

2006

Real -time Retinex image enhancement: Algorithm and architecture optimizations

Glenn Derrick Hines

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Hines, Glenn Derrick, "Real -time Retinex image enhancement: Algorithm and architecture optimizations" (2006). *Dissertations, Theses, and Masters Projects*. Paper 1539623490.

<https://dx.doi.org/doi:10.21220/s2-zgww-7r76>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

REAL-TIME RETINEX IMAGE ENHANCEMENT: ALGORITHM AND ARCHITECTURE OPTIMIZATIONS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Glenn Derrick Hines

2006

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

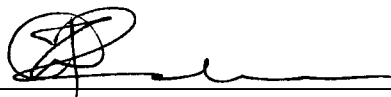


Glenn Hines

Approved, January 2006



J. Philip Kearns
Dissertation Advisor



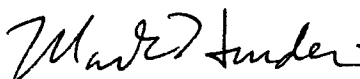
Zia-ur Rahman
Dissertation Co-Advisor



Weizhen Mao



Andreas Stathopoulos



Mark Hinder
Department of Applied Science

To my lovely wife Sunita and our adorable children Jordan, Jada and Jamison

Table of Contents

Acknowledgments	viii
List of Tables	x
List of Figures	xv
Abstract	xvi
1 Introduction	2
2 Retinex Image Enhancement	7
3 Digital Signal Processors	12
3.1 TMS320C6711	13
3.2 TMS320C6713	16
3.3 TMS320DM642	17
3.4 TMS320C6416	19
4 Test Environment	21
4.1 DSP Evaluation Modules	21

4.1.1	Video Capture and Display for C6711 and C6713 EVMs	23
4.1.2	Video Capture and Display for DM642 and C6416 EVMs	26
4.2	Development Tools	27
4.3	Test-Bed Components and Operation	28
4.4	Performance Analysis	30
4.5	Real-time Parameter Updates	31
4.6	Retinex Task Within DSP/BIOS	32
5	Optimizations and Performance Results	33
5.1	Single-Scale Monochrome Retinex Optimizations	34
5.1.1	Apply Convolution Equivalence	34
5.1.2	Pre-Compute the Kernel	36
5.1.3	Baseline Algorithm Performance	37
5.1.4	Pre-Compute the Logarithm	39
5.1.5	Use DMA to Transfer Columns	43
5.1.6	Reduce Gaussian Kernel Computations	44
5.1.7	Merge Algorithm Components	45
5.1.8	Minimize Data Transfer Overhead	46
5.1.9	Use Cache-optimized FFTs	48
5.2	Map Optimized SSMR to C6713	49
5.3	Map Optimized SSMR to DM642	50
5.3.1	Apply Intrinsic	52
5.3.2	Modify the Architecture	53

5.4	Multi-Spectral Multi-Scale Retinex Optimizations	54
5.4.1	Reuse Transformed Input Image	54
5.4.2	Reduce Computations	55
5.4.3	Buffer Across Spectral Bands	58
5.4.4	Allocate Log Values in L2 Memory	59
5.4.5	MSR Performance Results	59
6	Enhanced Vision System Case Study	68
6.1	Background	68
6.2	Image Processing Functions	70
6.3	Additional Requirements	73
6.4	Results	76
7	Future Research	82
7.1	Luma-only Retinex	82
7.2	Improving Current Performance	83
7.3	Processing Larger Format Images	84
7.4	Migrating to a Multiprocessor Environment	88
8	Conclusions	90
A	Multi-Image Registration	93
A.1	Background	94
A.2	Registration algorithms	98
A.2.1	SS algorithm	99

A.2.2	MLR algorithm	100
A.3	Results	101
A.3.1	SS algorithm	102
A.3.2	MLR algorithm	104
A.3.3	Discussion	106
A.4	Summary	108
B	Field Programmable Gate Arrays	110
C	DM642 EVM Flash Programming Guidelines	114
	Bibliography	119
	Vita	127

ACKNOWLEDGMENTS

The list of people I wish to acknowledge would require another dissertation but I will point out a few. First, and foremost I thank my research advisor, Dr. Zia-ur Rahman for his guidance, patience, and great ideas. One day the world will know and value your genius. I also thank Daniel Jobson for his wonderful musings on image processing research, and Glenn Woodell for always lending a helping hand. The seeds of the Retinex that the three of you planted have grown into quite a tree. Thank you for allowing me to contribute to a branch.

I also thank my professors at William and Mary, Doctors Torczon, Bynum, Stockmeyer, Stathopoulos, Zhang, Noonan, Prosl, Rahman, and Kearns, for all of your enlightening classroom sessions, and patiently satisfying all of my “What if you tried this?” questions. In particular, special consideration is given to Professor Kearns for giving me the opportunity to continue my research in image processing and serving as my advisor.

Grateful acknowledgement is given to Vanessa Godwin. Your advice throughout my tenure as a student was invaluable. Thanks for “penetrating the bureaucracy” for me. I also sincerely thank my committee members Weizhen Mao, Andreas Stathopoulos, and Mark Hinders for all of your comments and suggestions, and for your willingness to take time out of your schedule for me.

The work contained in this dissertation was supported by funding from NASA Langley Research Center. Sincere appreciation is extended to my managers at NASA LaRC for giving me the opportunity to pursue my research dreams. This list spans many years now but includes Dr. Thomas Shull, Pam Rinsland, Steve Jurczyk, Steve Sandford, Randy Reagan, and Kathryn Stacy. Thanks also goes to Steven Harrah for project support, Cathryn Murray-Wooddell for working your magic with resources, and George Allison for keeping the Ph.D. pipeline going at NASA LaRC. Many thanks also go out to my colleagues at NASA but especially to my old lunch bunch including Duane, Cy, Danette, Michael, Marilee, Shelley, Felicia, and Lloyd who is no longer with us. We’ve solved many of the world’s problems on napkins, now if only we can convince everyone else to listen.

I also wish to thank a few special people — My parents David and Helen Hines for raising me and my brothers, Ronnie and Brian, in a home that always valued knowledge and education, and for your good genes! Neville and Dorothy Etwaroo for doing the same for Sunita! My many extended family members for all of your words of encouragement and prayers, and my many friends including Roger Bailey, Charles Stump, Shawn Williams, Steve Green, Kenneth Arrington, Andres Alvarez, Levi Little, and Bruce Hornsby for keeping me laughing throughout the years.

And although I dedicated this document to my immediate family, I again give my utmost gratitude to my wife Sunita Etwaroo and our children Jordan Milan, Jada Nalini, and Jamison Glenn for all of your love and support. You are my air.

List of Tables

3.1	DSP Specifications	20
5.1	Initial performance results from the first implementation of the SSMR. . . .	37
5.2	Performance measurements after using logarithm tables and combining α and β	41
5.3	Performance results after using 2D DMA data transfers.	43
5.4	Performance results after using 2D DMA data transfers.	45
5.5	Performance results after merging algorithm stages. Since the forward and inverse column execution times are effectively merged together, the time to process columns is now in item “processcols”	47
5.6	Final SSMR performance results using the C6711 DSP.	49
5.7	Measured Retinex performance on DM642 and C6416 processors. The 133 and 200 refer to the clock speed of the EMIF bus. Measurement units are in both milliseconds, and frames per second in parentheses.	64
5.8	Comparison of final SSMR performance using the C6711 and the C6416 DSPs.	64
5.9	C6416 CPU Loading for different Retinex configurations.	65

6.1	Sensor Specifications	72
7.1	FFT Benchmarks for C6711, DM642 and C6416.	85
7.2	FFT Processing Time Benchmarks using C6711 and DM642 for various sized images.	86
7.3	FFT storage requirements and transfer times (based on row oriented data) for various sized images. Storage is based on complex image data stored as integers. Transfer times are based on a 64-bit EMIF bus clocked at 133 MHz.	86
A.1	Sensor Specifications	97
A.2	Updated Sensor Specifications	98
A.3	Visible to SWIR MLR Coefficients	104
A.4	LWIR to visible SWIR MLR Coefficients	105

List of Figures

2.1	The top row of images from left to right have simulated tungsten, fluorescent, and sunlight illumination sources. The bottom row has the same images after Retinex processing. The effects of the different illumination sources is nearly completely removed.	8
2.2	Many image processing algorithms would either saturate the bright regions or clip the dark regions of the image on the left. The Retinex processed image on the right appears almost uniformly illuminated without exhibiting these effects.	9
2.3	On the left is a low contrast, dimly lit grayscale digital image; on the right is the single-scale Retinex processed image — single-scale processing increases the contrast and sharpness.	10
3.1	Primary DSP components include the CPU, L1 Data Cache, L1 Program Cache, L2 memory (SRAM/Cache) and EDMA Controller.	13
3.2	General outline of 2-level internal memory architecture of C67x processors. The dashed boxes are user addressable memory.	15
3.3	Configuration modes for the C6711 L2 memory.	16

3.4	Block diagram of primary DM642 components. The DM642 has special instruction extensions to accelerate video applications.	17
3.5	Block diagram of primary C6416 Components. Note the larger L2 memory and 64-bit EMIF bus.	20
4.1	Picture of DM642 EVM board. Numerous components are on the EVM circuit board to support testing the DSP for a wide variety of applications. We primarily use the peripherals associated with video capture and display.	22
4.2	IDC video capture subsystem	24
4.3	IDC video display subsystem	25
4.4	DM642 EVM block diagram	27
4.5	C6416 EVM block diagram	28
4.6	Block diagram of the test-bed — the Host PC only provides setup information to the EVM; after initiation, the DSP executes independently.	29
5.1	Capture Video Frame with input from camera on the left, and Retinex output on the right. Retinex parameters are $\alpha = 175$, $\beta = 135$, and $\sigma = 80$ — note that we are nearly reaching the noise limit of the camera.	50
5.2	Retinex performance in time (bottom axis) and frames per second (top axis) to process 1 spectral band of image data on DM642 with 133 MHz EMIF (dotted line), DM642 with 200 MHz EMIF (dashed line), and C6416 (full line).	61
5.3	Retinex performance in time (bottom axis) and frames per second (top axis) to process 2 spectral bands of image data on DM642 with 133 MHz EMIF (dotted line), DM642 with 200 MHz EMIF (dashed line), and C6416 (full line).	62

5.4	Retinex performance in time (bottom axis) and frames per second (top axis) to process 3 spectral bands of image data on DM642 with 133 MHz EMIF (dotted line), DM642 with 200 MHz EMIF (dashed line), and C6416 (full line).	63
5.5	First snapshot taken 40 seconds into the video recorded at NASA LaRC. The frame as captured by the camera is on the left and the real-time Retinex processed frame is on the right.	66
5.6	Second snapshot taken 6 minutes and 28 seconds into the video. Colors are nearly completely indeterminable and objects are difficult to distinguish in the unprocessed image. Colors and objects are still clear in the processed frame.	66
5.7	Third snapshot taken 14 minutes 28 seconds into the video. The only distinguishable object in the unprocessed frame is the tail-lights on the vehicle. Although noisy, the real-time Retinex processed image still clearly shows most of the major objects in the first snapshot including spheres, tree lines, and parked vehicles.	67
6.1	The EVS LWIR, SWIR, and visible-band cameras mounted to a baseplate, and the enclosure shell. Inaccurate bore-sighting can cause image registration problems.	69
6.2	EVS camera enclosure mounted forward-looking underneath the NASA 757.	70
6.3	The EVS acquires data during the entire flight but take-off and landing phases are critical. The simulated shaded area depicts the field of view (FOV) of the cameras.	71

