858 columns of video information [19]. The actual frames of video are 486 rows by 720 columns [19]. Horizontal and vertical blanking make up the majority of the non-image based information in a NTSC frame [19]. There are also Start of Active Video (SAV) and End of Active Video (EAV) rows and

columns as well [19]. Figure 4.4 shows how the NTSC frame is formatted.

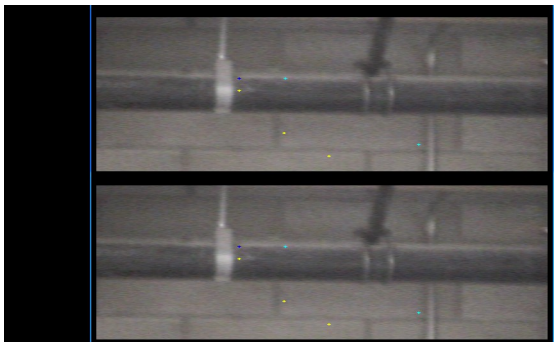

Figure 4.4: NTSC (4:2:2) Frame

UYVY (4:2:2) is a color spaced format for digital component video [19]. 4:2:2 means that for every two horizontal Y samples, there is one sample each of Cb and Cr [19]. That means a NTSC in UYVY video frame is 525 rows by 1716 columns. Without blanking included, a standard definition frame in UYVY format is 486 rows by 1440 columns. Cb and Cr are combined with Y their respective Y value to create a RGB color [19]. U is Cb, V is Cr, and Y is its respective luma value [19]. UYVY is the order that the Blackfin's video ADC outputs the color in [4]. Equations 4.1, 4.2, and 4.3 shows how to convert UYVY into RGB.

$$R = 1.164(Y - 16) + 1.596(Cr - 128) \tag{4.1}$$

$$G = 1.164(Y - 16) - 0.813(Cr - 128) - 0.391(Cb - 128) \tag{4.2}$$

$$B = 1.164(Y - 16) + 2.018(Cb - 128) \tag{4.3}$$

This means to store a full image it requires 699840 bytes and each output frame is 900,900 bytes. This forces history frames and other full frames to be stored in L3. Tables 4.2 and 4.3 show how each memory bank is segmented.

These memory locations are chosen to optimize memory placement. Memory banks and MDMA channels operate faster if the memory banks are not accessed

Sub Bank	Purpose	Size in Bytes
SDRAM 0	Frame In 0	900900
SDRAM 0	Frame In 2	900900
SDRAM 0	Translation Frame	900900
SDRAM 1	Frame Out 0	900900
SDRAM 1	Frame Out 2	900900
SDRAM 1	Full Grayscale Image 2	349920
SDRAM 1	Smoothed Image 1	27216
SDRAM 1	Binary Imaged 1	27216

Table 4.2: L3 Sub Bank 0 and 1 Memory Placement

at the same time. All four banks are used to maximize the distance between different chunks of memory. They are also positioned so that the memory that cannot interact are close to each other. In Table 4.2, inbound frames 0 and 2 are in the same bank. Since inbound frames 1 and 3 are between those, they will never have memory conflicts between the two. Different memory placements affect the performance of the algorithm. Section 5.2.5 shows how different memory location setups affect algorithm run time.

L2 Memory

L2 is 128KB of low latency memory [5]. The optimized algorithm only utilizes 8.96KB out of the 128KB. The solution uses 57.6KB if frame size is not optimized. L2 is separated into eight 16KB sub-banks.

L2 is not used much in this algorithm. Table 4.4 shows how L2 is allocated and depending on the optimization enabled, L2's use increases or decreases. If the algorithm is optimized to only use 112 width search windows, there is plenty of room left in L2. It is not wasted space because the memory residing in L3 could not fit in L2. The binary image memory locations shown in table 4.4 are

Sub Bank	Purpose	Size in Bytes
SDRAM 2	Full Grayscale Image 1	349920
SDRAM 2	Grayscale Image 1	27216
SDRAM 2	Binary Image 2	27216
SDRAM 2	Consistent Background	699840
SDRAM 2	Frame In 1	900900
SDRAM 2	Frame In 3	900900
SDRAM 3	Frame Out 1	900900
SDRAM 3	History Frame 9	699840
SDRAM 3	History Frame 0	699840
SDRAM 3	History Frame 1	699840
SDRAM 3	History Frame 2	699840
SDRAM 3	History Frame 3	699840
SDRAM 3	History Frame 4	699840
SDRAM 3	History Frame 5	699840
SDRAM 3	History Frame 6	699840
SDRAM 3	History Frame 7	699840
SDRAM 3	History Frame 8	699840
SDRAM 2	Grayscale Image 1	27216
SDRAM 3	Frame Out 3	900900

Table 4.3: L3 Sub Bank 2 and 3 Memory Placement

in L2 because they are the most accessed part of memory in the algorithm. They are too large for L1 and small enough where converting from L3 to L1 would not be faster because it would require more transfers.

Data points for the results section are located in L2. This memory location requires 4000 bytes and is located in sub bank 3. This means that it will minimize conflicts with the rest of the algorithm [5].

L1 Memory

L1 memory is the fastest available memory on the Blackfin 561 processor [5]. Each core has its own 100KB of internal L1 memory. L1 is split up as follows:

Sub Bank	Purpose	Size in Bytes
SRAM 4	Binary Image 1 0	2240 to 14400
SRAM 5	Binary Image 2 0	2240 to 14400
SRAM 6	Binary Image 3 0	2240 to 14400
SRAM 7	Binary Image 4 0	2240 to 14400

Table 4.4: L2 Memory Allocation

- 32KB of instruction memory:
  - 16KB of instruction SRAM
  - 16KB of instruction cache or SRAM
- 64KB of data memory:
  - 32KB of data cache/SRAM
  - 32KB of SRAM
- 4KB of data scratch pad SRAM

The core that runs the stabilization algorithm, core A, contains all the rest of the code and variables in L1. Core A has instruction caching on and data cache off. Instruction caching is beneficial because the Compute Point code loops a lot per frame. Data cache is disabled because memory management is optimized manually as described in section 4.2.2.

Memory Movement

The Blackfin 561 has access to a specific type of Direct Memory Access (DMA) for Memory. Memory Direct Memory Access, or MDMA, is used to transfer blocks of memory from one location to another without using CPU time. This allows the processor to setup and schedule a memory transfer and continue executing

the algorithm. When the transfer is complete, the algorithm utilizes the memory in its new location.

MDMA is used for multiple reasons. This program uses MDMA for moving small chunks of memory from slow to fast memory, moving memory in the background, and for performing steps of the algorithm. Section 4.2.3 shows how transferring the next frame in the background improves the algorithms processing time. Section 4.2.4 describes where the algorithm utilizes MDMA to transfer chunks of memory from slow to fast access memory to improve runtime. MDMA helping with parts of the algorithm is explained in section 4.2.4.

4.2.3 Deinterlacing and Asynchronous Frame Capture

Deinterlacing

National Television Standards Committee (NTSC) is interlaced video. Interlaced video is captured at 60Hz and split into two fields [32]. This means that for every frame, the even and odd video lines are from different capture times [19]. Under normal conditions, this method looks natural on a television. Interlaced video causes two problems for stabilizing UAV video. The first problem is operators misinformation due to interlacing artifacts from vibration. For example, a license plate may be difficult to read with heavy vibration affecting the video. Even though stabilization is applied, the field offset may distort numbers and characters. The second problem is that the algorithm creating a progressive scanned image out of the two fields may affect algorithm performance.

Figure 4.5 shows what happens when a frame has interlacing artifacts. Interlacing artifacts ghost objects which makes the frame look like some objects

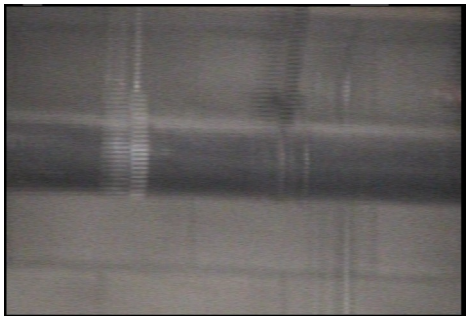

Figure 4.5: Interlaced Frame

are doubled. There are many different ways to deinterlace an image. It is not possible to fix interlacing issues in analog [19]. The format must be in digital form, like YCbCr or RGB to deinterlace [19]. Poynton [32] and Jack [19] describe a few methods for deinterlacing an image after it is decoded.

The first method they explain to remove deinterlacing is called line replication [32]. This method duplicates every odd line to create the even lines. This halves the vertical resolution [32]. It adds blockiness that makes the image look grainy [32]. The results of line replication are shown in figure 4.6. It requires the least amount of additional processing because it just replicates each line.

The interfield averaging method combats the blockiness of line replication [32]. This process takes the odd field and averages odd line to generate the even line between them [32]. This fixes the grainy image created by line replication, but does not fix the reduction in vertical resolution [32].

Other deinterlacing methods are described in Poynton's book [32]. These include using a low pass filter and multiple frames to present a clearer picture. These are not used in this solution because the algorithm would benefit the most by reducing runtime.

This solution uses line replication because it reduces processing time and

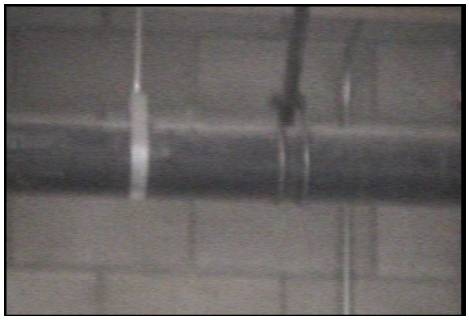

Figure 4.6: Deinterlaced Frame

improves the image substantially in video with vibration. Even though the image is more grainy, the reduction in ghosting helps the user see a clearer picture. The difference between figures 4.5 and 4.6 clearly shows an improvement. The deinterlaced frame is grainier, but there is no ghosting and the details of each support beam are clear. The algorithm speeds up substantially too because of deinterlacing. Those results are discussed in section 5.2.7.

The choice to deinterlace affects the flow of the algorithm. This is discussed in the next section.

Asynchronous Frame Capture

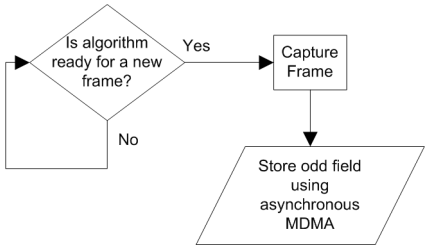

Figure 4.7: Asynchronous Frame Capture Part 1

The first step with the Blackfin optimized version of the stabilization algo-

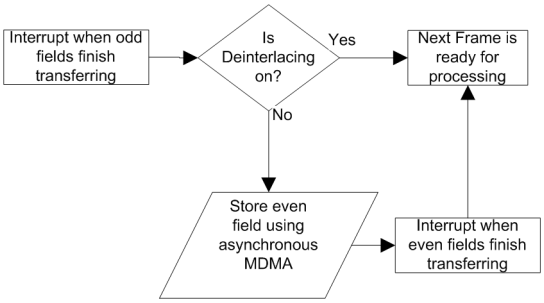

Figure 4.8: Asynchronous Frame Capture Part 2

rithm is to start the capture of the next frame. This section describes the "Start async new frame capture" box, or step 1, in figure 4.3. Figures 4.7 and 4.8 show what happens asynchronously to the main algorithm. Figure 4.7 shows what happens every time an input frame is captured by the decoder. The decoder fires an interrupt that checks to see if the algorithm is ready for a new frame. If a new frame is not needed, then the algorithm doesn't need to process that frame. This saves processor time. If the algorithm is ready for the next frame, an asynchronous Memory Direct Memory Access (MDMA) transfer is setup to move the input frame into a history frame. The asynchronous MDMA transfer allows the current frame to continue processing with the processor while the MDMA controller independently moves the memory without disturbing the stabilization algorithm. The odd field is transferred first.

Once the odd field is transferred, another interrupt fires. Figure 4.8 shows the second interrupt. This interrupt checks to see if deinterlacing is enabled or not. Deinterlacing is described in section 4.2.3. If deinterlacing is enabled, it does not transfer the even field to the history frame location and the frame is ready for processing when the current frame is done processing. If deinterlacing is not enabled, MDMA transfers the even field into the history frame and the another interrupt is set. Once the even field is transferred, the next frame is ready for

processing. Section 5.2.10 shows the results of this optimization. The Blackfin processor can setup MDMA to interrupt when one field is done being copied, so the algorithm can begin work without waiting for the second field. This buys valuable milliseconds that speed up stabilization.

4.2.4 Feature Selection

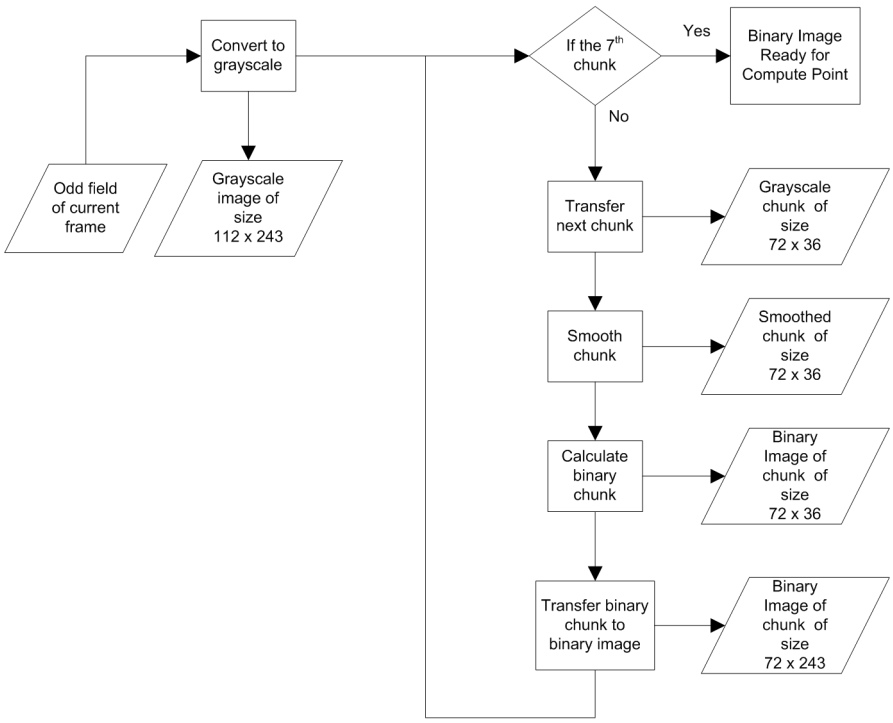

Figure 4.9: Generating the Binary Image

After the next frame capture process begins, the algorithm searches for new features in the current frame. Figure 4.9 described in the next sections. Blackfin specific optimizations are discussed throughout the sections.

Convert to Grayscale

Normally the entire frame is used to identify features [21]. For this embedded solution, algorithm quality must be sacrificed for speed. This algorithm searches only one field, the middle 112 pixels of the field, and looks for a smaller corner than what the original BCD algorithm asks for. These changes are different than the original algorithm described in section 4.1.

The first optimization and step that the algorithm performs is only using the middle part of the screen. The results for searching just the middle 112×243 pixels is in section 5.2.2. 112×243 is chosen because it provides over 15 FPS while still finding enough features to track. Memory Direct Memory Access (MDMA) is used to transfer from the full image to working area of 112×243.

The algorithm expects a grayscale image [34]. This is step number 2 in 4.3. The Blackfin has the advantage of highly configurable MDMA transfers [5]. The code in listing 5.2 shows how to setup a MDMA transfer to copy just the luma values from the UYVY image. The results of this optimization are in section 5.2.3. This process is only done once per frame. A grayscale image example is shown in figure 5.1.

Binary Corner Detection

Binary corner detector [34], described in section 4.1.1, is used in this algorithm. This is step number 3 in 4.3. Binary corner detection is used because it finds strong features and noise reduction algorithms are not needed [21]. It is also faster to compute [34]. Since this algorithm's main focus is on speed, the runtime is very important.

The first step in the algorithm requires that the slow memory moves into fast L1 memory. The algorithm takes chunks 72×36 from the grayscale image in L3 and moves it into L1. This allows the algorithm to calculate the binary image from quick to access memory. The algorithm goes through 7 chunks, which means it covers all 243 rows of pixels with overlap. Depending on how many columns of pixels the algorithm is optimized to handle, it will execute between just 1 chunk horizontally up to 20. For the optimized algorithm, 1 chunk is read per row to cover 72 of the 112 pixels of width.

One method to move this data is to use `memcpy`. An optimized method is to use MDMA to transfer the data. The results of this optimization are discussed in section 5.2.4.

Smoothing the Image

The first step of Binary Corner Detection is to smooth the image [35]. This helps reduce image noise [32]. A Gaussian smoothing is applied with a σ value of 0.8 [35]. This value is strategic because it allows a filter to be approximated by $[0.25, 0.5, 1.0, 0.5, 0.25]$ [35]. The Gaussian smoothing filter with σ of 0.8 is shown below :

$$Smoothed(i,j) = \frac{\frac{I_{i,j-2}}{4} + \frac{I_{i,j-1}}{2} + I_{i,j} + \frac{I_{i,j+1}}{2} + \frac{I_{i,j+2}}{4}}{2} \tag{4.4}$$

In the figure above, $I_{i,j}$ stands for the image intensity at row i, column j. As noted above, the σ value of 0.8 creates a filter that uses divisions that are multiples of two. Embedded processors work well with divisions that are in multiples of two because logical shifts are available to speed up execution. Divisions are the most expensive command on the Blackfin processor [5] while shifts are the least

expensive command on the BlackFin processor [5].

This means that the Blackfin can substitute the divides for the following equation:

$$\begin{aligned} sum &= 0 \\ sum &= sum + (I_{i,j-2} \gg 2) \\ sum &= sum + (I_{i,j-1} \gg 1) \\ sum &= sum + I_{i,j} \\ sum &= sum + (I_{i,j+1} \gg 1) \\ sum &= sum + (I_{i,j+2} \gg 2) \\ sum &= sum \gg 1 \\ Smoothed(i,j) &= sum \end{aligned} \tag{4.5}$$

This equation calculates the same result as above but in an Blackfin optimized way. A smoothed image example is shown in figure 5.2.

Binary Image Calculation

After smoothing the image, the image is ready to create a binary image. This creates distinct pixels that show where corners are. The first step to making a binary image is to generate a Laplacian image [34]. Horn [17] approximates the Laplacian with the following equation:

$$Laplacian_{i,j} = I_{i-1,j} + I_{i,j-1} + I_{i+1,j} + I_{i,j+1} - (4 \cdot I_{i,j}) \tag{4.6}$$

$I_{i,j}$ is the intensity value at each i row and j column in the smoothed image.

The Blackfin optimized version splits the image up into multiple pointers. This aids the Blackfin processor by doing less math to determine what memory

location to access. The memory will increment in steps of 1, which means that one addition is much quicker than multiplications and additions [5]. The inflection points are noted by setting the pixel's intensity value to white. If it is not an inflection point then set the pixel's intensity to black. After this algorithm is complete, the entire frame will be just black and white pixels. The white pixels showcase the edges and the black pixels showcase the non-edges. After each chunk is processed, it is then put back into L3 to create a full binary image, piece by piece. A binary image example is shown in figure 5.3.

Compute Point

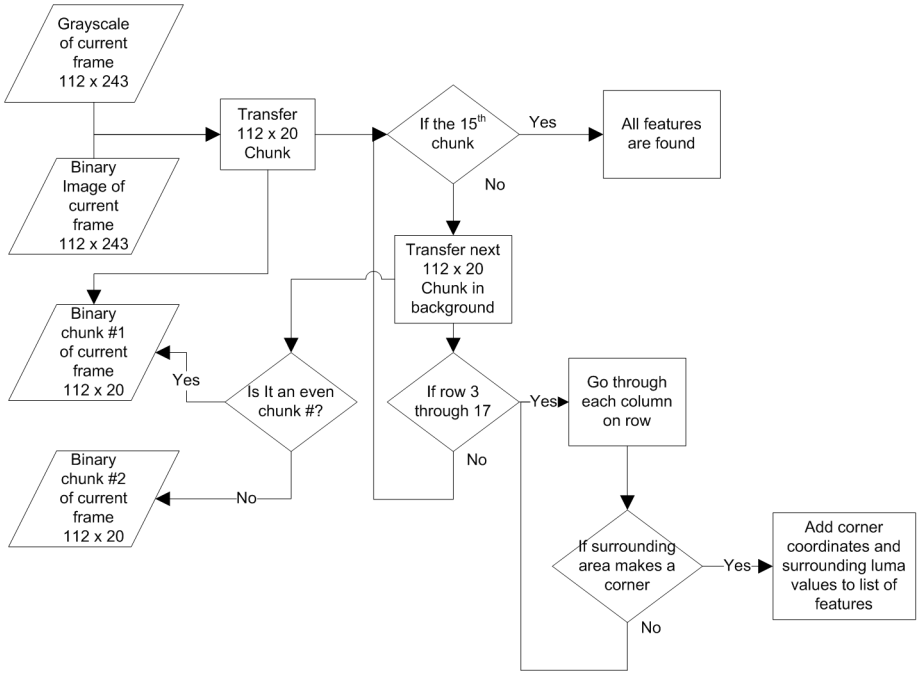

Figure 4.10: Using the Binary Image to Find Corners

Compute Point is the process of deciding if a certain area of pixels is a corner or not. Figure 4.10 shows the algorithm for compute point. Like the previous sections, the binary image that is stored in L3 is moved into L2 to allow the

algorithm to operate in quick memory. This part of the algorithm uses MDMA to transfer data in the background while the processor executes upon a previous chunk. This means the algorithm only waits for the first chunk to transfer into L2 memory. The algorithm then works in L2 memory without any of the disadvantages of memory transfer speeds.

The Blackfin utilizes multiple pointers to reduce the number of indirect memory accesses. This means that instead of indexing to other spots of the potential corner with multiplications and additions, there are only simple additions at the expense of using more registers. With the Blackfin, this is not a problem since there are eight 32 bit data registers [5].

Small Corner Detection

The original binary corner detector requires a mask of 37 neighboring pixels to determine if it is a corner or not. This is made for use on a full 720×486 video frame. Since the proposed algorithm only uses 112×243, each corner's height is halved. This means that a mask will be comprised of 17 neighboring pixels to determine if a spot is a corner or not. This reduces the computation time for the algorithm because the number of comparisons are halved.

The number of white spots in the corner and how they are arranged is then computed using those 17 neighboring pixels. If the white pixels indicate that there is a corner, then the spot is recorded for further analysis.

The algorithm stores each corner's information in a list. While this list is not full, any corner is accepted. Once the list is full, corners with a higher difference in intensity between the center and corner are added to the list. This was added to overcome the issue that Binary Corner Detection does not have a value system

which can show what is a better corner [34].

After all of the features are collected, this algorithm uses the minimum separation technique mentioned in section 4.1.1. The solution on the Blackfin has a minimum corner distance of 14. This was chosen to make sure that the entire field is utilized. All the features are looked at and compared to make sure none are too close. Ones that are too close are culled.

Branch Prediction

Although the branch prediction optimization does not work in all solutions [13], this algorithm uses branch prediction in compute point because the expected frequency of some branches are known. As seen in results section 5.2.9, there is a branch in compute point which is false one fourth of the time.

The Blackfin processor allows the programmer to specify if a branch should be predicted to be true or false. This allows the pipeline to proceed with instructions that more often than not will execute. On a full 720×243 frame, some branches are checked as often as 8,900 times a frame. This can make an enormous difference because on average it checks that branch 2,000 times. The results for a branch prediction optimization is in section 5.2.9. The results in that section use a 112×243 frame which on average checks that branch 84 times, but the maximum is 947 times per frame.

The other two reasons to use branch prediction are if there is an uneven amount of code on one side of the branch and if the logic contains a lot of sub branches [13]. These are both true for the case in compute point. The expected false path is a direct return while the true block is another conditional statement.

Loop Unrolling

Loop unrolling is one of the oldest forms of optimizations [13]. Compute point benefits from loop unrolling. The results are in section 5.2.8. Unrolling decreases execution time by reducing the number of conditional branches [13]. This is not the primary reason for unrolling this part of compute point. The main reason for unrolling is navigate the corner point calculating part of memory without doing multiplications. Unrolling allows the algorithm to compare the potential corner to the "mask" [34] with simple additions and subtractions to locations in memory.

Loop unrolling has some disadvantages. It increases code space required [13]. This is not an issue because the Blackfin 561 contains 32KB of space for code. For this optimization, the original unrolled compute point is 8 lines of code. The unrolled version is 150 lines of code. That is 17 times larger than the original version.

Information Storage

Information gathered from feature selection must be stored. A list of unmatched features is retained for the next frame. The grayscale version of the last frame is also retained. The next frame uses this grayscale image to find matches. This is described in section 4.2.5.

4.2.5 Feature Matching

Feature matching is required to stabilize video because the algorithm utilizes motion vectors. This is step number 5 in 4.3. Without pairing features, the

algorithm will not know where to move the screen and the algorithm will fail. The process used in this solution to match features looks through the list of unmatched features and then compares surrounding areas with potential new locations. It uses the grayscale image saved in section 4.2.4.

The sum of absolute differences (SAD) is used to compare sections of 16×16 blocks of memory. The result outputs the raw difference between the two chunks of memory.

The first step is to clear the list of paired features. Then a search window around the original feature's `x` and `y` coordinates is iterated through. This search window is a 20×20 square. At each spot a comparison is done to see if that new pixel location is the best match.

The original algorithm uses a pyramidal template matching algorithm. The modified Blackfin algorithm uses the profile matching instead of a pyramidal template matching. This is because the Blackfin has an optimized Sum of Absolute Differences (SAD) function. This is chosen because the Blackfin is a fixed point processor [5] and it has built in functions for doing multiple byte comparisons in one clock instruction. The Blackfin manual has an example sum of absolute differences function [7]. The Blackfin was made to do video processing because of these builtin functions. Being able to do multiple compares per clock instruction is a huge advantage over conventional desktop processors and floating point embedded processors. The next section describes the different SAD implementations.

Sum Of Absolute Differences

Sum of Absolute Differences (SAD) is the process of computing the difference between two values. This algorithm uses it to compare two different blocks of memory while looking for where a feature moved between frames. If the difference between the blocks is low, then the new location is found.

The algorithm the solution uses compares two 16×16 blocks of memory. Three different implementations of this SAD function are tested. The next sections describe the straightforward double nested loop approach, the solution utilizing Blackfin built in macros to improve performance, and one that uses knowledge of the memory architecture to optimize execution. Each algorithm outputs exactly the same values given the same input blocks. They only differ in execution time.