6.4	Examples of the imagery generated by each camera in good weather conditions. The images from cameras must be registered, enhanced, fused and displayed to the pilot in real-time.	72
6.5	Image processing architecture and functions of the EVS. Analog NTSC camera outputs are currently processed. The SWIR data is used as the baseline for registration since it has the smallest field of view.	73
6.6	DM642 EVM, signal splitter boards, and power supply in flight box.	75
6.7	Flight box in flight pallet on NASA 757.	75
6.8	A frame from the EVS SWIR camera before processing. The faint vertical lines were part of the input image and probably caused by subsampling in the video distribution system.	78
6.9	A frame from the EVS LWIR camera before processing. The LWIR camera output is actually rotated 180° from what is shown.	79
6.10	SWIR frame after enhancement.	80
6.11	LWIR frame after enhancement and registration to the SWIR image.	80
6.12	Enhanced, registered and fused output image.	81
7.1	Data flow diagram of MSR tasks	89
A.1	Original SWIR	102
A.2	Original LWIR	102
A.3	Original Visible	102
A.4	Cropped SWIR	103
A.5	SS Reg. LWIR	103

A.6 SS Reg. visible	103
A.7 SWIR and SS Registered LWIR	104
A.8 SS Registered LWIR and visible	104
A.9 Repeated SWIR	105
A.10 MLR Reg. visible	105
A.11 SWIR and MLR Reg. visible	105
A.12 Repeated MLR Registered visible	106
A.13 MLR Reg. LWIR	106
A.14 MLR Reg. visible and LWIR	106
A.15 Orig. SWIR at Time 26:14:28	107
A.16 Orig. LWIR at Time 26:14:28	107
A.17 Orig. visible at Time 26:14:28	107
A.18 MLR Registered visible at Time 26:14:18	107
A.19 SWIR and MLR Registered visible	107
A.20 MLR Registered LWIR at Time 26:14:18	108
A.21 MLR Registered visible and LWIR	108
B.1 High-level block diagram of a typical FPGA Architecture	111

ABSTRACT

The field of digital image processing encompasses the study of algorithms applied to two-dimensional digital images, such as photographs, or three-dimensional signals, such as digital video. Digital image processing algorithms are generally divided into several distinct branches including image analysis, synthesis, segmentation, compression, restoration, and enhancement. One particular image enhancement algorithm that is rapidly gaining widespread acceptance as a near optimal solution for providing good visual representations of scenes is the Retinex.

The Retinex algorithm performs a non-linear transform that improves the brightness, contrast and sharpness of an image. It simultaneously provides dynamic range compression, color constancy, and color rendition. It has been successfully applied to still imagery captured from a wide variety of sources including medical radiometry, forensic investigations, and consumer photography. Many potential users require a real-time implementation of the algorithm. However, prior to this research effort, no real-time version of the algorithm had ever been achieved.

In this dissertation, we research and provide solutions to the issues associated with performing real-time Retinex image enhancement. We design, develop, test, and evaluate the algorithm and architecture optimizations that we developed to enable the implementation of the real-time Retinex specifically targeting specialized, embedded digital signal processors (DSPs). This includes optimization and mapping of the algorithm to different DSPs, and configuration of these architectures to support real-time processing.

First, we developed and implemented the single-scale monochrome Retinex on a Texas Instruments TMS320C6711 floating-point DSP and attained 21 frames per second (fps) performance. This design was then transferred to the faster TMS320C6713 floating-point DSP and ran at 28 fps. Then we modified our design for the fixed-point TMS320DM642 DSP and achieved an execution rate of 70 fps. Finally, we migrated this design to the fixed-point TMS320C6416 DSP. After making several additional optimizations and exploiting the enhanced architecture of the TMS320C6416, we achieved 108 fps and 20 fps performance for the single-scale, monochrome Retinex and three-scale, color Retinex, respectively. We also applied a version of our real-time Retinex in an Enhanced Vision System. This provides a general basis for using the algorithm in other applications.

REAL-TIME RETINEX IMAGE ENHANCEMENT: ALGORITHM AND ARCHITECTURE OPTIMIZATIONS

Chapter 1

Introduction

Digital image processing encompasses the research and application of signal processing techniques applied to two-dimensional digital images, or three-dimensional signals such as digital video. The field originates from the confluence of large-scale digital computation and the requirement to improve the imagery generated by the U.S. space program in the mid-1960's [20]. Over the last 40 years computation technologies have experienced phenomenal growth and digital image processing has benefited from this progress to become a tool that is used in a wide variety of applications. There are now several branches of digital image processing, each representing different aspects of the field. These branches include image analysis, segmentation, compression, synthesis, restoration, and enhancement [20, 30]. One particular image enhancement algorithm that is rapidly gaining wide-spread acceptance as a near optimal solution for providing good visual representations of scenes is the Retinex.

The Retinex performs a computationally intensive, non-linear spatial/spectral transform that synthesizes strong local contrast enhancement and color constancy [33]. It is used to improve the brightness, contrast and sharpness of an image. It has been successfully

applied to still imagery captured from a broad range of sources including aviation safety, medical radiometry, forensic investigations, military operations, homeland security, and consumer photography [103, 55]. It is offered in the commercially available software package PhotoFlair by TruView [99]. Several users require a real-time, embedded implementation of the Retinex, but prior to this research effort, no real-time version of the algorithm had ever been achieved. Real-time¹ is defined here as continuously capturing, processing and displaying 15–30, 256×256 sized images² (frames) per second. Embedded implies a system or component that is, in general, relatively small, inexpensive, and consumes very little power [19].

One reason that a real-time version of the Retinex had not been achieved is because the Retinex is inherently computationally intensive due to the large volume of data that must be stored, processed, and transferred between processor and memory. The algorithm also entails performing multiple, large convolutions and requires orthogonal data accesses that exacerbate the problem. Another reason is the inefficiency of most general-purpose computing platforms for real-time Retinex processing — as well as for many other digital image processing algorithms. Today’s general-purpose processors, such as 2.5 GHz Pentium 4s, possess sufficient computation power to provide reasonable processing rates for Retinex processing of small, still images. However, in general, they do not have the proper architecture, operating system, or development tools to effectively meet the time constraints required for real-time Retinex processing. In addition, many applications limit the processor selection to components that can be embedded into a system. Many general-purpose

¹A real-time system is one that satisfies explicit bounded response-time constraints to avoid failure [39].

²All image sizes, such as 256×256 , in this dissertation are expressed using 8-bit pixels.

processors consume too much power or are too expensive to be used for these types of applications.

Several specialized, high-performance hardware architectures and technologies are suitable for this task. Application specific integrated circuits (ASICs) [62] are one-of-a-kind custom devices targeted towards a specific task and provide excellent performance at the expense of long development times and high cost. Field programmable gate arrays (FPGAs) [58, 51] are an attractive alternative that offer relative ease of programming, high performance and reconfigurability to support custom applications. Digital signal processors (DSPs) [4] are inexpensive, easy to program -- usually in common high level languages such as C — and offer good performance. DSPs are optimized for processing signals in real-time and offer some limited flexibility in architecture configuration. Several other esoteric technologies, such as array processors, are also available [36, 35]. However, for quick, low cost development, DSPs are a suitable and sufficient design choice.

In this dissertation, we examine and provide solutions for the issues associated with performing real-time Retinex image enhancement. We design, develop, test and evaluate the algorithm and architecture optimizations required to enable the implementation of the real-time Retinex specifically targeted for specialized, embedded DSPs. This includes optimization and mapping of the algorithm to different DSPs and configuration of these architectures to support real-time processing. We also develop and apply a particular instance of our research efforts for the real-time Retinex into an Enhanced Vision System [98]. This provides a general basis for using the algorithm in other applications or missions.

First, we developed and implemented the single-scale monochrome Retinex executing on a Texas Instruments TMS320C6711 floating-point DSP and attained 21 frames per second

(fps) performance [24]. This design was later transferred to the slightly faster TMS320C6713 floating-point DSP and ran at 28 fps [25]. We then modified our design targeting the fixed-point TMS320DM642 DSP and initially achieved an execution rate of 34 fps [25]. Further refinements and optimizations improved our performance to nearly 70 fps. This design was implemented as part of an Enhanced Vision System (EVS) and demonstrated during EVS flight tests in August and September of 2005. Inputs from two single-band cameras were Retinex enhanced, registered, and fused. The system operated at over 34 fps. Finally, we migrated our design to a TMS320C6416 fixed-point DSP. After making several additional optimizations and exploiting the enhanced architecture of the TMS320C6416 we obtained 108 fps performance for the single-scale, single-band (monochrome) Retinex and 20 fps performance for the three-scale, three-band (color) Retinex.

Several different user communities will benefit from this enabling technology. The Aviation Safety Program Office at NASA LaRC will continue to support applying the real-time Retinex in future technology demonstrations on the NASA LaRC ARIES 757 (NASA 757) research aircraft. The Transportation Security Administration is interested in using the Retinex in applications to improve Homeland Security. The U.S. Army has provided funding to study using the real-time Retinex as part of a system to find improvised explosive devices (IEDs) from unmanned aerial vehicles (UAVs). The real-time Retinex also has been identified for potential use in future NASA space programs including lunar and planetary exploration missions and autonomous landing systems.

In Chapter 2 of this dissertation, we discuss the mathematics behind the Retinex algorithm. In Chapter 3 we give an overview of the architectures of our chosen DSP hardware. In Chapter 4 we describe our test environment, and the software tools used to develop,

implement and measure the performance of the real-time Retinex. Chapter 5 is the heart of this dissertation. In it we discuss the optimization techniques we developed and applied to achieve real-time Retinex performance. In Chapter 6 we describe the EVS, and discuss how particular instances of the real-time Retinex were used in this context. In Chapter 7 we discuss future Retinex research issues and their potential solutions. This includes discussions of distributing the core structures developed for the DSP platforms into a multiprocessor environment, and the algorithm and architecture modifications required to process larger format images. Finally, in Chapter 8 we give our conclusions to this research.