Reference Algorithm

The reference algorithm for SAD is shown in listing 6. This function is referenced in the Blackfin Compiler and Library Manual [6]. This function iterates through each pixel in the first 16×16 block and takes the absolute value of the difference between that same location's intensity value on the second 16×16 block. The value is summed up and returned at the end.

The function is below:

$SAD = \sum_{i=1}^{16} \sum_{j=1}^{16} |a(i,j) - b(i,j)|$

Both a and b are 16×16 blocks of intensity values. A is the potential new feature location and b is the reference feature block.

This function is simple to write and read, but the Blackfin's instruction language allows for improvements. The next section shows that improvement.

With Builtin Macros

The Blackfin programming manual contains a SAD function that is optimized for the Blackfin [6]. This function was not always an optimized representation of the reference SAD function. This function originally had errors in it. See code in listing 5. It was completely broken. It originally did not use the image width. It also only compared the top row. The function was optimized and is now a valid optimization. Listing 7 contains the source code for the SAD function with built in macro support. Like the reference function described in listing 6, the optimized function takes two 16×16 blocks of memory and compares them by summing up the absolute value of the differences between each corresponding pixel.

The Subtract Absolute Accumulate (`SAA`) macros improve the performance of the SAD function. `SAA` is split up into two macros: `SAA()` and `SAAR()`. `SAA` is the first step. Theses macros take two 8 byte chunks of memory and subtract the difference between each byte and store the result of each byte's difference in its corresponding byte position. It "adds the upper half words and the two lower half words of each accumulator and places each result in a 32 bit data register [23]." Table 4.5 shows how this macro works [7].

src_reg_0	$a(i, j+3)$		$a(i, j+2)$		$a(i, j+1)$		$a(i, j)$
src_reg_1	$b(i, j+3)$		$b(i, j+2)$		$b(i, j+1)$		$b(i, j)$
A1.H	$+=$	A1.L	$+=$	A0.H	$+=$	A0.L	$+=$
	$\lvert a(i, j+3)$		$\lvert a(i, j+2)$		$\lvert a(i, j+1)$		$\lvert a(i, j)$
	$-b(i, j+3)\rvert$		$-b(i, j+2)\rvert$		$-b(i, j+1)\rvert$		$-b(i, j)\rvert$

Table 4.5: `SAA` Macro

This means that it calculates the absolute difference between 4 of the 8 bytes (`R0` and `R2`) and stores that into an accumulator register. Those same registers are then filled with the next 4 bytes that are not already loaded into `R1` and `R3`.

The `SAAR()` macro must execute next.

The `SAAR()` macro reverses the source registers [7]. This is done to "reduce the overhead of reloading both register pair operands to maintain byte order for each calculation [7]." This means that the second instruction reverses the byte order so that `R1` and `R3` are used. The difference is then added to the sum held in the accumulator. The combination of `SAA` and `SAAR` essentially creates a leapfrogging approach to calculating the differences between two blocks. One cycle is wasted at the beginning loading all four registers, but after that the cycles are halved because only 2 registers are loaded each time. This means that `SAA` and `SAAR` must alternate calls to optimize speed [23].

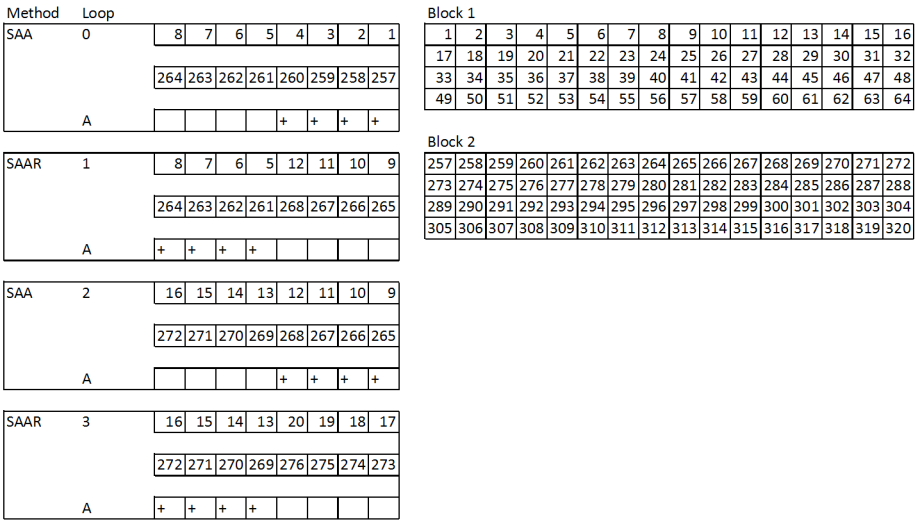

Figure 4.11: Leapfrogging `SAA` and `SAAR` Calls

Figure 4.11 shows the leapfrogging between `SAA` and `SAAR`. On the right there are two blocks that have each pixel labeled. On the left there are four example iterations of going through comparing the pixels. Loop 0 uses subtract absolute accumulate (`SAA`) on the lower four bytes and adds it to the sum in the accumulator. After the `SAA` call, the lower bytes are replaced with the next four bytes.

Loop 1 then calls `SAA`, but reverses the input registers (`SAAR`). This means that it will use the higher four bytes and calculates and adds to the higher bytes in the accumulator. Then the next four bytes are loaded into the higher bytes. In this example, this process is repeated until all four rows are compared. In the solution that is 276 pixels (16×16).

Bulitin Macros and Aligned Memory

The function in listing 7 produces a lot of unnecessary assembly. Changing a few lines to what is shown in listing 8 optimizes the built in macro SAD function. There are wasted instructions aligning memory. A misaligned access exception occurs when the processor executes a nonaligned memory operation [7]. Alignment exceptions may be disabled by using the `DISALGNEXCPT` instruction [7]. `SAA` instructions automatically disable alignment exceptions [7].

This means that the function does not need to explicitly align the memory because the `SAA` function suppresses the alignment exceptions and the 16×16 blocks are aligned in 4 byte boundaries anyway.

The Sum of Absolute Differences optimization's results are shown in section 5.2.6.

4.2.6 Motion Estimation

The next step is to take the paired features and estimate the frame to frame motion. This is step number 5 in 4.3. This motion describes the difference between the last frame and the current frame. Section 4.1.3 describes a few different methods. The frame to frame estimation used to be calculated by an affine model, but due to the increased horsepower requirements of rotation and

scaling, a simple summation is used now:

$$
\begin{aligned}
N &= NumberofPairedFeatures \\
Motionx &= \frac{\sum_{i=1}^{N} Lastx_i - Currentx_i}{N} && (4.7) \\
Motiony &= \frac{\sum_{i=1}^{N} Lasty_i - Currenty_i}{N} && (4.8)
\end{aligned}
$$

The translation model calculates by summing the differences between the paired frames. This is shown in equations 4.7 and 4.8. These sum the difference between the last and current locations for both `x` and `y`. The average is stored into an array for use by the Parabolic Fit Camera defined in section 4.2.7.

4.2.7 Intended Motion Calculation

The next step is to estimate the intended motion of the video. This is step number 6 in 4.3. The Parabolic Fit Camera (PFC) estimates the intended motion for the frame at the end of the "before" list. The algorithm displays that frame next. As described in section 4.1.4, the array of movement vectors are the input into the PFC algorithm. This solution uses 4 "before" and 31 "after" frames for PFC. The algorithm uses 4 "before" frames because it is quick enough to handle quick changes in direction, but does not overcompensate. The algorithm uses 31 "after" frames because it allows the history to influence the future but not completely control it.

The Blackfin algorithm first finds the last "before" frame's `x` and `y` information. The Blackfin adapted version uses a scale of 1 and rotation of 0 because it uses the translational model to estimate motion.

There are built in methods for matrix transposing, multiplication, inverting,

addition and subtraction [6]. These are hand optimized by Analog Devices. The solution uses these to compute the least squares solution that solves PFC. This is not as fast as the fixed point version, but the compiler does a good job at making the matrix multiplication quick by avoiding library calls where possible and utilizing parallel instructions [6].

The output of PFC gives an intended motion estimation. This is the spot where the algorithm thinks the video meant to be. Section 5.1.2 shows how well 4 "before" and 31 "after" frames predicts the intended motion and how it compares to the measured unintended motion. That section also compares different combinations of "before" and "after" numbers. Section 4.2.8 discusses how the algorithm uses the output of PFC to stabilize the image.

4.2.8 Offset And Display The Image

The last steps are to take the output of the Parabolic Fit Camera (PFC) from section 4.2.7 and translate the last "before" frame by that offset. This is step numbers 7 and 8 in 4.3. This is done by taking the difference between the measured offset and the output of the PFC. These offsets are shown in figure 5.11.

The image is transferred to a common frame before it is sent to an output frame. This is done because each frame is layered on top of the previous frames. This creates a blur effect on the borders of the video frame instead of a black border. Since there are four output buffers, a common layering frame is used to prevent the background layers jumping around. If the memory is directly transferred into each output buffer, the border is distracting. This feature could be removed to improve speed at the cost of aesthetics.

There are four output buffers for this algorithm. Every time the outbound Parallel Peripheral Interface (PPI) interrupts, the interrupt checks which buffer to output. It outputs the same frame until a frame is processed and put into the next outbound buffer. The algorithm then outputs that frame.

Chapter 5

Results

5.1 Stabilization Results

5.1.1 Binary Corner Detection

Section 4.2.4 describes how the algorithm must strip the image of its color values and transfer only the luma values to another location in L3. Figure 5.1 shows the odd field of a frame after its conversion to grayscale. Notice how the vertical resolution is halved. This is due to the image being only one field and has a resolution of 720×243.

After the image is only luma values, the algorithm smooths the image out to reduce noise and highlight strong corners. Section 4.2.4 describes this step. Figure 5.2 shows the smoothed version of figure 5.1. Notice how it smooths out the non-unique corners in the bricks on the wall. This is advantageous because corners on bricks can create false positives due to there being so many similar corners within the search area.

Figure 5.1: Grayscale image of one field

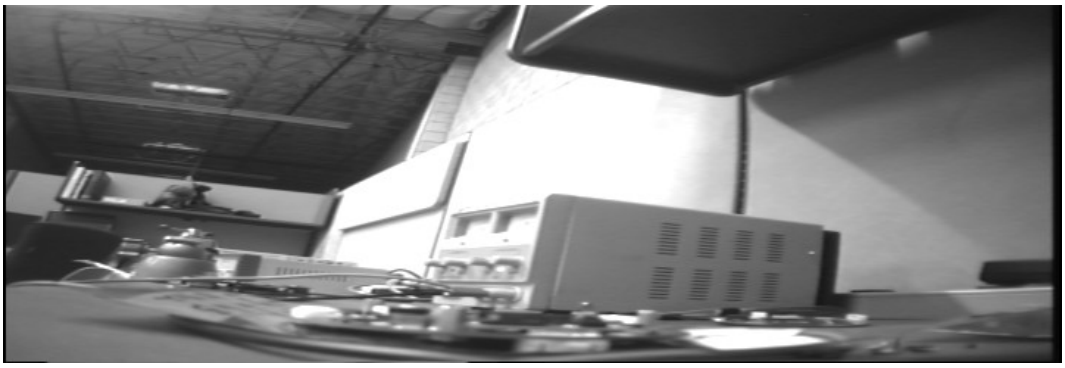

Figure 5.2: Smoothed image of one field

The image is then converted to a binary image like described in section 4.2.4. Figure 5.3 shows what the binary image of figure 5.2 looks like. It completely rejects any corners on the bright portion of the cubicle wall. There are lots of false positive parts in the cubicle walls but those will not pass the binary corner detection process. There are strong corners around the power supply.

5.1.2 Parabolic Fit Camera

The stabilization algorithm measures unintended motion and calculates intended motion. This process is described in section 4.2.7. This solution uses the

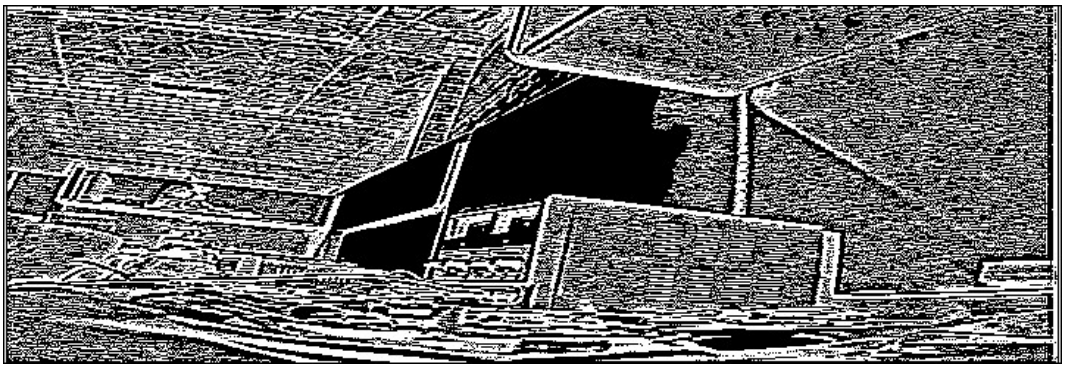

Figure 5.3: Binary image of one field

Parabolic Fit Camera (PFC) approach to estimate intended motion. PFC uses a certain amount of "before" and "after" frames to filter the unintended motion. "Before" frames are frames that the algorithm captured, but have not been displayed yet. "After" frames are frames that have already been displayed. There can be between 1 and 8 "before" frames and between 1 and 31 "after" frames. The following section describes how different number combinations of "before" and "after" frames affects the stabilization output. The following section defines the measured frame to frame movement as "unintended motion" and the output of PFC as "intended motion."

The number of "before" frames changes how quickly the algorithm responds to unintended motion. The number of "after" changes how much the past influences the future. Figure 5.4 shows what happens when the filter only uses the minimum criteria.

Figure 5.4 shows that the filter does not do anything when there are only two frames worth of information. This is not useful for stabilizing the video.

Figure 5.5 shows what happens when only the "before" frames are used. It filters out high frequency noise but does not filter out the jump in movement

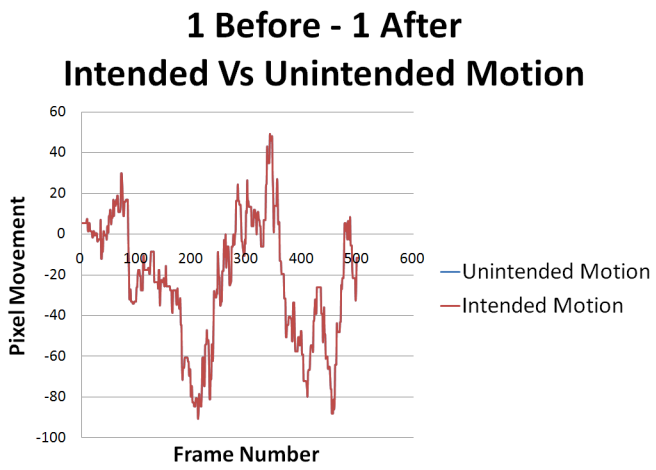

Figure 5.4: Vertical Vibration - Intended Vs Unintended Motion. 1 Before Frames, 1 After Frames

between frame 312 and 320.

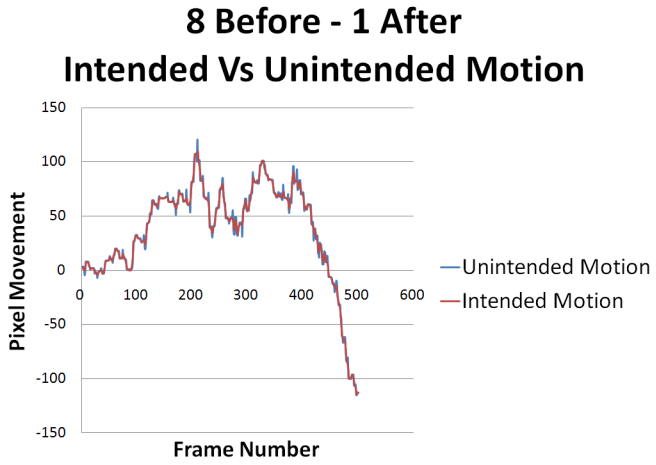

Figure 5.5: Vertical Vibration - Intended Vs Unintended Motion. 8 Before Frames, 1 After Frames

This solution uses 4 "before" and 31 "after" frames. Figure 5.6 shows how PFC estimates intended vs unintended motion on the x axis with the algorithms. The horizontal vibration is handled well with the intended motion following trends and avoiding high frequency changes. Frames 25 to 50 show how well

the algorithm will cut out high frequency noise. Frames between 200 and 220 show how a relatively slow, dramatic change in direction affects the algorithm.

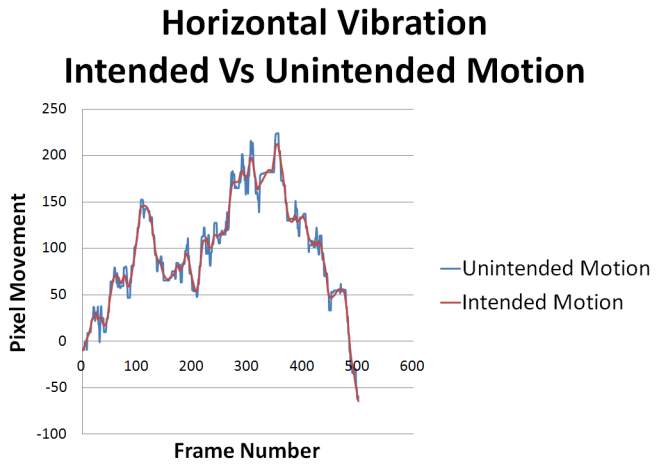

Figure 5.6: Horizontal Vibration - Intended Vs Unintended Motion. 4 Before Frames, 31 After Frames

Figure 5.7 shows how the same parameters affect vertical vibration. It successfully filters out high frequency noise while following trend lines to provide a smooth video output.

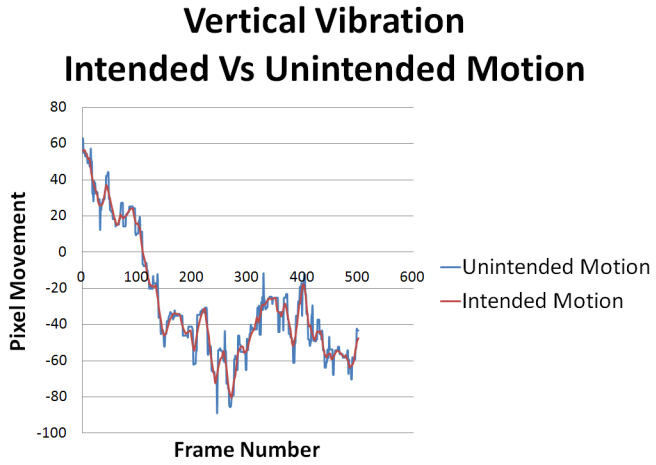

Figure 5.7: Vertical Vibration - Intended Vs Unintended Motion. 4 Before Frames, 31 After Frames

4 “before” and 31 “after” are used for a reason. A too low “before” number causes the intended motion calculation to overshoot the unintended motion curve. This is seen in figure 5.8. This means that the algorithm is just responding to the newest frames and not using the “future” information well. This smaller “before” number shrinks the delay between a captured frame and display by 200ms at 15 FPS.

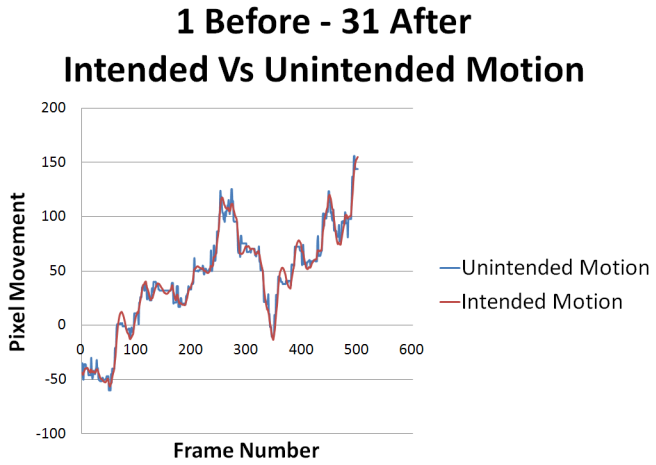

Figure 5.8: Vertical Vibration - Intended Vs Unintended Motion. 1 Before Frames, 31 After Frames

A high “before” number causes the algorithm to predict the future frames extremely well because it has access to more frames that are buffered. A high “before” number stabilizes the video better than a smaller number, but also increases the delay in the video. Figure 5.9 shows how changing the “before” parameter to its maximum affects the intended motion calculation. The intended motion line follows the unintended motion when the unintended motion stays at a local maximum or minimum. It still filters large changes and removes the bursts. The four additional frames adds a fourth of a second additional latency at 15 FPS.

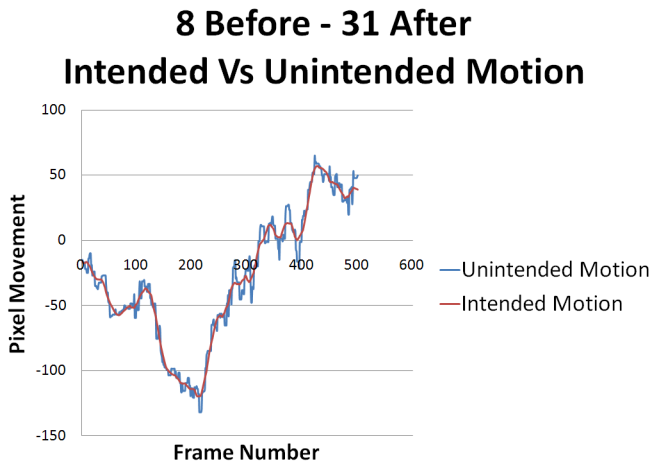

Figure 5.9: Vertical Vibration - Intended Vs Unintended Motion. 8 Before Frames, 31 After Frames

A too low "after" number causes the algorithm to "hug" the unintended motion and provides very little stabilization. Figure 5.10 shows the intended motion line following the unintended motion. It does help in some situations. Around frames 290, 330, and 480 the algorithm successfully filters out high frequency noise.

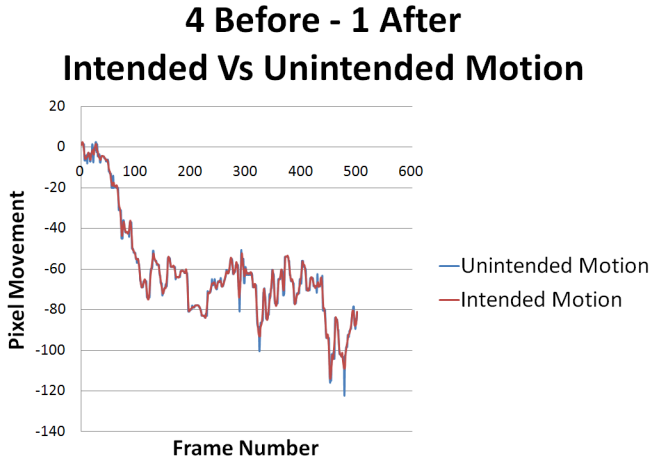

Figure 5.10: Vertical Vibration - Intended Vs Unintended Motion. 4 Before Frames, 1 After Frames

A too high "after" number does not change the results much. 31 is chosen based on the graph in [21] for a good match to 4 "before". This graph is used to find the "after" value that "minimizes the energy in the specified frequency range [21]." This is to bring structural stability to a continuous function.

5.1.3 Stabilization Offsets

The outputs of the Parabolic Fit Camera (PFC) are `x` and `y` offsets. Figure 5.11 shows how the `y` offset looks compared to the calculated intended and unintended motion. The offsets are erratic because the unintended motion is moving at such a high frequency. The offsets are cyclic because this is measuring movement induced by vibration.

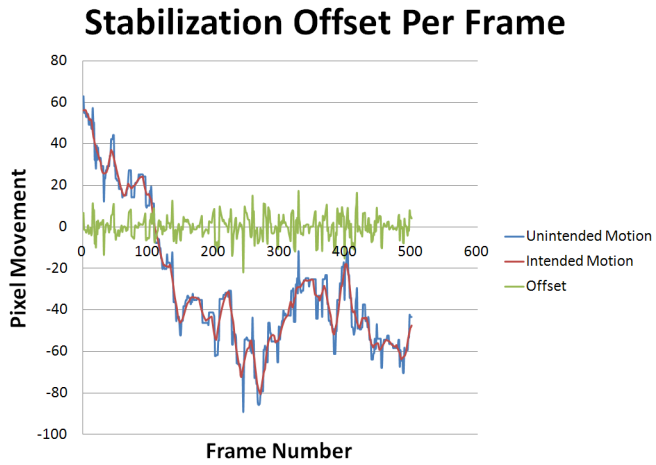

Figure 5.11: Stabilization Offset Per Frame

5.2 Optimization Results

5.2.1 Source Videos

Three videos were used for the results section. Two videos contain the same video content. These videos show a pipe mounted on a wall. Each is comprised of two sections. The first nine seconds show a blurry sign for test identification purposes. The second section is sixty seconds of 40Hz vibration video. The video loops until the algorithm processes 1000 frames. The difference between the two videos is one is more blurry to force the algorithm to not find features. The second video is clear enough where features will be detected. If the source video was the blurry version, the 10 seconds of blurry video does affect the results and that can be seen in the plots of data. This is because of the increased number of features found during this time.

The third video is a solid blue screen. The blue screen guarantees that the algorithm does not find any features.

Most of the results used the version that did not find features or the blue screen. This showcases the change in the algorithms during optimizations without worrying about a difference in feature count. Any optimization that required features used the second video.

5.2.2 Searching Middle Pixels

The feature selection part of the algorithm requires the most computation time. The original algorithm searches one 720×243 pixel field. If the whole field is scanned, it will analyze 174,960 pixels. Each pixel requires a binary corner scan that does 17 comparisons. Since the algorithm only scans a 112×243 pixels

area, it only analyzes 27,216 pixels. Not scanning 147,744 pixels saves 2,511,648 comparisons compared to the non-embedded version.

Table 5.1 shows the different FPS results from different area scans. As expected, only searching 72×243 pixels increases the algorithm speed by 348%. The algorithm uses a section sized 112×243 to achieve 20 FPS. This is higher than the 15 FPS goal of this algorithm.