Chapter 2

Retinex Image Enhancement

The Retinex is a general-purpose image enhancement algorithm that is used to produce good visual representations of scenes. The algorithm is derived from the last version of Edward Land's Retinex model [37] of the innate ability of human vision to perceive vivid color and detail across widely varying lighting conditions. In addition, this perception is relatively independent of the spectral characteristics of the illuminant. Jobson, et al. extended and improved Land's Retinex into a general-purpose enhancement algorithm that simultaneously provides dynamic range compression, color constancy, and color and lightness rendition. The first version of their work, the single-scale Retinex (SSR), provided good performance, but traded-off dynamic range compression for color rendition [33]. They improved their design by using multiple scales (multi-scale) within the Retinex (MSR) to address this tradeoff, and additionally added a method of color restoration to improve color rendition when gray-world violations occur within an image [32]. Other methods, such as post-processing using a white balance technique [56] have also been added. These additions extend the potential utility of the Retinex, but they also increase the computational require-

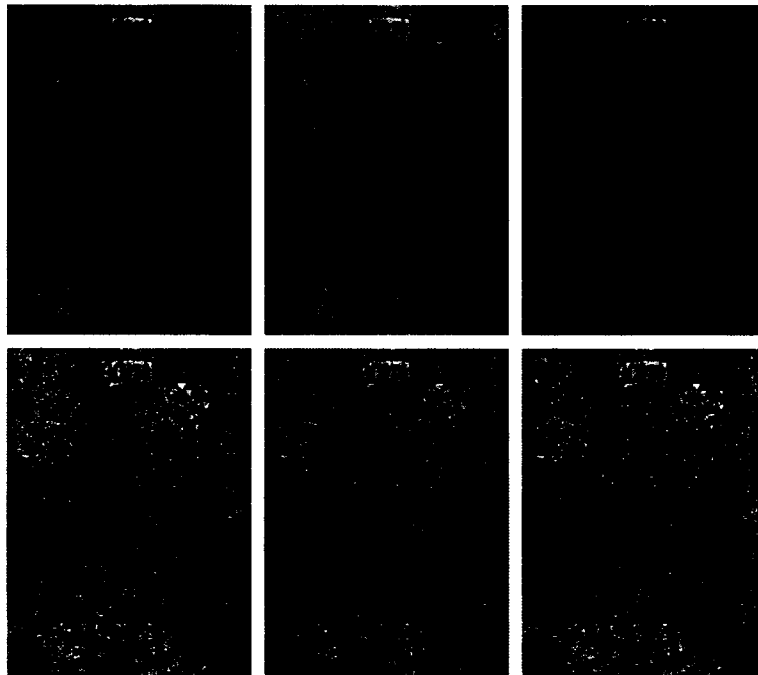


Figure 2.1: The top row of images from left to right have simulated tungsten, fluorescent, and sunlight illumination sources. The bottom row has the same images after Retinex processing. The effects of the different illumination sources is nearly completely removed.

ments of the algorithm. We concentrate on the SSR and MSR versions of the algorithm.

Figure 2.1 is an example that shows the color constancy property of the Retinex. The top row of images have simulated tungsten, fluorescent, and sunlight illumination sources from left to right respectively, and the bottom row is the image after Retinex enhancement. The Retinex processing has almost totally removed the effect of different illuminants on the scene. Figure 2.2 is a good visual illustration of the dynamic range compression property. Retinex processing of the image on the left dramatically brings out the details in the dark regions of the image without saturating the bright regions. Both of these examples are processed using the color version of the MSR. Figure 2.3 shows an example of monochrome SSR processing. The contrast and sharpness of the original is improved significantly.



Figure 2.2: Many image processing algorithms would either saturate the bright regions or clip the dark regions of the image on the left. The Retinex processed image on the right appears almost uniformly illuminated without exhibiting these effects.

The Retinex is a member of the class of center/surround functions which are similar to well known difference-of-Gaussian (DOG) functions [27, 54]. For the Retinex, the center is one pixel wide and its magnitude is the pixel value and the surround is a Gaussian. The single-scale Retinex is given by

$$R_i(x_1, x_2) = \log(I_i(x_1, x_2)) - \log(I_i(x_1, x_2) * F(x_1, x_2)), \quad i = 1, \dots, S \quad (2.1)$$

where I_i and R_i are the i th spectral band of the input and output image, respectively. For a grayscale image $S = 1$ and for a standard color image $S = 3$. The log is the natural logarithm function and “*” represents convolution. F is a Gaussian surround (or kernel) function defined by

$$F(x_1, x_2) = \kappa \exp[-(x_1^2 + x_2^2)/\sigma^2] \quad (2.2)$$

where σ controls the spatial extent of the surround, and $\kappa = 1/(\sum_{x_1} \sum_{x_2} F(x_1, x_2))$ is a

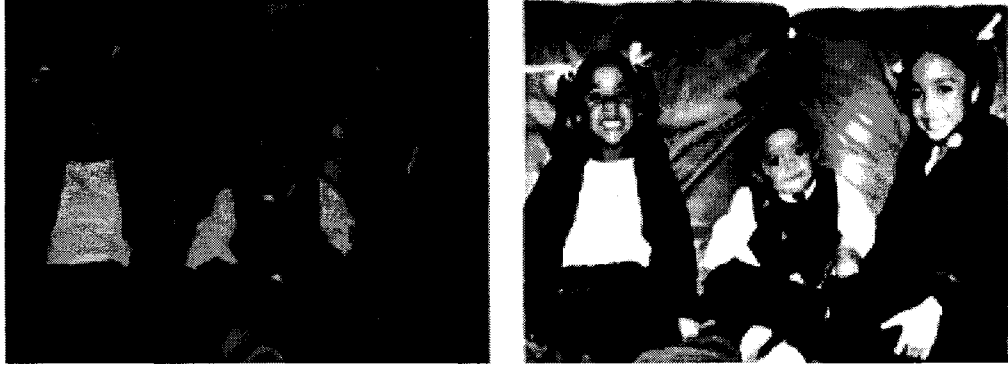


Figure 2.3: On the left is a low contrast, dimly lit grayscale digital image; on the right is the single-scale Retinex processed image — single-scale processing increases the contrast and sharpness.

normalization factor. Canonical gain, α , and offset, β , values are applied to convert the Retinex output into the user display domain, so the final form of the single-scale Retinex is

$$R_i(x_1, x_2) = \alpha \left(\log(I_i(x_1, x_2)) - \log(I_i(x_1, x_2) * F(x_1, x_2)) \right) - \beta, \quad i = 1, \dots, S \quad (2.3)$$

Values for α , β , and σ are application dependent and determined empirically. For example, in normal room light conditions values of 200, -120, 80 respectively produce good results.

The multi-scale Retinex is defined as the weighted sum of K SSR outputs, where K is the number of scales. Thus the MSR is given by

$$R_i(x_1, x_2) = \sum_{k=1}^K W_k \left(\log(I_i(x_1, x_2)) - \log(I_i(x_1, x_2) * F_k(x_1, x_2)) \right) \quad (2.4)$$

where the F_k are now defined as

$$F_k(x_1, x_2) = \kappa_k \exp[-(x_1^2 + x_2^2)/\sigma_k^2]. \quad (2.5)$$

The W_k are the weighting factors and the κ_k are the normalization factors associated with each scale. Jobson et al. [32] have shown, empirically, that three scales with reasonable local

to global coverage, and equal weights provide good performance for most images. Again a canonical gain α , and offset, β , are applied thus the final form of the MSR is

$$R_i(x_1, x_2) = \alpha \sum_{k=1}^K W_k (\log(I_i(x_1, x_2)) - \log(I_i(x_1, x_2) * F_k(x_1, x_2))) - \beta. \quad (2.6)$$

The derivation of the computational complexity of the Retinex is straightforward. Assume that the input image dimension size is $N \times N$, the extent of the surround, F , is $M \times M$, circular convolution is performed in the spatial domain, and ignore the operations involving α , β , W_k and the computations required to generate F_k . We show in Section 5.1.2, that these are all valid assumptions. Then for the single-scale monochrome Retinex, there are M^2 multiplies and $M^2 - 1$ additions for every pixel. There are also $2N^2$ logarithm operations — two logarithms for each pixel, and N^2 subtractions. Thus, the running time of the algorithm is driven by the convolution operation and the complexity is $O(N^2M^2)$. As the extent of F approaches the size of the image, i.e. $M \rightarrow N$, the complexity becomes $O(N^4)$. For the one scale, multi-spectral case, the monochrome algorithm is performed S times, once for each spectral band. The complexity remains the same, $O(N^2M^2)$. For the multi-scale, multi-spectral case, the convolution and the other arithmetic operations are performed K times, once for each scale. This is subsequently repeated S times, once for each spectral band. Additionally, as discussed in Section 5.4, for any multi-spectral case, functions may be required to divide the spectrum into its individual component parts for processing, and to combine the processed components back together again. However, the complexity still remains the same — $O(N^2M^2)$. Methods to reduce the running time of the algorithm are discussed in Chapter 5.

Chapter 3

Digital Signal Processors

For our research we have selected four state-of-the-art Texas Instruments (TI) DSPs for implementation and performance evaluation of the real-time Retinex (RTR). TI processors were chosen because of their flexible and powerful architecture, good support tools, availability of the DSPs to the researchers, low cost of evaluation boards, and our past familiarity with using TI processors. Many other DSPs, such as Analog Devices SHARC processors, would also provide reasonable hardware platforms for implementation. All of the TI DSPs that were chosen are based on an advanced very-long-instruction-word (VLIW) [71] architecture. This type of architecture achieves high performance by exploiting instruction-level parallelism. Multiple execution units operate in parallel to execute multiple instructions during a single clock cycle. Our four target DSPs are the TMS320C6711, TMS320C6713, TMS320DM642, and TMS320C6416. In this chapter we discuss the relevant details of each of these processors.

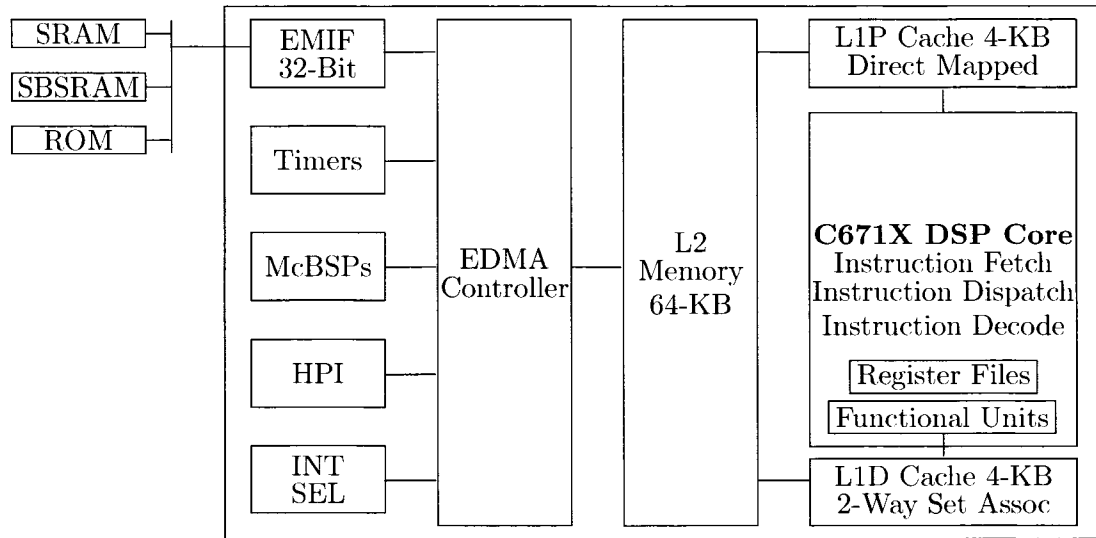


Figure 3.1: Primary DSP components include the CPU, L1 Data Cache, L1 Program Cache, L2 memory (SRAM/Cache) and EDMA Controller.

3.1 TMS320C6711

Our first target, the TMS320C6711B (C6711) DSP, is a 32-bit floating point processor that offers up to 1200 millions instructions per second (MIPS)/900 million floating point operations per second (MFLOPs) performance at a clock rate of 150 MHz (6.67 ns cycle time) [73]. As shown in Figure 3.1 the processor is divided into three main components: the CPU (or core), memory, and peripherals.

The CPU has eight independent functional units and a 256-bit wide instruction word that allows up to eight 32-bit instructions to be supplied to the units during every clock cycle. The functional units are mapped into two sets where each set contains four units and a register file. In total the eight functional units provide four fixed/floating point arithmetic logical units (ALUs), two fixed-point ALUs, and two fixed/floating-point multipliers. Two multiply-and-accumulate (MACs) per cycle can be performed for a total of up to 300 Million MACs (MMACs) per second. Each of the two register files contains sixteen 32-bit registers

for a total of 32 general-purpose registers. Six of the functional units have access to the register file on the opposite side via a cross path. Like a MIPS processor, the CPU uses a load/store architecture, where all instructions operate on registers. There are dual 64-bit load data paths and dual 32-bit store data paths.