Size	Average FPS
72×243	24.76
112×243	20.21
200×243	14.27
320×243	12.49
424×243	10.4
520×243	8.95
624×243	7.79
720×243	7.12

Table 5.1: Frames Per Second of Algorithm With Different Search Sizes

5.2.3 Grayscale Conversion

Since the algorithm does not use color and only uses intensity, the image's color is removed. The format of the input video is UVYV. This means that every pair of two pixels has a unique intensity, but shares a color between them. An image with just intensity values is a grayscale image. Typically this would be done with a double for loop. Listing 5.1 shows this.

```
unsigned char* sPtr = historyFrames[curHistoryNum] + 1;
// Intensity
unsigned char* dPtr = tNewFrame;

// For one field of video
for(int rowNdx = 0; rowNdx < 243; rowNdx++)
{
   // For each pixel in the input frame
```

```
    for(int colNdx = 0; colNdx < 720; colNdx++)
    {
11      *dPtr = *sPtr;
        dPtr += 1;        // Fill each dest pixel
        sPtr += 2;        // Skip color pixels
    }
}
```

Listing 5.1: Manually Copying Pixels

The code in listing 5.1 copies just the intensity values into a new 2D array. With the Blackfin architecture, this computation can be reduced by using MDMA. MDMA allows memory movement without utilizing the CPU. The Blackfin also has support for 2D MDMA transfers. Listing 5.2 sets up and commands a blocking call that waits for the 2D MDMA transfer to complete.

```
// Start address at intensity value
srcTran.StartAddress = historyFrames[curHistoryNum] + 1;
srcTran.XCount = 1440 / 2 ;    // Want intensity only
srcTran.XModify = 2;           // Skip every other pixel
5 srcTran.YCount = 243;        // Only grayscale 1 field
srcTran.YModify = 2;   // Skip 1 pixel after row change

destTran.StartAddress = tNewFrame;
destTran.XCount = 1440 / 2;
10 destTran.XModify = 1;
destTran.YCount = 243;
destTran.YModify = 1;

// Copys 1 byte at a time using MDMA channel 2
15 myResult = adi_dma_MemoryCopy2D(StreamHandle2,
            &destTran, &srcTran, 1, NULL);
```

Listing 5.2: Copying Pixels via MDMA

The code in listing 5.2 involves two structures. One is for the source and one is for the destination. These are used to describe how the MDMA transfer will operate. The StartAddress holds the starting address. The XCount is the number of transfers to do for each YCount. The X axis is comparable to width and the Y axis is comparable to height. The X and Y Modify contains how many transfers to skip after a single count on the XCount. On the last XCount the

XModify is not applied and the YCount is applied. The source and destination structures can have different values for XCount and YCount, but their product must be the same between the two. This is required because every source transfer needs a destination. The last part of the setup is the size of the transfers. Since this algorithm requires only a byte at a time transfer, the MDMA will transfer at 1 byte at a time.

To describe the setup above, the source will start 1 pixel into the field. This is done because in UYVY, the first pixel is a color value and the second will be the first intensity value. The destination starts at the beginning of the array because it will be completely filled with intensity values. The XCount and YCount on the source is equal to 1 UYVY color field. That is 720×243 pixels, but because of color it will be 1440×243. Since the destination is half the size, we must half the amount of pixels. This number is normally also divided by the transfer size, but in this case it is only 1 byte at a time so it doesn't matter. The destination X and YCount are the same. The XModify and YModify will be 2 in the source because it will skip every other pixel while transferring. Since it contains no blanking information, it does not need to skip any additional pixels per line. The XModify and YModify on the destination is only 1 because it will fill every value. In this case, since the algorithm does not have anything else to do while waiting for a grayscale image, it will block and wait for it to finish.

Name	**MDMA On**	**MDMA Off**
Average	9.32	25.06
Min	3.93	14.91
Median	10.80	30.24
Max	12.06	32.02

Table 5.2: Converting a Field from Color to Grayscale in Millions of Cycles

Table 5.2 shows how much the algorithm is sped up by using MDMA. Even by transferring only 1 byte at a time, it on average doubles the speed. This is a speed increase of .023 seconds per frame processing. The max run time with MDMA on is faster than the fastest time without MDMA on. The huge difference in min and max is going to depend mostly on memory placement. This algorithm has 10 different locations to use as a source and 2 different destinations. There are also 4 memory banks. Future investigation could be used to identify which memory transfer is the slowest and eliminate that if possible which could potentially speed the algorithm up by another factor, making it 3.79 times faster than using a double nested for loop.

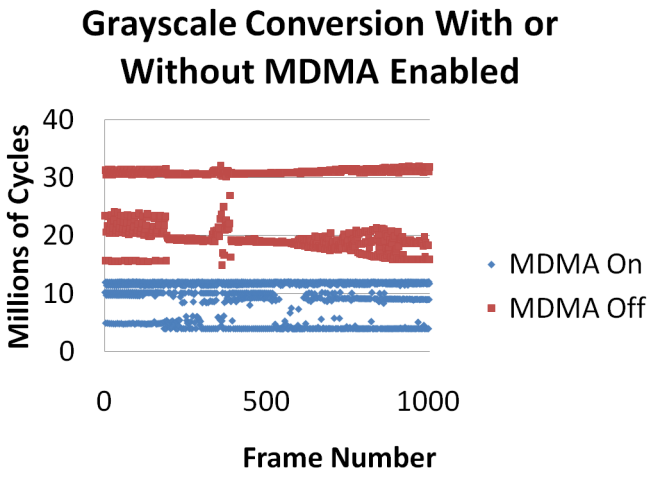

Figure 5.12: Grayscale Conversion with or without MDMA Enabled

Figure 5.12 shows two to three trend lines per method. The speed varies drastically between frames. Memory conflicts during memory transfer is most likely the cause of the difference between each frame in a cyclic pattern. As seen in the statistics, all data points for the transfer without MDMA are greater than all of the data points for transferring with MDMA enabled. This shows that even with memory conflicts this optimization always increases performance.

5.2.4 Memory Movement

Memory access is often the bottleneck to an embedded program [38]. This solution spends 80% of its time transferring memory. Each algorithm takes much-more time to execute when memory is in slow L3 memory. Section 4.2.2 describes the speed and how L3 memory interacts with executing code.

The most common memory movement will be moving data from slow memory, L3, to fast memory, L1. This allows the code to execute much quicker. After the memory is modified in L1, it will typically transfer back into L3 to free up L1 space.

MDMA instead of `memcpy`

One optimization is to use MDMA instead of `memcpy`. MDMA will move up to four bytes at a time. The algorithm spends time to find out if the source is on a four byte or two byte boundary. This is to see if a transfer size of four or two is allowed. If neither is, it will transfer one byte at a time.

Section 4.2.2 shows how to setup a MDMA transfer. The algorithm uses this optimization when it is executing Binary Corner Detection.

The results between the two algorithms on 1000 frames is shown in table 5.3.

Name	MDMA On	MDMA Off
Average	1.56	4.71
Min	1.44	3.53
Median	1.47	4.08
Max	1.85	7.75

Table 5.3: Comparing `Memcpy` vs MDMA L3 to L1 transfers in Millions of Cycles

Table 5.3 shows that there is a three times increase in speed on average. This is not the maximum improvement of four times, but that is due to memory alignment not allowing all of the transfers to be four bytes at a time. Even with the modulo calls, the algorithm is significantly faster than `memcpy`. The spread between the minimum and maximum for MDMA is small. The spread between the minimum and maximum for `memcpy` is large.

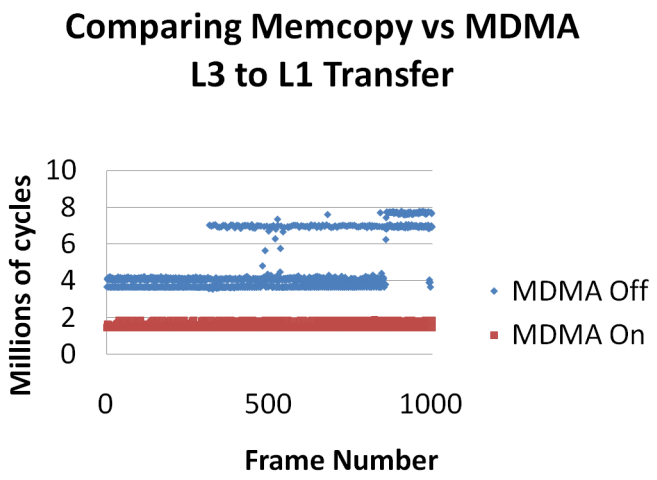

Figure 5.13: L3 to L1 conversion with or without MDMA Enabled

Figure 5.13 shows how much of an improvement MDMA is. All of the data points for using MDMA are below the `memcpy` data points. Figure 5.13 also shows a few trend lines in the `memcpy` data points. After 800 frames the `memcpy` algorithm doubled the required cycle counts to transfer the data. The reason for this is unknown. Interrupts containing a lot of instructions could cause this. This requires further investigation and is discussed in section 6.2.

5.2.5 Memory Placement

This test shows how memory placement affects algorithm run time. The test compares having all L3 memory in two SDRAM locations, randomly distributing the memory, and strategically placed throughout the 4 different segments. This slows down data transfer and access because when DMA and the processor access the same sub bank in the same cycle one of the accesses must stall [7]. If all of the memory locations are in the same bank, more stalls will occur. It is described in section 4.2.2. Table 5.4 shows what happens to the run time of the algorithm using different memory configurations.

Name	Optimized	Random	Worst Case
Average	30.43	51.23	80.05
Min	22.16	30.96	71.62
Median	30.19	46.84	80.08
Max	43.25	78.00	82.74

Table 5.4: Optimizing memory placement

Table 5.4 shows that optimizing memory placement greatly decreases overall algorithm run time. On average, it speeds up the algorithm by 2.6 times more than the worst case. Figure 5.14 shows the trend lines for each memory placement method. The ceiling is well defined by not utilizing all four banks and just placing the memory in two banks. This creates the worst case situation for memory placement. Random gives values in between the optimized and worst case. The trend lines within the random data points are discussed in the next section.

Random Memory

Figure 5.14 has interesting trend lines in the random data set. It follows a pattern that is seen in figure 5.15. The random data location causes different

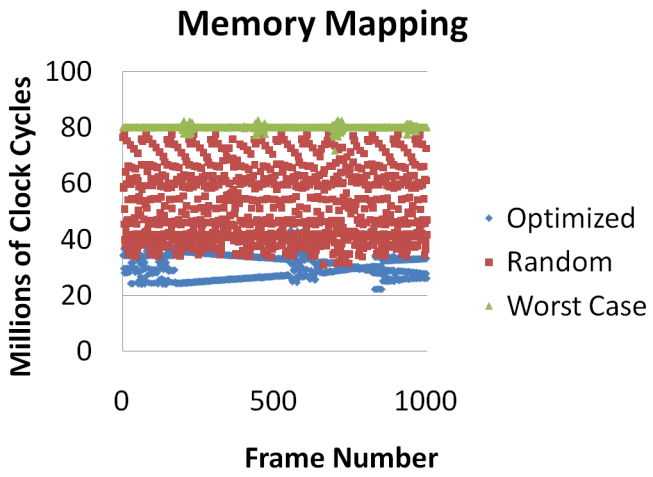

Figure 5.14: Full frame processing with different memory mapping

conflicts rates and creates a pattern that repeats every four frames. This means that the randomly chosen memory locations produced one good setup and three slower setups. These three randomly generated setups are still faster than the worst case setup because they are utilizing all four memory banks.

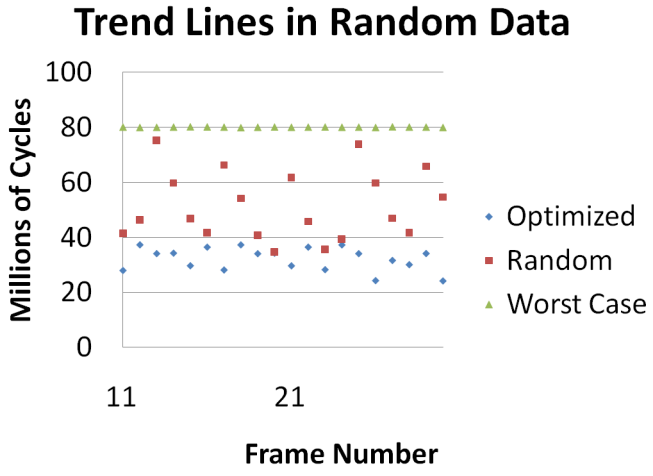

Figure 5.15: Trend lines in random memory mapping

5.2.6 Sum of Absolute Differences

Sum of Absolute Differences (SAD) is a process that compares two values. In video processing, it is typically used to determine the difference between two chunks of video. This solution uses SAD to find out where a corner has moved to in the sequential frame. The Blackfin processor has built in macros that allow it to operate more effectively.

Table 5.5 shows the difference between one execution of each different SAD function. Each function comparing a single 16×16 block against another 16×16 block. The three different algorithms are described in section 4.2.5. In table 5.5,

Name	Reference	With Macros	Macros and Aligned Memory
Cycles	876	570	301

Table 5.5: Comparing Single Execution of Different SAD Algorithms in Cycles

the reference function takes 876 cycles to complete a single comparison. The function with built-in Blackfin macros completes the same task in 570 cycles. This code is setup to only compare chunks of memory that align on four byte boundaries, so the function that assumes that the memory is aligned completes the comparison in only 301 cycles. This is nearly a three times increase in speed per comparison.

The next test calculates the cycles required to search for a match per feature found. The speed increase does not hold up when comparing the entire search process. On average, the function with built-in macros performs in 46% of the cycle counts. The function that assumes aligned memory provides a 52% decrease in cycle counts per search. The search bounding box causes some iterations to fail immediately. The minimum values show that the function will return before

Name	Reference	Builtin Macros	Assume Aligned Memory
Average	1.50	0.84	0.73
Min	0.01	0.01	0.01
Median	1.59	0.83	0.68
Max	2.20	1.44	1.32

Table 5.6: Comparing Different SAD Algorithms in Million of Cycles

any processing is done. The median values show a bigger increase in speed by both functions. There is a 45% reduction in cycles for the function with built-in macros and a 67% decrease in cycles for the built in macros with aligned memory. The maximum values are more consistent with the average values and still show a significant increase in speed even in the worst case.

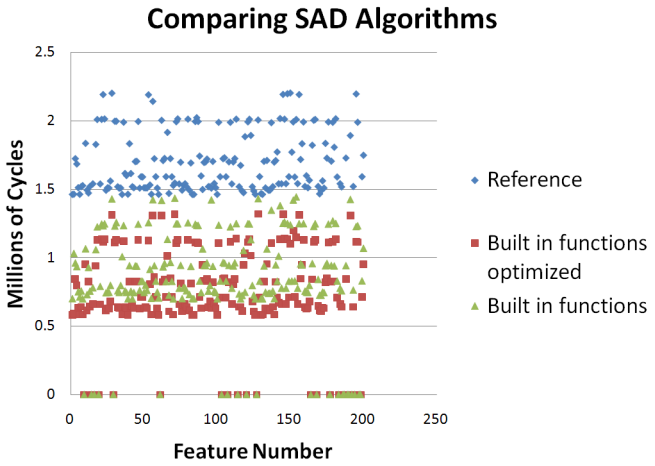

Figure 5.16: Comparing SAD Algorithms

The original algorithm, described in section 4.1.2, uses the Pyramid Template Matching scheme to find matches. Since the Blackfin has built in macros to aid in computing the sum of absolute differences between two blocks, the search algorithm must change to utilize that. Table 5.7 shows the difference between searching for a matching feature. The search by SAD method is significantly

Name	Reference	Triangle
Average	1.50	3.98
Min	0.01	1.42
Median	1.59	2.70
Max	2.20	16.41

Table 5.7: SAD vs Pyramid Template Matching in Million of Cycles

faster than the pyramid template matching. On average, it is over 263% faster. The bounding box is more flexible during pyramid template matching so it had a higher minimum value because no features failed the bounding box test. The median values are the best comparison between the two algorithms because the bounding box created a lot of outliers for the reference algorithm. The pyramid template matching algorithm is only about half the speed of the SAD function when comparing the median value. The max value for the pyramid template matching algorithm is eight times slower than the SAD function because it will search quite a bit more area if it finds close enough matches each iteration. This is likely due to the 5% failure rate discussed in [21].

Figure 5.17 shows why the median value is the appropriate value to compare between the two functions. Figure 5.17 shows the computation times for the reference sum of absolute differences versus the pyramid template matching (PTM) algorithm. The outliers are clearly defined in the figure. The standard deviation for the PTM algorithm's results is 3.24. There are 27 data points higher than the first standard deviation. This accounts for 13.5% of the data points. There are 17 data points over the second standard deviation. With worst case optimizations, the pyramid template matching algorithm should compete closely with the SAD function.

These results are for searching for a single match. This means that this

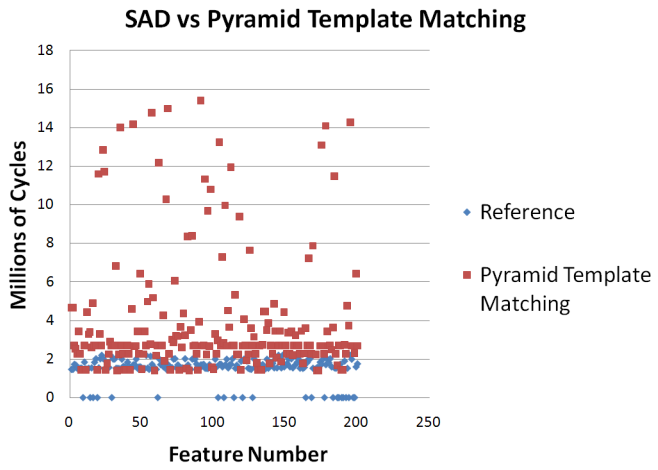

Figure 5.17: Comparing SAD vs Pyramid Template Matching

optimization is highly affected by the amount of features found. If there are only a few features found then this optimization doesn't help as much on a per frame basis. If there are a lot of found features though, then this will quickly become a very important optimization.

5.2.7 Deinterlacing

Deinterlacing provides the option of not copying an entire field at the cost of some quality. This is discussed in section 4.2.3.

Table 5.8 shows the speed increase by enabling de-interlacing. Table 5.8 shows

Name	Deinterlacing	No Deinterlacing
Average	23.49	30.03
Min	19.91	20.10
Median	23.82	30.01
Max	33.39	41.51

Table 5.8: Algorithm Runtime With or Without Deinterlacing enabled in Millions of Cycles

that deinterlacing reduces the clock cycles required to process a frame by 22% on average. The minimum cycles are comparable when the frames associated with that iteration are not residing in conflicting memory locations. This allows the background transfer to complete just as quickly regardless of the additional work required by not deinterlacing. The median is nearly identical to the average. The maximum has a 20% difference in clock cycles. This is expected because the increase of the feature count has the same effect on either method.

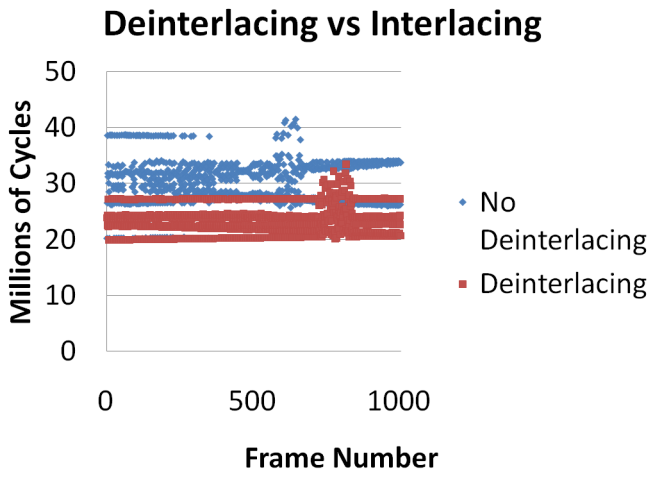

Figure 5.18: Frame Processing With vs Without Deinterlacing Enabled

Figure 5.18 shows the difference between running deinterlacing and not running it. The burst of features is shown in both graphs. They are offset by 20% because of the difference in speed. The deinterlacing function handles 20% more frames than the no deinterlacing algorithm. This means that the feature inducing segment will show up with a fewer number of frames. In this case that is around frame 600.

When speed is the number one priority, deinterlacing should always be on. It provides a 20% increase in speed.

5.2.8 Loop Unrolling

Loop unrolling is a classic optimization often performed by compilers [13]. It is described in section 4.2.4. The loop tested in this solution is part of the feature detection algorithm and is described in section 4.2.4. It iterates a specific number of times and accesses a memory location in different locations every other instruction. By unrolling the loop and giving individual pointers for each position in the diamond, the number of instructions and clock cycles is reduced dramatically.

Name	Loop	Unrolled Loop
Average	35.00	30.03
Min	29.26	19.98
Median	33.16	30.07
Max	44.10	40.87

Table 5.9: Loop vs Unrolling a Loop in Millions of Cycles

Table 5.9 shows the difference in run times between a normal loop with offsets to access memory versus unrolling the loop and having individual pointers for accessing memory. On average, the algorithm runs 14% faster when unrolling the loop. The unrolled code works better on minimum cases as well because of early returns condition and not wasting time with checking those return conditions until it is possible to fail. The median values are closer to each other with only a 10% increase in speed. The maximum difference is only 7.5%.

Figure 5.19 is the data points plotted comparing the speed before and after the unrolling of the function.

Figure 5.19 shows 484 of the unrolled data points are below the minimum value on the loop results line. It also shows that since the loop results line grabs frames slower than the unrolled algorithm, it has the feature increase earlier in

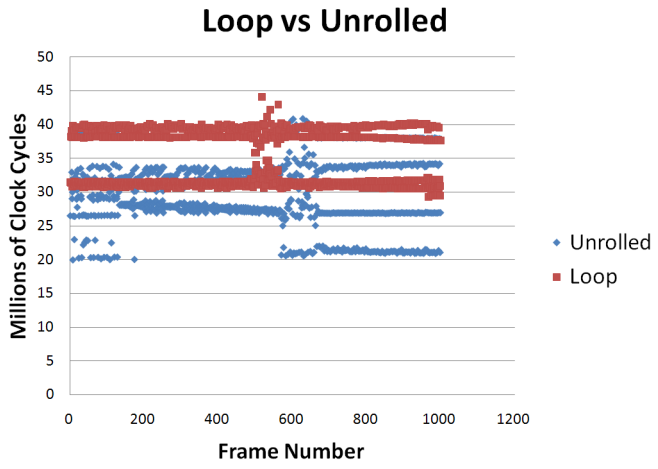

Figure 5.19: Loop vs Unrolling a Loop

the graph than the unrolled. There are also three trend lines per algorithm. This could happen because of memory placement and memory conflicts.

5.2.9 Branch Prediction

One of the optimizations the Blackfin provides is the ability to identify if a conditional block is expected be true or false most of the time.

One of the inner loops in the algorithm is best if assumed false. This if block will be true about 255 out of every 1000 times the branch is executed. Marking the if block to expect a false condition allows the compiler to generate faster running code. The results are worse if no branch prediction is used because the compiler assumes that the if expression will be true by default. The algorithm slows down by an average of 1.4 million cycles if branch prediction is not used. This is roughly an increase of 5% speed. Table 5.10 shows this 5% average increase. This increase is present in all values except the worst case.

Name	Expected True	Expected False	Neither
Average	31.17	30.01	31.46
Min	22.39	20.13	21.25
Median	31.29	29.81	31.86
Max	42.69	42.02	42.95

Table 5.10: Branch Prediction

5.2.10 Transfer Next Frame in Background

The algorithm utilizes MDMA to transfer the next frame into memory while the algorithm is processing the current frame. This minimizes the wait time between frames. Table 5.11 shows the difference in wait time between using MDMA to transfer the next frame in the background while processing a frame and if the algorithm started the transfer at the end of the processing.

Name	Transfer in Background	Transfer Between Frames
Average	0.000136	9.85
Min	0.000135	8.49
Median	0.000136	9.74
Max	0.000139	11.43

Table 5.11: Wait time Between Frames in Millions of Cycles

Table 5.11 shows that using MDMA to capture the next frame while processing the current frame on average drastically reduces the cycle counts between frames. On average, there is a difference of nearly ten million cycles, which is a delay of 16.67ms. That is over half of the processing time per frame at 30 FPS and one third of the time for this solution's 15 FPS goal. In the minimum and median values, only 135 and 136 cycles, respectively, are executed between the end and beginning of a frame. The maximum clock cycle count only required an additional 3 clock cycles.

Figure 5.20 graphically displays the significant increase in wait times between

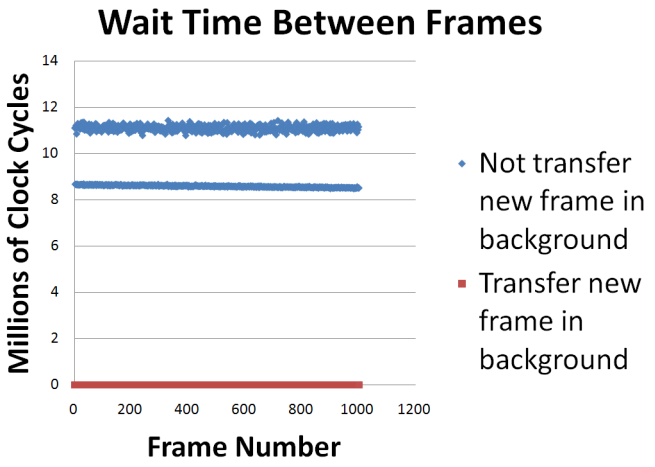

Figure 5.20: Wait Time Between Frames in Millions of Cycles

frames if MDMA is not used to transfer memory in the background process. There are two distinct trend lines in the wait time while transferring a frame. This is because of different memory paths.

The results presented in table 5.11 show significant improvement in reducing wasted time between each frame. This makes sense because the algorithm only needs a new frame to start the next iteration. Transferring data in the background while the current frame is processed reduces wait time between frames, but this additional memory utilization could affect the performance of the algorithm because of memory conflicts. Table 5.12 compares the cycles required to process one frame with or without the additional overhead of transferring the next frame in the background. Table 5.12 shows that the algorithm is not affected by transferring the next frame in the background. On average the algorithm even operates slightly faster while transferring the next frame by .73 million cycles frames. Asynchronous MDMA transfers do not require clock cycles to execute. Additional clock cycles would be from the synchronous MDMA transfers while the algorithm waits for it to complete if a memory conflict occurs. The median

Name	MDMA in Background	No MDMA in Background
Average	30.00	30.73
Min	19.67	20.55
Median	29.25	30.17
Max	40.10	37.26

Table 5.12: Algorithm Clock Cycles with Transferring Next Frame in Background vs Not Transferring

also shows that they are comparable algorithms most of the time. The minimum and maximum values are close to each other. The algorithms are very similar to each other while only looking at statistics.