The DSP has a two-level memory architecture for both program and data [88]. Figure 3.2 is a general outline of the architecture. This hierarchical architecture is used to reduce the average memory access time by exploiting the temporal or spatial locality of data [87]. The Level 1 data cache (L1D) is a 32-Kbit 2-way set associative cache that services data accesses from the CPU. It has a 32-Byte line size and 64 sets. The L1D is implemented with a single bank of dual-ported 64-bit memory and can service up to two data accesses from the CPU on every cycle. The L1D is a read-allocate cache, but does not write-allocate¹. A 32-bit by 4-entry write buffer between the L1D and the L2 memories is used to capture write misses. The Level 1 program cache (L1P) is a 32-Kbit direct-mapped, read-allocate cache that services program fetches from the CPU. It has a 64-Byte line size and 64 sets.

The Level 2 (L2) memory space is 64-KBytes that can be configured as all SRAM, all cache, or combinations of the two in 16-KByte increments. This memory services requests from the L1P, L1D, enhanced direct memory access (EDMA), or internal cache operations, with request priority from highest to lowest as listed. It is divided into four 64-bit banks that operate at the CPU's clock rate, 150 MHz, but pipelines accesses over two cycles. Any portion of L2 configured as cache (L2 Cache) is organized as 128 sets with 128-Byte line size. The associativity varies from 1-way for when the cache capacity is 16-KBytes, up

¹A read/write-allocate cache allocates space (i.e. selects a location in the cache) on a read/write miss according to the cache allocation policy.

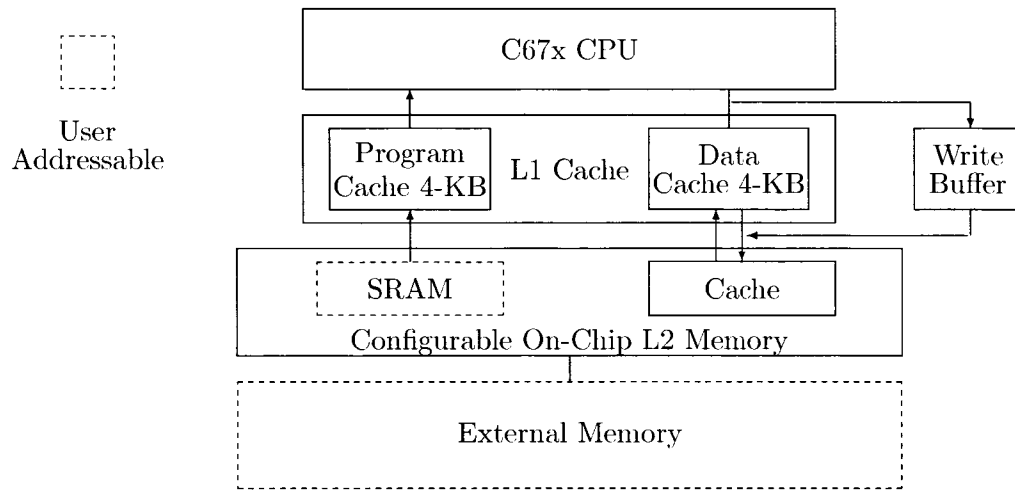


Figure 3.2: General outline of 2-level internal memory architecture of C67x processors. The dashed boxes are user addressable memory.

to 4-way at 64-KBytes. The different configuration modes are shown in Figure 3.3. The operation of L2 Cache is similar to that of both the L1P and L1D caches. On a cache hit the L2 cache services the request directly. The L2 Cache is a writeback² cache so external memory is not updated until the line is either evicted or written back using cache control registers. Unlike the L1D, the L2 Cache is read-allocate and write-allocate. A least-recently used policy (LRU) is used for line selection.

Several peripherals are located within the processor. There is a multichannel EDMA controller that supports up to 16 channels of data transfers. There is a host port interface (HPI) that allows a host processor to directly address the CPU's memory space. There is also a 32-bit external memory interface (EMIF) that provides an interface to external devices such as synchronous dynamic random access memory (SDRAM) and read-only memories (ROMs) [78].

²Writeback is the process of writing data that has been modified from a valid, but now dirty cache line to lower-level memory. Write hits to a writeback cache are not immediately forwarded to lower-level memory.

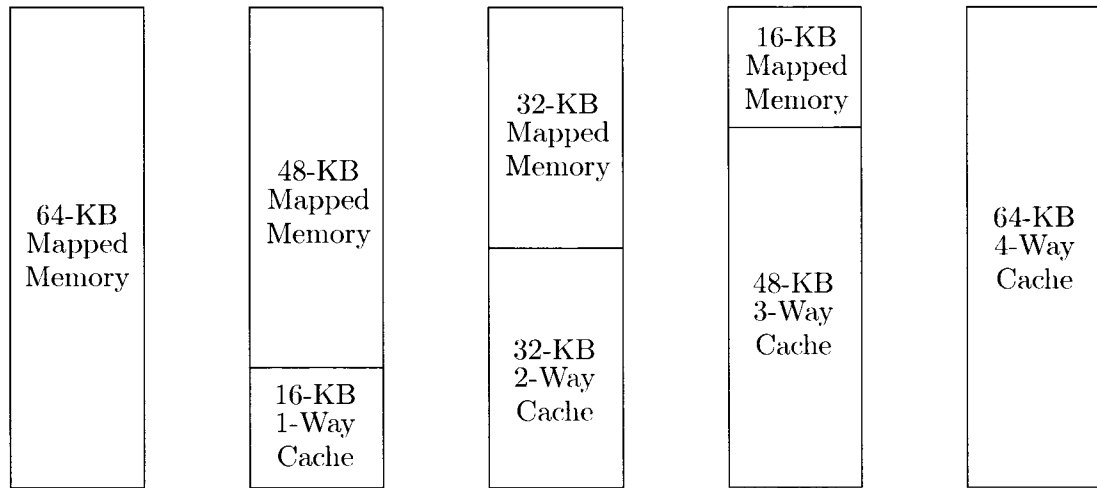


Figure 3.3: Configuration modes for the C6711 L2 memory.

3.2 TMS320C6713

Our second target, the TMS320C6713 (C6713), is a 32-bit floating point processor that performs up to 1800 MIPS/1350 MFLOPS at a clock rate of 225 MHz (4.4 ns instruction cycle time) [84]. The architecture of the C6713 is very similar to the C6711, and code operating on one device directly ports over to the other [92]. The most relevant differences in the two devices are listed below.

- The C6713 operates at 225 MHz while the C6711 only operates at 150 MHz.
- The C6713 has a larger internal memory. The L1 caches are the same, but the C6713 has an additional 192-KBytes of SRAM in L2 that only functions as mapped memory.
- The C6713 has a software-configurable Phase-Lock Loop (PLL) controller that can be used to select different clock frequencies for the DSP core, peripherals and the EMIF [94]. Speeding up EMIF transfers can enable faster throughput.

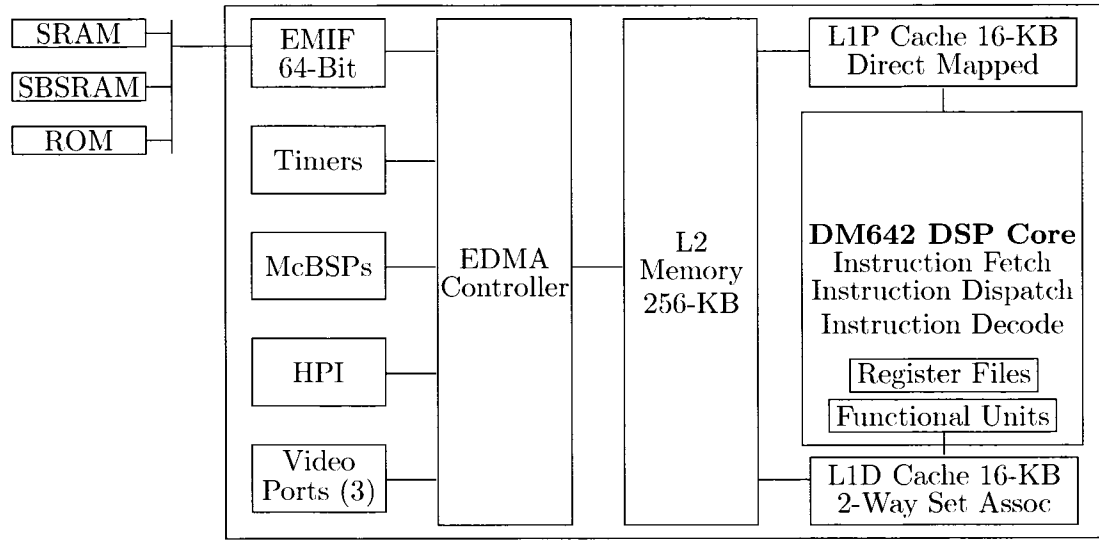


Figure 3.4: Block diagram of primary DM642 components. The DM642 has special instruction extensions to accelerate video applications.

3.3 TMS320DM642

Our third target is the TMS320DM642 (DM642). The DM642 is a 32-bit fixed-point processor that performs up to 4800 MIPS at a clock rate of 600 MHz (1.67 ns instruction cycle time) [86]. A block diagram of the processor is shown in Figure 3.4. The DM642 also has eight independent functional units consisting of six ALUs and two enhanced multipliers. In addition to standard multiplies, the multiply units include hardware that can perform bit-count, rotates, and bidirectional variable shifts. Four 32-bit MACs per cycle can be performed for a total of 2400 MMACs per second, or eight 8-bit MACs per cycle for a total of 4800 MMACS. There are new instruction extensions to accelerate video and imaging applications, and to improve the parallelism of the architecture [79]. This includes support for packed 8-bit and 64-bit data types, and instructions that perform non-aligned loads and stores of words or double words.

The DM642 also has a two-level cache [95]. The L1P is a 16-KByte direct-mapped

cache with 32-Byte line size and 512 sets. Multiple cache misses are pipelined. The L1D is 16-KBytes deep and is 2-way set associative with a 64-Byte line size and 128 sets. It is implemented as eight 32-bit wide banks of single-ported memory, as opposed to the single bank of dual-ported memory of the C671X devices. Each single-ported bank allows only one access per cycle. The L1D is a read-allocate only cache where new lines are allocated for L1D read misses but not write misses. The L1D implements a LRU line allocation policy for read misses and pipelines multiple misses. A 64-bit by 4-entry write buffer between L1D and L2 memory captures data from write misses. This buffer is an enhanced version of the one in the C671X in that the L2 can process a new request from the write buffer every cycle, as opposed to every 2 cycles on the C671X, provided that the L2 bank is not busy. Additionally, the DM642 write buffer allows merging of write requests, thus effectively increasing the write buffer capacity, reducing the stall penalty, and reducing the overall number of write operations the L2 must process.

The L2 memory is 256-KBytes that can be configured as local SRAM, cache or combinations of the two. This memory services cache misses from the L1P, the L1D, the EDMA controller and internal cache operations with request priority from highest to lowest as listed. It is divided into eight 64-bit banks that operate at the CPU's clock rate, 600 MHz, but pipelines accesses over two cycles. Four L2 Cache configuration modes are supported: 32-KByte capacity organized as 64 sets, 64-KByte capacity as 128 sets, 128-KByte capacity as 256 sets, and 256-KByte capacity as 512 sets. L2 Cache is always 4-way set associative with 128-Byte line sizes and operates as a write-back cache. A cache line is allocated for both read and write misses, and a LRU policy is used for line selection.