These statistics may be misleading. The plotted data provides additional insight into how the algorithms compare since the statistics in table 5.12 do not show a significant difference between the two algorithms. Figure 5.21 shows an

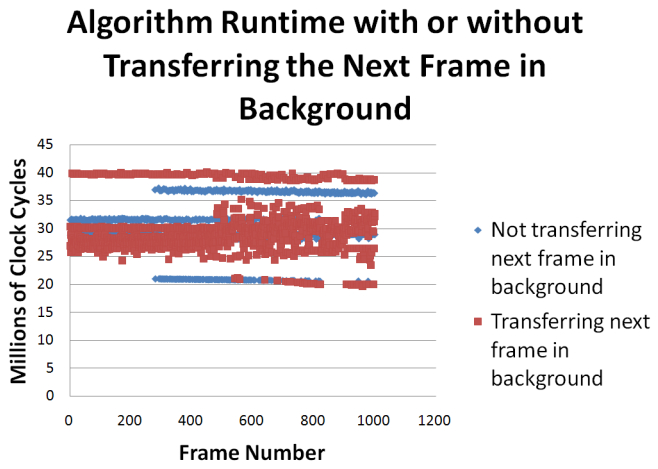

Figure 5.21: Algorithm Clock Cycles with Transferring Next Frame in Background vs Not Transferring

interesting trend with the minimum and maximum values. Processing a frame without transferring a frame in the background requires the minimum amount of cycles more often than with transferring. Out of 1000 samples, the data for the algorithm that transfers a frame in the background only contains 26 data

points under 21 million clock cycles. This compares to the 94 data points under 21 million clock cycles for the algorithm that does not transfer the frame in the background. The maximum shows more of this trend. There are 129 data points in the transferring next frame in the background data that are greater than the maximum of not transferring in the background. The average and median between the two sets of data even out because over a fourth of the samples are near the upper bounds of range for the algorithm not transferring data in the background.

The last test associated with using asynchronous MDMA is to combine the previous two tests and show that the overall algorithm is optimized. The results are shown in table 5.13. Table 5.13 shows that on average, the background

Name	Transfer in Background	Transfer Between Frames
Average	30.25	40.04
Min	24.14	27.38
Median	28.84	40.04
Max	40.49	52.68

Table 5.13: Algorithm and Frame Capture Time in Millions of Cycles

MDMA gives a net gain of about 16.67ms per frame processed. This means that it is worth it to transfer data in the background while processing the current frame. The maximum and min spreads are very far apart.

Figure 5.22 shows different trend lines. The majority of data points for transferring the next frame asynchronously are closer to the minimum values. For the algorithm that synchronously waits for the next frame to transfer, 913 data points are above 37 million cycle points compared to 127 of the asynchronous transfer. Exactly half of the synchronous data points are above the maximum of the asynchronous trend line.

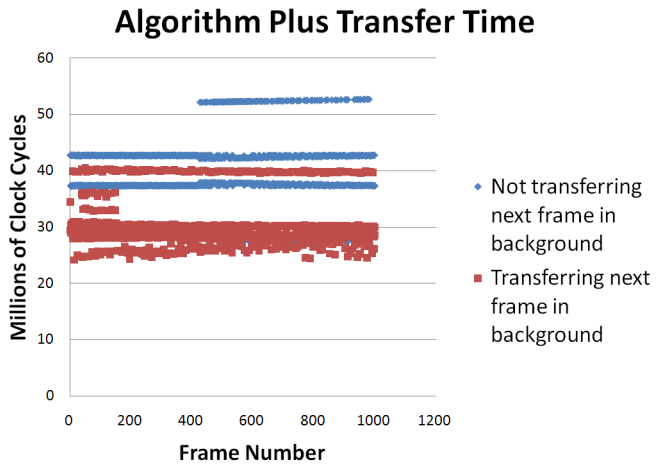

Figure 5.22: Algorithm and Frame Capture Time in Millions of Cycles

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This solution successfully stabilizes video. It cuts out high frequency jitter while following trend lines to establish true intended motion. The delay is acceptable at one fourth of a second and the algorithm succeeds the initial goal of 15 Frames Per Second (FPS). The average frame in its final configuration takes an average of 30.43 million cycles. On average this means around 20 FPS. The median is 30.19 million cycles. The min is 22.16 million cycles and the longest is 43.23 million cycles.

If the operator chooses to enable de-interlacing, then the clock cycle count on average is 23.49 million cycles. This means on average the algorithm runs at 25 FPS. This is 10 FPS higher than the expected results.

The optimizations presented in this solution make this algorithm able to run on a Blackfin embedded processor. Without any of the optimizations, the algorithm would run at 3.2 FPS. With optimizations the frame rate improves to over

20 FPS. Unoptimized, the algorithm takes 187 million clock cycles per frame to process and have a display delay of 1.2 seconds. Operators would not be able to control the platform with a delay that large. This FPS is not above the requested minimum FPS. The optimizations reduced the clock cycles required per frame by 157 million cycles from 187 million to 30 million. The delay between a captured and displayed frame is reduced from 1.2 seconds to 200 milliseconds.

Optimization Type	**Reduction in Clock Cycles**
Searching only the middle pixels	54.58
L3 Memory Placement	49.62
MDMA instead of double for loop	15.74
Profile matching w/ SAD instead of PTM	9.92
Transfer next frame in background	9.79
Deinterlacing	6.54
Loop unrolling	5
Transfer MDMA vs Memcopy	3.15
Built in SAD vs Reference	3.08
Branch prediction	1.45

Table 6.1: How Much Each Optimization Reduced Clock Cycles per Frame in Millions of Clock Cycles

Table 6.1 shows how each optimization affected the frame's processing speed. The change that made the biggest difference is searching only a subset of the frame. Since this is fairly obvious, and not embedded programming specific, the biggest Blackfin specific optimization is memory location. Optimizing the memory location increased the algorithm from 7.49 FPS to 20 FPS. The biggest optimization a programmer can use with embedded programming is to analyze how and when memory is accessed.

The optimization that least affected the algorithm is the branch prediction. Even though that branch is evaluated on average 84 times a frame, it only sped up the algorithm by 1.45 million clock cycles, which is less than one FPS.

A few of the optimizations really shine when the full area is searched and not just the middle. The branch prediction is evaluated on average over 2,000 times per frame. This is also true for the feature matching optimizations. On average with a search field of 112×243 there are less than 4 features found per full sized frame. A lot more features are found with a full 720×243 frame.

6.2 Future Work

There are potential improvements and applications for this algorithm. The first improvement is to find out what is causing the different trend lines in the results. Each test has two to four unique trend lines that are currently not explained. This is seen in figure 5.12. Memory placement and memory conflicts are the expected explanation for these oddities. If the reason is found, then the algorithm will become much faster on average.

The second improvement is to figure out why Figure 5.13 had a slow down towards the end of the test. This is an oddity and the algorithm would benefit from knowing the answer.

The algorithm would stabilize a lot better with more than just a search window of 112×243. Not only is this a small percentage of the screen, the center is also the place where operators place objects of interest. These objects of interest will usually have independent motion from the screen and may bias the stabilization incorrectly.

Although most of the memory that is in L3 and L2 could not fit into the additional space generated by disabling instruction caching, it would be interesting to see the advantages of utilizing that additional instruction space for more data

variables. This could allow data arrays to be split into different sub banks in L1 to improve speed even further. This could highly optimize the Compute Point part of the algorithm. It currently moves data into L2 and searches multiple lines at the same time. If these are in the same sub bank, then there are potential memory conflicts. If more L1 is available, the algorithm may be able to split up the lines between some sub banks in L1 and L2.

David Johansen suggested that a low pass filter tuned correctly may have better results than the Parabolic Fit Camera. This could be explored to see if the stabilization is improved. He also uses Harris Corner detection in his algorithm. Optimizing that algorithm to give the user the option of increasing stabilization performance at the cost of speed is an advantageous feature.

Appendix

Reference Code

Memory Transfer

```
for(rowCount = 0; rowCount < 7; rowCount++)
{
   gimgOffset = rowCount * (BCD_MOVE_A_LINE_OFFSET);
   for(colCount = 0; colCount < BCD_COL_COUNT; colCount++)
   {
       int memRowCount;
       for(memRowCount = 0; memRowCount < BCD_L1_HEIGHT;
          memRowCount++)
       {
           memcpy(firstBlock + (memRowCount * BCD_L1_WIDTH)
              , gimgInput + gimgOffset + (memRowCount * (
              FS_WIDTH)), BCD_L1_WIDTH);
       }
       binaryCornerDetectionOptimized(firstBlock, bcdSmooth
          , bcdBinary);
       for(memRowCount = 1; memRowCount < BCD_L1_HEIGHT -
          1; memRowCount++)
       {
           memcpy(binaryImage + gimgOffset + (memRowCount *
               (FS_WIDTH)) + 1, bcdBinary + (memRowCount *
              BCD_L1_WIDTH) + 1, BCD_L1_WIDTH - 2);
       }
       gimgOffset += BCD_MOVE_A_BOX_OFFSET;
   }
}
```

Listing 1: L3 to L1 Memory Transfer via MemCopy

```
int myTransferSize = 4;
l1Src.StartAddress = bcdBinary;

for(rowCount = 0; rowCount < 7; rowCount++)
{
   gimgOffset = rowCount * (BCD_MOVE_A_LINE_OFFSET);
   for(colCount = 0; colCount < BCD_COL_COUNT; colCount++)
   {
       int memRowCount;
       newFrameDMA.StartAddress = gimgInput + gimgOffset;

       if((unsigned int)newFrameDMA.StartAddress % 4 != 0)
       {
          if((unsigned int)newFrameDMA.StartAddress % 2 !=
             0)
          {
              newFrameDMA.XCount = BCD_L1_WIDTH;
              myTransferSize = 1;
              newFrameDMA.YModify = FS_WIDTH - BCD_L1_WIDTH
                  + 1;
              smallAreaDMA.XCount = BCD_L1_WIDTH;
          }
          else
          {
              newFrameDMA.XCount = BCD_L1_WIDTH / 2;
              myTransferSize = 2;
              newFrameDMA.YModify = FS_WIDTH - BCD_L1_WIDTH
                  + 2;
              smallAreaDMA.XCount = BCD_L1_WIDTH / 2;
          }
          newFrameDMA.XModify = myTransferSize;

       }
       else
       {
          newFrameDMA.XModify = 4;
          newFrameDMA.XCount = BCD_L1_WIDTH / 4;
          newFrameDMA.YModify = FS_WIDTH - BCD_L1_WIDTH +
             4;
          smallAreaDMA.XCount = BCD_L1_WIDTH / 4;
       }
       newFrameDMA.YCount = BCD_L1_HEIGHT;

       smallAreaDMA.StartAddress = firstBlock;
       smallAreaDMA.XModify = myTransferSize;
       smallAreaDMA.YCount = BCD_L1_HEIGHT;
       smallAreaDMA.YModify = myTransferSize;

       ADI_DMA_RESULT dmaVal = adi_dma_MemoryCopy2D(
```

```
            dmaHandle1, &smallAreaDMA, &newFrameDMA,
            myTransferSize, 0);

47          binaryCornerDetectionOptimized(firstBlock, bcdSmooth
            , bcdBinary);

            l3Src.StartAddress = binaryImage + gimgOffset +
            FS_WIDTH;
            int myTransferSize2 = 4;
            if((unsigned int)l3Src.StartAddress % 4 != 0)
52          {
            if((unsigned int)l3Src.StartAddress % 2 != 0)
            {
                l3Src.XCount = BCD_L1_WIDTH;
                l3Src.YModify = FS_WIDTH - BCD_L1_WIDTH + 1;
57              l1Src.XCount = BCD_L1_WIDTH;
                myTransferSize2 = 1;
            }
            else
            {
62              l3Src.XCount = BCD_L1_WIDTH / 2;
                l3Src.YModify = FS_WIDTH - BCD_L1_WIDTH + 2;
                l1Src.XCount = BCD_L1_WIDTH / 2;
                myTransferSize2 = 2;
            }
67          l3Src.XModify = myTransferSize2;
            }
            else
            {
                 l3Src.XModify = 4;
72               l3Src.XCount = BCD_L1_WIDTH / 4;
                 l3Src.YModify = FS_WIDTH - BCD_L1_WIDTH + 4;
                 l1Src.XCount = BCD_L1_WIDTH / 4;
            }

77          l1Src.XModify = myTransferSize2;
            l1Src.YModify = myTransferSize2;
            l3Src.YCount = BCD_L1_HEIGHT - 2;
            l1Src.YCount = BCD_L1_HEIGHT - 2;

82          dmaVal = adi_dma_MemoryCopy2D(dmaHandle2, &l3Src, &
            l1Src, myTransferSize2, 0);
            gimgOffset += BCD_MOVE_A_BOX_OFFSET;
        }
    }
```

Listing 2: L3 to L1 Memory Transfer via MDMA

Different Memory Placements

```
// Bank 0

// Bank 1

// Bank 2
RESOLVE(_sFrame1IN,      0x02000000)
RESOLVE(_sFrame1OUT,     0x02100000)
RESOLVE(_sFrame3IN,      0x02200000)
RESOLVE(_binaryImage1,   0x02300000)
RESOLVE(_gimgFull1,      0x02400000)
RESOLVE(_gimg1,          0x02500000)
RESOLVE(_gimg2,          0x02600000)
RESOLVE(_binaryImage2,   0x02700000)
RESOLVE(_movedFrame,     0x02800000)
RESOLVE(_smooth1,        0x02900000)
RESOLVE(_gimgFull2,      0x02A00000)
RESOLVE(_sFrame2OUT,     0x02B00000)
RESOLVE(_sFrame0OUT,     0x02C00000)
RESOLVE(_sFrame3OUT,     0x02D00000)
RESOLVE(_fullBlankFrame,0x02E00000)

// Bank 3
RESOLVE(_sFrame2IN,      0x03000000)
RESOLVE(_sFrame0IN,      0x03100000)
RESOLVE(_historyFrame9, 0x03200000)
RESOLVE(_historyFrame0, 0x03400000)
RESOLVE(_historyFrame1, 0x03500000)
RESOLVE(_historyFrame2, 0x03600000)
RESOLVE(_historyFrame3, 0x03700000)
RESOLVE(_historyFrame4, 0x03800000)
RESOLVE(_historyFrame5, 0x03900000)
RESOLVE(_historyFrame6, 0x03A00000)
RESOLVE(_historyFrame7, 0x03B00000)
RESOLVE(_historyFrame8, 0x03C00000)
```

Listing 3: L3 Memory in two banks

```
// Bank 0
RESOLVE(_historyFrame9, 0x00100000)
RESOLVE(_historyFrame1, 0x00200000)
RESOLVE(_historyFrame5, 0x00300000)
RESOLVE(_sFrame3IN,     0x00400000)
RESOLVE(_gimgFull1,     0x00500000)
RESOLVE(_smooth1,       0x00600000)
RESOLVE(_fullBlankFrame,0x00700000)

// Bank 1
RESOLVE(_historyFrame4, 0x01000000)
RESOLVE(_gimg2,         0x01100000)
RESOLVE(_gimgFull2,     0x01200000)
RESOLVE(_sFrame0OUT,    0x01300000)
RESOLVE(_sFrame3OUT,    0x01400000)



// Bank 2
RESOLVE(_historyFrame0, 0x02000000)
RESOLVE(_historyFrame3, 0x02100000)
RESOLVE(_historyFrame7, 0x02200000)
RESOLVE(_historyFrame8, 0x02300000)
RESOLVE(_gimg1,         0x02400000)

// Bank 3
RESOLVE(_sFrame1IN,     0x03000000)
RESOLVE(_sFrame2IN,     0x03100000)
RESOLVE(_sFrame0IN,     0x03200000)
RESOLVE(_historyFrame2, 0x03300000)
RESOLVE(_historyFrame6, 0x03400000)
RESOLVE(_sFrame1OUT,    0x03500000)
RESOLVE(_binaryImage1,  0x03600000)
RESOLVE(_binaryImage2,  0x03700000)
RESOLVE(_movedFrame,    0x03800000)
RESOLVE(_sFrame2OUT,    0x03900000)
```

Listing 4: L3 Memory Random Banks

Sum of Absolute Difference Functions

```
int vid_SAD16x16(char *image, char *block, int imgWidth)
{
    int dist = 0;
    int x, y;
    long long srcI, srcB;
    char *ptrI, *ptrB;
    int bytesI1, bytesI2, bytesB1, bytesB2;
    int sum1, sum2, res1, res2;

    sum1 = sum2 = 0;
    bytesI2 = bytesB2 = 0;

    // Does not use imgwidth?
    // ptrI and ptrB never change because image and block
       never change
    for(y = 0; y < 16; y++) {
        ptrI = image;
        ptrB = block;
        for (x = 0; x < 16; x += 8) {
            bytesI1 = loadbytes((int*)ptrI); ptrI += 4;
            bytesB1 = loadbytes((int *)ptrB); ptrB += 4;
            srcI = compose_i64(bytesI1, bytesI2);
            srcB = compose_i64(bytesB1, bytesB2);
            saa(srcI, ptrI, srcB, ptrB, sum1, sum2, sum1,
               sum2);

            bytesI2 = loadbytes((int *)ptrI); ptrI += 4;
            bytesB2 = loadbytes((int *)ptrB); ptrB += 4;
            srcI = compose_i64(bytesI1, bytesI2);
            srcB = compose_i64(bytesB1, bytesB2);
            saar(srcI, ptrI, srcB, ptrB, sum1, sum2, sum1,
               sum2);
        }
    }
    extract_and_add(sum1, sum2, res1, res2);
    dist = res1 + res2;

    return dist;
}
```

Listing 5: Sum of Absolute Differences Broken Function

```
int ref_SAD16x16(unsigned char *image, unsigned char *block
    , int imgWidth)
{
    int dist = 0;
    int x, y;

    for (y = 0; y < 16; y++) {
      for (x = 0; x < 16; x++)
        dist += abs(image[x] - block[x]);
      image += 16 + (imgWidth-16);
      block += 16;
    }
    return dist;
}
```

Listing 6: Sum of Absolute Differences Reference Function

```
int newer_SAD16x16(char *image, char *block, int imgWidth){
        int dist = 0;
        int x, y;
        long long srcI, srcB;
        char *ptrI, *ptrB;
        int bytesI1, bytesI2, bytesB1, bytesB2;
        int sum1, sum2, res1, res2;
        sum1 = sum2 = 0;
        bytesI2 = bytesB2 = 0;

        ptrI = image;
        ptrB = block;

        for (y = 0; y < 16; y++) {
          for (x = 0; x < 16; x += 8) {
            bytesI1 = loadbytes((int *)ptrI); ptrI+=4;
            bytesI2 = loadbytes((int *)ptrI); ptrI+=4;
            bytesB1 = *(int *)ptrB; ptrB += 4;
            bytesB2 = *(int *)ptrB; ptrB += 4;
            srcI = compose_i64(bytesI1, bytesI2);

            srcB = compose_i64(bytesB1, bytesB2);

            saa(srcI, image, srcB, block, sum1, sum2, sum1
               , sum2);
            bytesI1 = loadbytes((int *)ptrI);
            srcI = compose_i64(bytesI1, bytesI2);
            saar(srcI, image, srcB, block, sum1, sum2,
               sum1, sum2);

          }
          ptrI += (imgWidth - 16);
        }
        extract_and_add(sum1, sum2, res1, res2);

        dist = res1 + res2;
        return dist;
 }
```

Listing 7: Sum of Absolute Differences Built in Macros

```
int better_SAD16x16(unsigned char *image, unsigned char *
   block, int imgWidth){
  int x, y;
  long long srcI, srcB;
  int bytesI1, bytesI2, bytesB1, bytesB2;
  int sum1, sum2, res1, res2;
  sum1 = sum2 = 0;
  bytesI2 = bytesB2 = 0;

  /* get 4 byte aligned pointers */
  int *iPtr = ((int)image)&~3;
  int *bPtr = ((int)block)&~3;

  for (y = 0; y < 16; y++) {
    bytesI1 = *iPtr;
    bytesB1 = *bPtr;

    for (x = 0; x < 16; x += 8) {
      iPtr++; bytesI2 = *iPtr++;
      bPtr++; bytesB2 = *bPtr++;

      srcI = compose_i64(bytesI1, bytesI2);
      srcB = compose_i64(bytesB1, bytesB2);

      saa(srcI, image, srcB, block, sum1, sum2, sum1, sum2)
         ;
      bytesI1 = *iPtr;
      bytesB1 = *bPtr;

      srcI = compose_i64(bytesI1, bytesI2);
      srcB = compose_i64(bytesB1, bytesB2);

      saar(srcI, image, srcB, block, sum1, sum2, sum1, sum2
         );
    }
    iPtr += (imgWidth - 16)/4;
  }
  extract_and_add(sum1, sum2, res1, res2);
  return res1 + res2;
}
```

Listing 8: Sum of Absolute Differences Built in Functions Assuming Aligned Data

Bibliography

[1] A. Ajay, S. Deepti, C. Rajendra, N. Venkatesan, and D. S. David. Resizing of images by arbitrary factors in the spatial domain and implementation on blackfin bf533 processor. Technical report, National institute of technology karnataka, surathkal, India, 2006.

[2] Altera. Video and image processing design using fpgas. Technical report, Altera.

[3] Analog Devices. *Chip Scale PAL/NTSC Video Encoder with Advanced Power Management*, 2004.

[4] Analog Devices. *MultiFormat SDTV Video Decoder ADV7183B Manual*, 2005.

[5] Analog Devices. *ADSP-BF561 Blackfin Processor Hardware Reference*, January 2010.

[6] Analog Devices. *Compiler and Library Manual for Blackfin Processors*, January 2011.

[7] Analog Devices. *Blackfin Processor Programming Reference*, September 2008.

[8] D. Apps. Extended-precision fixed-point arithmetic on the blackfin platform. Engineer to Engineer Note 186, Analog Devices, May 2003.

[9] J.-Y. Bouguet. Pyramidal implementation of the lucas kanade feature tracker. Description of the algorithm, Intel Corporation, 2000.

[10] P. J. Burt and E. H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, COM-31(4):532–540, 1983.

[11] P. Cavill. Fpga or dsp for military applications? both have their place. http://www.dsp-fpga.com/pdfs/Radstone.Oct05.pdf, 2005.

[12] I. Cohen and G. Medioni. Detection and tracking of objects in airborne video imagery. Technical report, University of Southern California.

[13] K. D. Cooper and L. Torczon. *Engineering a compiler*. Morgan Kaufmann Publishers, 2004.

[14] A. Devices. Adi blackfin outperforms arm-based freescale i.mx processors in eembc out of the box c benchmarks. http://embedded-computing.com/adi-mx-processors-eembc-out-of-the-box-benchmarks, March 2007.

[15] D. Franklin and J. Seng. Experiences with the blackfin architecture for embedded systems education. Technical report, California Polytechnic State University.

[16] R. Getz. Making the blackfin perform. Technical report, Analog Devices.

[17] B. K. P. Horn. *Robot Vision*. The MIT Press, Massachusetts, 1992.

[18] M. Irani and P. Anandan. All about direct methods. Technical report, The Weizmann Institute of Science, Israel and Microsoft Research of Redmond, 1999.

[19] K. Jack. *Video Demystified*. Newnes, 2007.

[20] X. T. Jian Sun, Weiwei Zhang and H.-Y. Shum. Background cut. Technical report, Microsoft Research Asia, Beijing, China, 2006.

[21] D. L. Johansen. Video stabilization and target localization using feature tracking with small uav video. Master's thesis, Brigham Young University, December 2006.

[22] B. Jung and G. S. Sukahatme. Detecting moving objects using a single camera on a mobile robot in an outdoor environment. *8th Conference on Intelligent Autonomous Systems*, pages 980–987, March 2004.

[23] G. Kadziolka. System development and programming for the adsp-bf533. Massachusetts, April 2006.

[24] D. Katz, T. Lukasiak, and R. Gentile. Use of video technology to improve automotive safety becomes more feasible with blackfin. *Analog Dialogue*, 38(1):9–14, 2004.

[25] R. Kumar, H. Sawhney, S. Samarasekera, S. Hsu, H. Tao, Y. Guo, K. Hanna, A. Pope, R. Wildes, D. Hirvonen, M. Hansen, and P. Burt. Aerial video surveillance and exploitation. *Proceedings of the IEEE*, 89(10):1518–1539, October 2001.

[26] K.-Y. Lee, Y.-Y. Chuang, B.-Y. Chen, and M. Ouhyoung. Video stabilization using robust feature trajectories. Technical report, National Taiwan University, 2009.

[27] Z. Li, D. Li, Q. Zhang, and Q. Wu. Design and implementation of blackfin dsp-based video encoder in network surveillance system. Technical report, Communication University of China, Beijing.

[28] D. G. Lowe. Distinctive image features from scale invariant keypoints. Technical report, University of British Columbia, 2004.

[29] B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. Technical report, Carnegie-Mellon University, 1981.

[30] A. Murray and B. Franke. Fast source-level data assignment to dual memory banks. In *University of Edinburgh*, 11th, 2008.

[31] J. Paur. Uav in a firefight of a different kind. http://www.wired.com/autopia/2009/08/firefighting-uav/, August 2009.

[32] C. Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann, 2003.

[33] V. Ramanathan. Western pacific autonomous uav campaign aerosol-dust-cloud interactions and climate effects. Technical report, Scripps Institution of Oceanography, University of California, San Diego, October 2006.

[34] P. Saeedi, P. Lawrence, and D. Lowe. An efficient binary corner detector. Technical report, University of British Columbia.

[35] P. Saeedi, P. Lawrence, and D. Lowe. Vision-based 3-d trajectory tracking for unknown environments. *IEEE Transactions On Robotics*, 22(1):119–136, February 2006.

[36] K. Sanghai. Video templates for developing multimedia applications on blackfin processors. Engineer to Engineer Note 301, Analog Devices, September 2006.

[37] K. Sanghai. System optimization techniques for blackfin processors. Engineer to Engineer Note 324, Analog Devices, July 2007.

[38] K. Singh and R. Babu. Video framework considerations for image processing on blackfin processors. Engineer to Engineer Note 276, Analog Devices, September 2005.

[39] K. J. Solosky. Uavs for airborne law enforcement. http://www.officer.com/article/10233503/uavs-for-airborne-law-enforcement, June 2009.

[40] P. H. S. Torr and A. Zisserman. Feature based methods for structure and motion estimation. Technical report, Microsoft Research at Cambridge and University of Oxford, 1999.

[41] J. Wise. Civilian uavs: No pilot, no problem. http://www.popularmechanics.com/science/space/4213464, October 2009.

[42] T. Yu, C. Zhang, M. Cohen, Y. Rui, and Y. Wu. Monocular video foreground/background segmentation by tracking spatial color gaussian mixture models. Technical report, GE Global Research and Microsoft Research and ECE Northwestern University, 2007.