The DM642 also has many of the same peripherals as the C671X devices with several

extensions and additions including a 64-bit EMIF and three configurable video port peripherals [89]. The video ports provide a glue-less interface to common video decoder and encoder devices. Each video port can be configured for either video capture or display, and each port supports up to two channels with a 5120-Byte buffer that is shared between the two channels.

3.4 TMS320C6416

Our fourth target is the TMS320C6416 (C6416). The C6416 is a 32-bit fixed-point processor that performs up to 8000 MIPS at a clock rate of 1000 MHz (1 ns instruction cycle time) [97]. A block diagram of the processor is shown in Figure 3.5. The C6416 has eight independent functional units consisting of six ALUs and two enhanced multipliers capable of performing four 16-bit \times 16-bit multiplies every clock cycle with add/subtract operations. Four 32-bit MACs per cycle can be performed for a total of 4000 MMACs per second, or eight 8-bit MACs per cycle for a total of 8000 MMACS. . The C6416 also includes support for packed 8-bit and 64-bit data types, and allows for non-aligned loads and stores of words/double words [79]. There are two register files, each containing 32, 32-bit registers for a total of 64 general-purpose registers. All eight of the functional units have access to the opposite register file and the dual load and store data paths are 64-bits wide.

The C6416 also has a two-level cache [97]. The L1P and L1D are the same size and operate the same as the respective memories on the DM642. The L2 memory has been increased to 1024-KBytes and can be configured as all mapped memory or combinations of cache (up to 256-KBytes) and mapped memory. Any portion of L2 memory partitioned

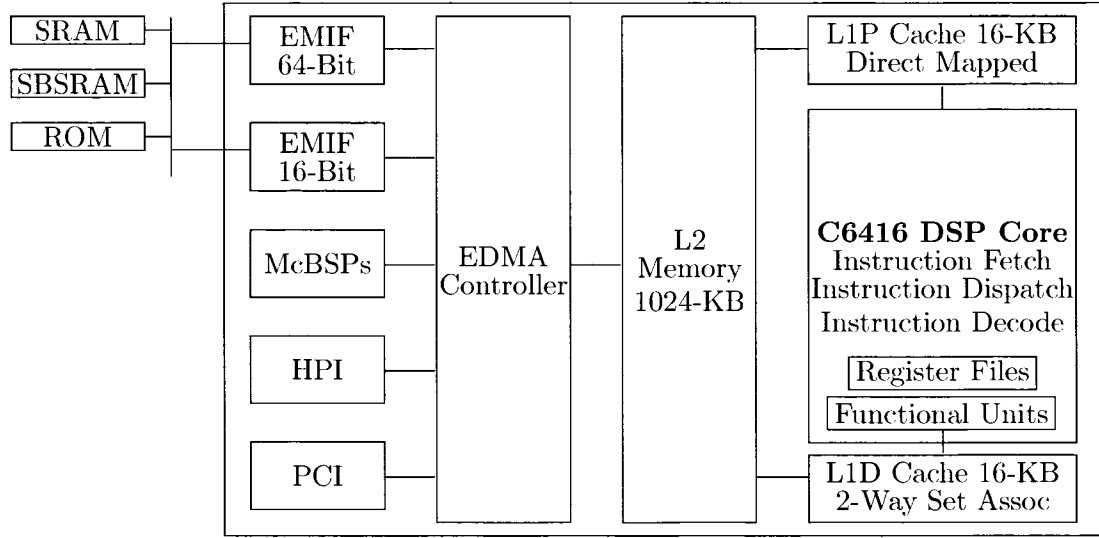


Figure 3.5: Block diagram of primary C6416 Components. Note the larger L2 memory and 64-bit EMIF bus.

as cache has the same modes as on the DM642. The C6416 has two EMIFs: one 64-Bits wide and one 16-Bits wide. The total external addressable memory space of 1280-MBytes.

Table 3.1 summarizes the pertinent parameters of the DSPs.

DSP	Type	Frequency (MHz)	L1 (K-Bytes)	L2 (K-Bytes)	EMIF (Width)	EMIF Clk (MHz)
C6711	Floating-pt	150	8	64	1 32-bit	100
C6713	Floating-pt	225	8	256	1 32-bit	90
DM642	Fixed-pt	720	32	256	1 64-bit	133
C6416	Fixed-pt	1000	32	1024	1 32-bit 1 64-bit	100 100

Table 3.1: DSP Specifications

Chapter 4

Test Environment

We now describe the platforms that support each DSP and the general hardware and software test environment. This environment will be used to test, analyze and evaluate our optimization techniques discussed in Chapter 5.

4.1 DSP Evaluation Modules

Each DSP is embedded on a different printed circuit board for test and evaluation. The circuit boards are called EVMs (evaluation modules). Figure 4.1 shows the EVM for the DM642. The other EVMs look similar to this. As can be seen in the figure, each EVM has several components and interfaces to support the associated DSP. We will briefly describe the EVMs for each of our selected DSPs only defining the parts relevant to our discussion. We will then describe the tools used for software development, optimization, and performance analysis.

The C6711 EVM has 16-MBytes of SDRAM clocked at 100 MHz that is used as external memory for the chip. There are 128-KBytes of flash memory which is usually used to

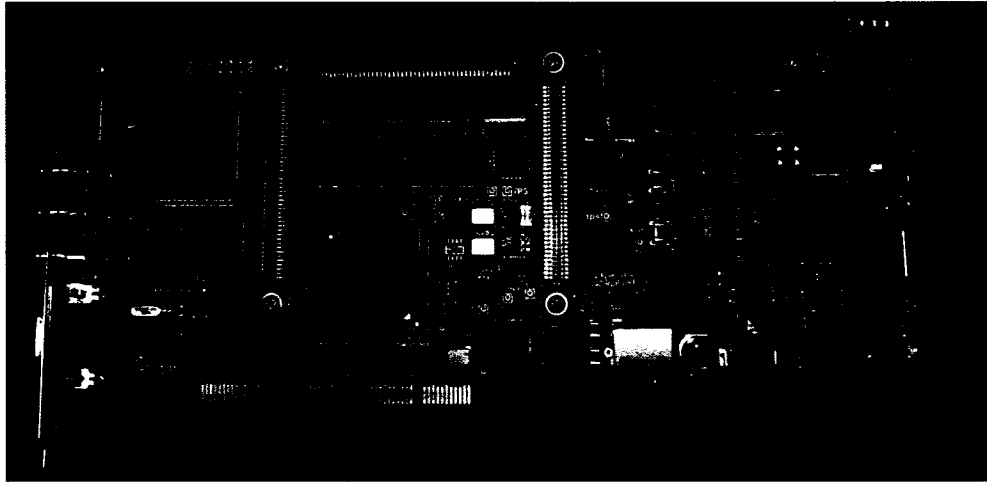


Figure 4.1: Picture of DM642 EVM board. Numerous components are on the EVM circuit board to support testing the DSP for a wide variety of applications. We primarily use the peripherals associated with video capture and display.

hold application code and parameters when power is disconnected from the board. Communication to a host PC — primarily for downloading code and gathering statistics — is through a parallel port. An embedded Joint Test Action Group (JTAG) controller is used for emulation and debugging [28]. The board also has an expansion connector to support adding additional memory, peripherals, or daughter-cards [77].

The C6713 EVM has 8-MBytes of SDRAM clocked at a default rate of 90 MHz and 512-KBytes of flash memory. Communication to a host PC is performed through a Universal Serial Bus (USB) port. An embedded USB JTAG controller is provided for debugging [66]. The EVM also has an Intel LXT971 Ethernet port for data transfers to an external device.

The DM642 EVM has 32-MBytes of SDRAM clocked at 133 MHz, 4-MBytes of flash memory, an Intel LXT971 Ethernet interface, and a standard JTAG connector for external emulation [67]. The C6416 EVM has 256-MBytes of SDRAM on the 64-bit EMIF bus and 8-MBytes on the 32-bit wide EMIF bus. Both busses are clocked at 100 MHz. The

board also has 4-MBytes of flash memory, and a dedicated JTAG connector for external emulation [3].

4.1.1 Video Capture and Display for C6711 and C6713 EVMs

For the C6711 and C6713 EVMs, video capture, display, and data formatting are performed by an imaging daughter-card (IDC) [76] that connects to each board's expansion connectors. The main components of the IDC are a TI TVP5022 digital video decoder chip [74], a TI TVP3026 RAMDAC digital video encoder chip [70], a Xilinx FPGA for control, buffer management and interface logic, and 2-MBytes of SDRAM for capture frame memory. The IDC also has a female Radio Corporation of America (RCA) connector that is used to receive video, and a standard 15-pin female video graphics array (VGA) connector that is used to supply red, green, blue (RGB) [69] video output to a monitor.

Figure 4.2 is a block diagram of the video capture subsystem [72]. A video input signal from an NTSC (or Phase Alternating Line (PAL)) source is digitized by the TVP5022 decoder chip into a standard $Y'C_BC_R$ 4:2:2 format¹. The $Y'C_BC_R$ is a color space used to represent digital component video where color is represented by a luma component (Y'), and two chroma components (C_B and C_R). The 4:2:2 notation² designates the ratio of Y' , C_B and C_R signals where C_B and C_R are co-sited and subsampled at half the horizontal resolution of Y' [53].

¹The ITR-R BT.601 standard defines the $Y'C_BC_R$ color space and the 4:2:2 sampling organization and resolutions. The BT.656 standard defines the serial and parallel interfaces for transmitting $Y'C_BC_R$ 4:2:2 digital video [29, 100].

²The number 4 originates from a multiplier of the BT.601 chosen baseline frequency of 3.375 MHz and corresponds to a sampling rate of 13.5 MHz, a standard frequency for digitizing NTSC or PAL; similarly the 2s correspond to 6.75 MHz [100]. Other common subsampling ratios include 4:4:4, 4:1:1 and 4:2:0 (where the chroma components are sited interstitially)

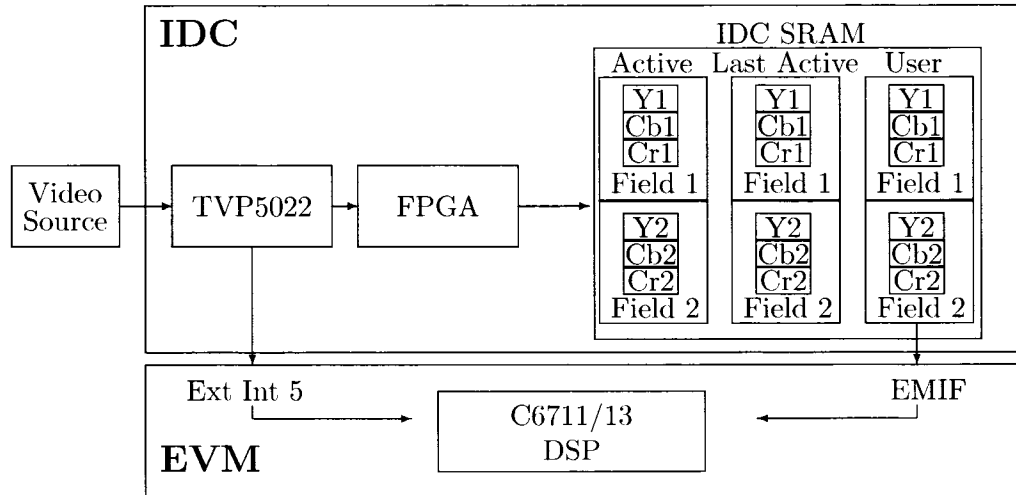


Figure 4.2: IDC video capture subsystem

The 8-bit wide $Y'CbCr$ pixel stream — interleaved as C_B , Y' , C_R , Y' , \dots , is fed into the FPGA. The FPGA separates and stores the stream into capture frame memory buffers as two separate fields (odd and even) in three separate blocks (Y' , C_B , C_R) as shown in Figure 4.2. The TVP5022 chip also controls all video input timing including a vertical synchronization signal that generates a CPU interrupt once per frame, and a blanking signal that indicates the presence of data on the pixel bus to the FPGA.

The capture frame memory buffers are memory-mapped into the DSP address space as read-only and are accessed via the EMIF. A triple buffering scheme is used to allow an application to obtain a new buffer of the most recently captured data without waiting. The “active” buffer is currently receiving data from the TVP5022. The “last active” buffer is the last buffer that was filled by the TVP5022. The “user” buffer is owned and read by the user application. If the application can maintain a full 30 fps processing rate, the buffers are physically walked through in a circular sequence by the FPGA and user application. If the user application attempts to access the buffers faster than 30 Hz, then duplicate frames will

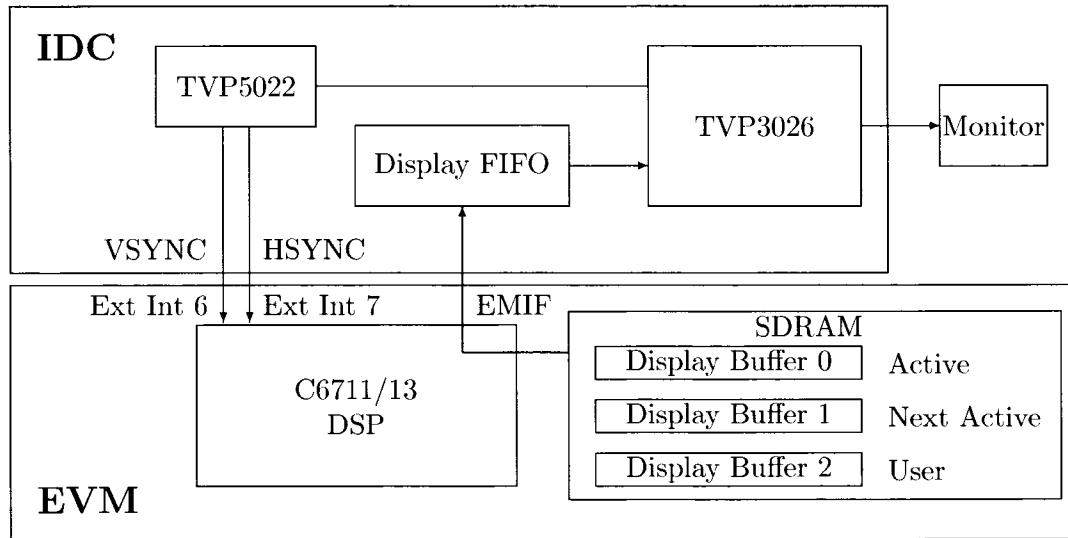


Figure 4.3: IDC video display subsystem

be returned. If the application executes slower, then captured frames will be overwritten.

Figure 4.3 is a block diagram of the video display subsystem [72]. Video display is limited to a max size of 800×600 pixels with 8-bits per pixel for grayscale or 16-bits per pixel for RGB 565 color³. A total output frame display buffer size of 2.88-MBytes ($800 \times 600 \times 16$ -bits for 3 buffers) is allocated and linked into the DSP's external memory space. Timing signals for video readout include a vertical synchronization (VSYNC) signal and a horizontal synchronization (HSYNC) signal. The VSYNC signal triggers a CPU interrupt and the associated interrupt service routine posts a display semaphore which is used to wait for new frames. The HSYNC signal triggers an EDMA event to copy one line of display data from the display buffer to the IDC display first-in-first-out (FIFO) buffer. The TVP3026 RAMDAC chip then transmits this line to the output port.

Analogous to the video capture system, a triple buffering scheme is used for data transfers. The “user” buffer is owned by the user application. The “next active” buffer will be

³RGB 565 represents color values using 5-bits for red, 6-bits for blue and 5-bits for green

returned on the next buffer request. The “active” buffer is being used for EDMA transfers. If the application attempts to access buffers too fast, frames will be dropped. If access is too slow, frames will be displayed repeatedly.

4.1.2 Video Capture and Display for DM642 and C6416 EVMs

The DM642 has three on-chip video ports. On the EVM two of the ports are configured as capture ports (video ports 0 and 1) and one is configured as a display port (video port 2). The capture ports interface to TI TVP5146 [96] and TVP 5150A [91] video decoders. The TVP5146 supports composite⁴ or Y/C format⁵ inputs, and the TVP5150A supports composite inputs only on the EVM. The output of the display port is routed through an FPGA (for functions such as on-screen display or overlays) to a Phillips SAA7105 video encoder. The SAA7105 drives either NTSC/PAL composite video, S-video, RGB, or high-definition component video. Figure 4.4 is a block diagram of the system. Analog input video is digitized into planar $Y'C_BC_R$ 4:2:2 component video and buffered in external memory similar to the method used for the IDC.

A block diagram of the C6416 EVM is shown in Figure 4.5. Analog video is digitized by a Conexant BT835 decoder into a $Y'C_BC_R$ 4:2:2 format and stored by the FPGA into the capture FIFO buffer. Instead of being written in planar form as on the C6711 EVM, the captured data is stored in $C_R, Y', C_B \dots$ interleaved order. The FIFO is memory-mapped into the address space of the DSP and accessed via the EMIF. Similarly, output data to be displayed is stored in $Y'C_BC_R$ 4:2:2 format and written using a EDMA channel into the

⁴Composite video combines luma, chroma and sync signals into a single waveform carried on a single wire pair.

⁵Y/C has the luma and chroma components carried on separate signal wire pairs to reduce signal crosstalk. Y/C is often incorrectly referred to as S-video, a magnetic tape modulation format.

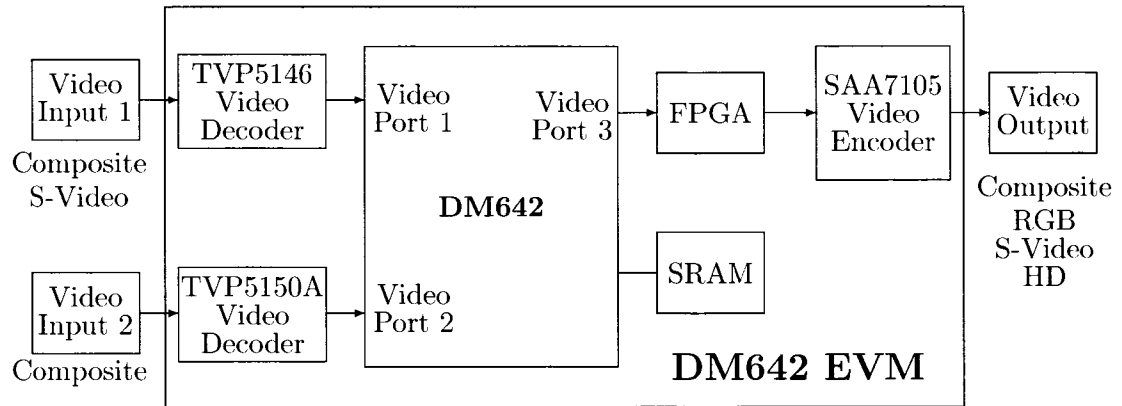


Figure 4.4: DM642 EVM block diagram

display FIFO by the DSP. The pixel stream is then transferred to a Conexant BT864 for digital-to-analog conversion (DAC) and NTSC/PAL encoding.

4.2 Development Tools

Several software development tools are used on all of the EVMs, including a C-compiler, assembly optimizer, and a debugger for visibility into source code execution. These tools are incorporated into TI's Code Composer Studio (CCS). Other rapid prototyping software tools used include a chip support library (CSL) [81] to configure and control on-chip peripherals, an image data manager (for the IDC) for DMA abstraction, and a C-callable DSP library (DSPLib) [90] that contains a collection of highly optimized functions such as the well-known Fast Fourier Transforms (FFT) [7, 49, 64]. A scalable real-time operating system (OS) kernel called DSP/BIOS (basic input output system) is used to provide preemptive multi-threading, hardware abstraction and real-time analysis [80].

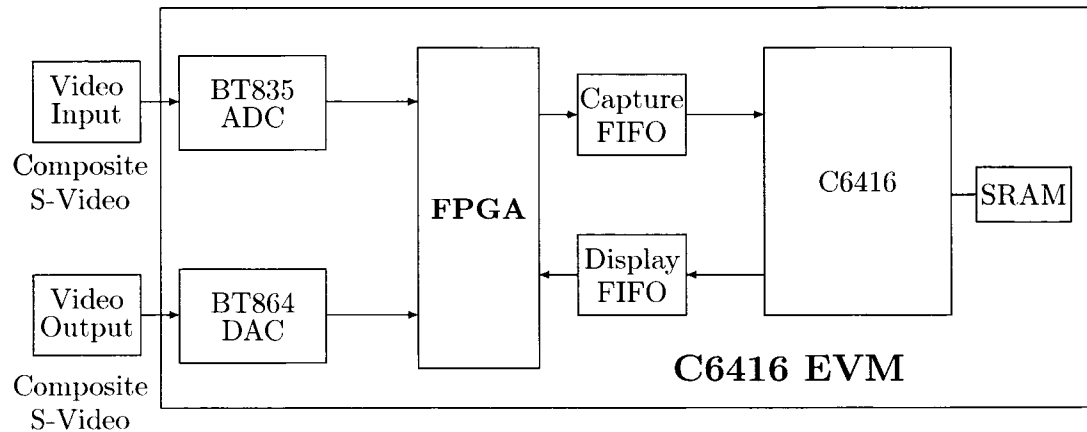


Figure 4.5: C6416 EVM block diagram

Compiler options are used to control speculative loading, auto in-lining thresholds, data alignment/placement information, and advanced loop optimizations [82]. Significant performance improvements can be gained by using target-specific instructions called intrinsics [93]. Intrinsics are special functions that allow certain assembly statements to be easily embedded in application code. For example to find the maximum value of two variables $x1$ and $x2$ we simply use the optimized in-line intrinsic function call for `max2` — `max2(x1, x2)`.

4.3 Test-Bed Components and Operation

A test-bed is used to implement and analyze the real-time Retinex algorithm and to support testing the algorithm within the context of the EVS for our case study. The baseline test-bed is composed of

- a standard NTSC video source (for example a video camera, DVD player or VCR),
- a monitor that accepts a composite video input to display the processed output,

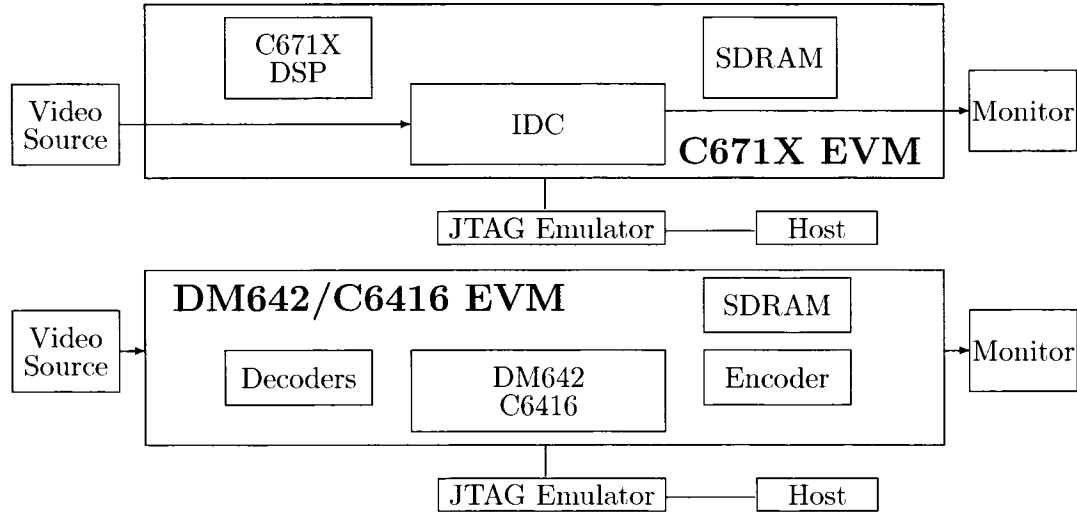


Figure 4.6: Block diagram of the test-bed — the Host PC only provides setup information to the EVM; after initiation, the DSP executes independently.

- a host personal computer (PC) running CCS for code development and analysis,
- a JTAG emulator for communication and debugging, and
- the target DSP on an EVM as discussed in Section 4.1.

Figure 4.6 shows general outlines of the test-bed using the C6711 and C6713 EVMs with IDCs, and the DM642 and C6416 EVMs. The host PC is not part of the image processing chain.

General operation of the test-bed is as follows. C code to perform the Retinex is written on the PC using the CCS software. This code is compiled, assembled and linked into a common object file format (COFF) and is downloaded into the DSP on the EVM. Execution of the algorithm is then triggered from the PC. From this point on, the EVM operates totally independent of the PC. The functions for performance analysis are (1) video frames are captured from the source, (2) a 256×256 pixel sized portion of the captured frame buffer is Retinex processed, and (3) the output product displayed.

4.4 Performance Analysis

The execution time of the Retinex is measured by using the real-time analysis tools within DSP/BIOS. These tools are composed of instrumentation code that is integrated into the target application. The code is executed at run time, and the events of interest are stored in memory on the target. This information is transferred to the host PC for display, further processing, or post-execution analysis. All instrumentation operations have fixed, short execution times and communication between the target and host is performed in the background using a low priority idle thread thus minimizing the impact on performance and program behavior.

The instrumentation modules can be called explicitly by the application through application programmer interfaces (API)s or implicitly through the calls used internally by DSP/BIOS[80]. Explicit instrumentation API modules include a statistics (STS) object manager and a trace (TRC) manager. STS objects store statistics about data variables or system performance including capturing count, maximum, total, and average values in real-time. The TRC module provides a means to enable or disable data acquisition in real-time through querying a set of bits.

Implicit instrumentation is built into DSP/BIOS and allows the user to display several values including CPU loading. CPU loading is defined as the percentage of instruction cycles that the CPU spends performing application related work -- running interrupts, tasks, periodic functions, performing I/O to the host, or running any other user routine. For the remaining time, the CPU is considered idle. CPU load is expressed by

$$CPUload = (c_w / (c_w + c_i)) \times 100 \quad (4.1)$$

where c_w and c_i are work and idle instruction cycles, respectively. CPU loading can be viewed graphically in a window with continuous updates if there are enough idle cycles to transfer this statistic to the host. Otherwise the values can be obtained after halting the target and retrieving the stored loading values.

4.5 Real-time Parameter Updates

A useful capability to test the Retinex algorithm is to be able to update parameters in real-time. TI provides a mechanism to interact with an application in real-time called real-time data exchange (RTDX)[80]. RTDX plug-ins provide a means to transfer data between a host computer and DSP devices via the JTAG interface with minimal interference with the target application. A small RTDX library runs on the target DSP while another runs on the host. An application executing on the target makes function calls to the RTDX target library's API to send or receive data. The host library, working within CCS, provides a component object model (COM)⁶ API for communication. Any object linking and embedding (OLE)⁷ automation client on the host can be used for display or analysis. We developed our own OLE client using Visual Basic to update Retinex parameters (α), offset (β), and the standard deviation of the Gaussian surround (σ).

⁶COM is a Microsoft developed technology that allows communication between software components.

⁷OLE is a Microsoft developed standard enables the creation of an object in one application that can be linked or embedded in a second application.

4.6 Retinex Task Within DSP/BIOS

Our code for the Retinex is written to execute as a task within the DSP/BIOS environment. This allows explicit use of the real-time analysis tools. In general, two tasks, “main” and “video processing” are scheduled. First, “main” performs a few initializations, such as setting up the chip support library, configuring the cache, and opening up an EDMA channel, and then returns. The “video processing” task is then set to run automatically by the DSP/BIOS scheduler. The video processing task consists of the following steps:

- set up several video parameters such as capture and display frame sizes,
- receive a frame from the capture frame buffer,
- call (and waits on) the Retinex processing function,
- display the Retinex output and optionally displays the unprocessed frame,
- exchange capture and display buffers, and then returns to read another frame.

STS objects are coded within the “video processing” task to determine the overall execution time of the Retinex processing function. Several STS objects are also placed within the Retinex processing function to determine internal performance characteristics. This helps to isolate the primary time consumers or “tall-poles” within the algorithm. STS API calls to set the time on an STS object, and then to check the change in time after execution of some portion of code requires approximately 18 and 21 instructions respectively. These values can be removed for a more accurate measure of performance.

Chapter 5

Optimizations and Performance

Results

We now describe the optimization techniques we developed and applied to implement the real-time Retinex. This discussion is the core of our research. Our discussion will focus on the major algorithm and architecture optimizations that significantly improved performance. Additionally, each optimization was developed under the basis that it would not cause any perceptible loss in image quality.

Our baseline algorithm and architecture targets are the single-scale monochrome version of the Retinex (SSMR) and the C6711 DSP on the C6711 EVM in our test-bed. The SSMR is the simplest form of the Retinex and the C6711 has the lowest performance of the processors in this study. However both allow us to establish our core algorithm and architecture techniques and provide a basis for future optimizations, extensions, and adaptation to other platforms. One change in the architecture at this point is to configure the L2 memory as 32-KBytes of cache and 32-KBytes of SRAM. The 32-KBytes of SRAM

are sufficient to store all the required variables in our first implementation.

5.1 Single-Scale Monochrome Retinex Optimizations

5.1.1 Apply Convolution Equivalence

A fundamental component of the Retinex computation is to convolve the input image with a Gaussian kernel. Good single-scale Retinex renditions are obtained with a large kernel ($\sigma > 80$), so performing this operation in the spatial domain is extremely time consuming. The first, and most obvious, optimization then is to use the well-known equivalence between convolution in the spatial domain and multiplication in the spatial-frequency domain [7, 20]

$$f(x, y) * g(x, y) \Leftrightarrow F(\mu, \nu)G(\mu, \nu) \quad (5.1)$$

where F and G are the spatial frequency domain representations of f and g respectively. We apply this concept to convolve an input image with a Gaussian kernel by employing the 2-dimensional $M \times N$ forward and inverse Discrete Fourier Transforms (DFTs) [20] defined by

$$\hat{F}(\mu, \nu) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \exp[-j2\pi(\mu x/M + \nu y/N)] \text{ and} \quad (5.2)$$

$$f(x, y) = \sum_{\mu=0}^{M-1} \sum_{\nu=0}^{N-1} \hat{F}(\mu, \nu) \exp[j2\pi(\mu x/M + \nu y/N)], \quad (5.3)$$

respectively, to rewrite the SSMR equation as:

$$R(x_1, x_2) = \alpha(\log(I(x_1, x_2)) - \log[\mathcal{F}^{-1}(\hat{I}(\mu, \nu)\hat{F}(\mu, \nu))]) - \beta. \quad (5.4)$$

The $\hat{I}(\mu, \nu)$ and $\hat{F}(\mu, \nu)$ represent the DFTs of an input image $I(x_1, x_2)$, and a Gaussian kernel $F(x_1, x_2)$, respectively, and \mathcal{F}^{-1} represents the inverse DFT.

Exploiting the separability of the DFT and the computational efficiency of the FFT, we compute 2-dimensional transforms by applying 1-dimensional FFTs to first the rows and then the columns of the image. The computational complexity of the FFT for the 1-dimensional case is $O((N/2) \log(N))$ where N is the size of the complex input [20]. Thus the computational complexity of the 2-dimensional case (where the input image dimensions are $N \times N$) is reduced to $O(N^2 \log(N))$. The FFTs are computed using the optimized TI DSPLib. This library restricts the number of input points to a power of two so we have chosen to process a 256×256 portion of each input frame to closely match the resolution of the cameras used in our case study as discussed in Chapter 6.

The specific FFT algorithm used is the floating-point radix-2 FFT [90]. TI benchmarks the number of cycles to compute this operation by

$$C = (2n \log_2 n) + 42 \tag{5.5}$$

where C is the number of cycles, \log_2 is the base 2 logarithm, and n is the length of the complex input array [90]. For a 256-point FFT this corresponds to 4138 execution cycles, thus the C6711 operating at 150 MHz performs this operation in 27.6 microseconds (μs) under ideal benchmark conditions. To forward transform the 256 rows of a 256×256 image requires ≈ 7 milliseconds (ms). All of the 256 columns of the transformed image must then be forward transformed and later, both the rows and columns must be inverse transformed (IFFT) resulting in a total of 1024, 256-point forward and inverse transforms for the input image. The Gaussian kernel must also be forward transformed resulting in and

additional 512 FFTs, so the total number of transforms is 1,536. Prior to implementation, we felt that of all the calculations performed within the algorithm, performing the 1,536 FFTs would consume the majority of the execution time. However, experimental evidence showed otherwise as we discuss in Section 5.1.3.

5.1.2 Pre-Compute the Kernel

To reduce the number of FFTs performed we developed our first optimization for the algorithm. As is commonly done in practice we pre-compute and store the coefficients (or “twiddle-factors” [61]) used to calculate the FFT/IFFT. Our basic idea then was to use a similar technique for the Gaussian surround functions. For the SSMR there is only one scale so we only had to generate one surround function. Two key concepts were implemented that not only reduced the number of FFTs, but also significantly reduced the amount of memory that must be used by the algorithm. First, the Gaussian kernel is directly generated and applied in the spatial frequency domain thus eliminating the requirement to perform the FFT of the kernel. Second, the Gaussian is separable *and* circularly symmetric [63], and is its own (scaled) Fourier transform so it can be expressed as the product of two 1-dimensional functions and can be decomposed into horizontal and vertical projections along these dimensions. Circular symmetry implies that the two projections are the same, and the left half of either projection is the same as the right half flipped about the halfway point. Thus we only need to keep a single 128-point array of surround values to multiply with the spatial frequency domain image data. In practice we used a 256-point array to simplify indexing. Using this array instead of the full spatial frequency domain representation of the kernel saves ≈ 0.5 -MBytes.

5.1.3 Baseline Algorithm Performance

Using our ideas for the Gaussian kernel we implemented the first DSP version of the Retinex. Table 5.1 summarizes the actual measured execution time of the overall algorithm and selected components within the algorithm. These times were obtained by placing STS objects, discussed in Section 4.4, within the algorithm.

	Time (ms)
retinex	1333.42 (0.75 fps)
fwdprocessrows	476.11
fftrows	9.76
logorig	461.72
fwdprocesscols	170.77
multkernel	13.46
invprocesscols	157.83
invprocessrows	528.71
rtxeq	507.80

Table 5.1: Initial performance results from the first implementation of the SSMR.

The “retinex” item is the total time to perform the SSMR for one frame. The time to “fwdprocessrows” is the summation of (1) reading a row of image data from external memory into local memory, 2) preparing a complex input array for the FFT, (3) performing the FFT on the data, (4) storing the transformed row data back in external memory for processing at a later stage of the algorithm, and (5) calculating the logarithm of each pixel in the row and storing it in external memory. The row FFTs are computed as the first stage of transforming the image data from the spatial domain into the spatial frequency domain. The time to perform just the FFTs of the rows (256, 256-pt FFTs) is the “fftrows” item in the table. The 9.76 ms time is relatively close to the 7 ms benchmark.

The logarithm computations on the input image are also performed at this point since

the input image pixel is already in the cache for the FFT. Like the FFT data, the results are used later in the computation of the SSMR so the values are stored in external memory. These calculations represented as the “logorig” item in the table, take a very long amount of time, 461.5 ms. This time is much larger than originally anticipated. We discuss a method that we developed and applied to reduce this time in Section 5.1.4.

Similar to the “fwdprocessrows”, the “fwdprocesscols” time is the summation of (1) reading a column of image data (that has already been row transformed) from external memory, (2) performing an FFT on the data completing the 2-dimensional image transform, (3) multiplying the now spatial frequency domain image data with the kernel, and (4) storing the processed image data back into external memory for further processing at a later time. The multiplication of the spatial frequency domain image data with the kernel also takes a considerable amount of execution time — 13.46 ms shown as “multkernel” in the table. We discuss a method that we developed and implemented to significantly reduce this in Section 5.1.6. The “invprocesscols” and the “invprocessrows” times are the summations of (1) reading a column/row from external memory (2) performing an inverse FFT on the column/row, and (3) storing the column/row in external memory. The “invprocessrow” item also includes the time to calculate the last stage of the algorithm — the final equation to generate each output pixel value after all preliminary values have been calculated. The time for the “rtxeq” item represents this value. The time to compute this stage is also very long because it contains the second calculation of the logarithm function applied to the convolved image data within it.

.

5.1.4 Pre-Compute the Logarithm

Directly executing the logarithm function is an expensive operation. The C6711 run-time support library benchmarks 952 execution cycles for a double-precision (64-bit) natural logarithm calculation and 152 execution cycles for a single precision (32-bit) calculation [85]. Thus with a clock speed of 150 MHz, each double-precision log operation requires $6.35\mu\text{s}$. This operation is performed for every pixel so the total benchmark time is 415.93 ms corresponds closely to the value obtained¹. Our initial implementation used this double-precision function call. However, using the single-precision function does not sacrifice image quality. Changing to the single-precision function reduced the “log_orig” time from 461.72 ms to 69.05 ms, and the “rtxeq” time from 507.80 ms to 92.82 ms. This reduced the total Retinex execution time, “retinex”, from 1333.42 ms (0.75 fps) to 525.83 ms (1.90 fps). This is a substantial decrease in the execution time of the algorithm, but the logarithm computation is still a significant portion of the total time.

To further eliminate this bottleneck we used the fact that the input to the logarithm is limited to integer values in the range of 0 to 255, and formulated the idea of pre-computing the logarithm values and storing the values in look-up tables (called log tables). We generated another optimization by embedding the Retinex parameters α and β into the log tables. In observing the SSMR equation from Chapter 2 (repeated here for convenience),

$$R_i(x_1, x_2) = \alpha(\log(I_i(x_1, x_2)) - \log(I_i(x_1, x_2) * F(x_1, x_2))) - \beta \quad (5.6)$$

¹The slight discrepancy is due to minor additional operations, such as data type conversions, that are performed within the measurement interval, and loop indexing and STS object overhead.

we can distribute α and group β with the first term to produce

$$R_i(x_1, x_2) = \underbrace{(\alpha \log(I_i(x_1, x_2)) - \beta)}_{P_i(x_1, x_2)} - \underbrace{(\alpha \log(I_i(x_1, x_2) * F(x_1, x_2)))}_{Q_i(x_1, x_2)}. \quad (5.7)$$

where

$$P_i(x_1, x_2) = (\alpha \log(I_i(x_1, x_2)) - \beta) \quad (5.8)$$

for $I_i(x_1, x_2) \in \{1, \dots, 255\}$

and

$$Q_i(x_1, x_2) = (\alpha \log(I_i(x_1, x_2) * F(x_1, x_2))) \quad (5.9)$$

for $(I_i(x_1, x_2) * F(x_1, x_2)) \in \{1, \dots, 255\}$.

If $I_i(x_1, x_2) = 0$ then we assign $P_i(x_1, x_2) = -\beta$, and if $(I_i(x_1, x_2) * F(x_1, x_2)) = 0$ then we assign $Q_i(x_1, x_2) = 0$. We can generate two log tables: the first one for $P_i(x_1, x_2)$ and the second one for $Q_i(x_1, x_2)$. The tables require 1-KByte each, so the additional memory for two tables instead of one is insignificant. The simple regrouping and embedding of α and β eliminates one multiplication and one addition per pixel per band (i in the equations above) and could save up to 131,042 execution cycles per band depending upon the order of implementation². The most important reduction though is just from using table look-up. The measurement results are shown in Table 5.2. The time to perform the logarithms is now 18 times less than when using direct single-precision logarithm calculations! The total

²If properly ordered the multiply-accumulate function of the DSP can perform this operation in 65,536 execution cycles per band.

execution time is now 385.05 ms which corresponds to 2.59 fps. This is still well below from our minimum target value of 15 fps for real-time processing.

	Time (ms)
retinex	385.05 (2.59 fps)
fwdprocessrows	21.0
ffthrows	9.94
logorig	3.67
fwdprocesscols	170.85
multkernel	12.39
invprocesscols	157.65
invprocessrows	35.52
rtxeq	14.46

Table 5.2: Performance measurements after using logarithm tables and combining α and β .

As can be seen from Table 5.2, there is a large discrepancy in the time it takes to process a row versus a column: the “fwdprocesscols” time is eight times that of the “fwdprocessrows” time! If the principal cost of computations were the FFT, the time to perform both of these operations should be roughly the same. We determined that the row and column times are substantially different because the processing is not driven by FFT computations, but rather by data transfers. To quantify this, additional STS objects were added to directly measure the column read and write times. To read a complex 256-point integer column from external memory and to write it back required 148.3 ms. This represents over 93% of the “fwdprocesscols” time.

The primary cause of the discrepancy between row and column execution time can be determined by examining the memory requirements of the algorithm and the DSP architecture. The most efficient data processing operations occur when the processor has very fast access to the data, i.e., when the data is located in the cache or in L2 memory. While we

do not have direct write access to the L1P or L1D caches, we do have access to, and some control over, the next fastest access location: L2 memory. The C6711 has a 64-KByte L2 memory that can be configured as cache, SRAM, or a combination of the two as discussed in Section 3.1. Optimum performance can be obtained if all of the transformed image data is located in the L2 memory, but unfortunately, 64-KBytes is nowhere near the required capacity: the input image itself is 64-KBytes. Additionally, the DSPlib FFT routines require input and output data in complex format, i.e. each point must have a real and imaginary (zero for our input purposes) component, which doubles the storage size. Also, the data is in floating point (four byte) format, so the actual memory required to store a transformed 256×256 image is 512-KBytes. Thus the image data must be kept and fetched from external memory.

Operating directly on data located in external memory incurs a large performance penalty, so for performing the FFT efficiently on a row of an image requires reading all the contiguous pixels of the row from external memory into a buffer located in L2 memory. The first pixel read of a row is accompanied by reading in 3 additional pixel points into the 32-Byte line size of the L1D cache. Accessing the first pixel causes L1D cache and L2 memory misses, but accessing the next three pixels in the row returns a cache hit and the data is retrieved in one clock cycle. To process a column requires accessing non-contiguous pixels with a stride difference equal to the number of columns. So, transferring a column of image data from external memory generates a L1D and L2 memory miss for each pixel thus severely degrading performance. Additionally, we cannot take advantage of any temporal locality for the data since we are only using the data once at this point within the algorithm. In order to improve the L2 memory transfer time for column-wise image data

we must change the mechanism for access.

5.1.5 Use DMA to Transfer Columns

Our next idea was to use the EDMA controller in the C6711 to handle data transfers between L2 memory and external memory. This saves processor cycles used to transfer the data, and, since the transfer can be performed in the background, this enables overlapping processor execution with data transfers if coordinated correctly. The chip support library for the C6711 provides the capability to perform 2-dimensional transfers by specifying the number of bytes per line, the number of lines, and the number of bytes between the start of one line and the next. If we set these parameters to transfer a column of image data, we can exploit the efficiency of this transfer to speed up column processing of the image.

	Time (ms)
retinex	134.44 (7.44 fps)
fwdprocessrows	19.05
fftrows	9.90
logorig	3.27
fwdprocesscols	39.24
multkernel	9.84
invprocesscols	28.73
invprocessrows	47.86
rtxeq	16.21

Table 5.3: Performance results after using 2D DMA data transfers.

The improvements gained by using this method are shown in Table 5.3. The total time to transfer and perform processing on the columns is now only 67.97 ms as compared to 328.5 ms earlier, thus reducing the total SSMR execution time down to 134.44 ms (7.44 fps). Note that the “multkernel” execution time is reduced because the processor does not have to wait for data to arrive from external memory to begin execution. However, the “invprocessrows”

time has increased. This occurs because the next processing stage must now wait until the last column transfer is complete to begin execution. In the prior implementation this function was part of the “invprocesscols” but was composed of execution cycles to transfer data rather than wait cycles. We discuss methods that we developed to eliminate this and other wait cycles in Section 5.1.8.

5.1.6 Reduce Gaussian Kernel Computations

A property of the Gaussian function that we can exploit to significantly improve performance is that the tails of the function rapidly decrease to zero for large σ . This implies that a large percentage of values in the 256-point Gaussian kernel array will be zero. If we preset (to zero) the buffer that will hold the convolution result, the loop to process the convolution can be terminated early with proper indexing and checks for the first zero value in the surround array. Table 5.4 shows the result of implementing this optimization. The time to multiply the kernel is reduced from ≈ 9 ms to $150\mu\text{s}$ with $\sigma = 80$. We should note that this time is dependent upon the extent of the surround, and the performance will degrade, ultimately back to 9 ms, as narrower surrounds³ are chosen.

We also discovered that performance can be improved by changing the way one initiates the complex array. To generate the complex input array for the FFT we must interleave a real (image data) value with an imaginary (zero) value. Ordinarily one would simply zero out the array by using some function call and then fill in every even indexed array value with the real components. We found that it is more efficient to write the real component and then immediately write zero into the next array value. This occurs because we only

³A narrow surround in the spatial domain is wide in the spatial frequency domain and vice versa.

	Time (ms)
retinex	125.04 (7.41 fps)
fwdprocessrows	18.99
fftrows	9.91
logorig	3.29
fwdprocesscols	29.33
multkernel	0.15
invprocesscols	28.96
invprocessrows	47.76
rtxeq	16.29

Table 5.4: Performance results after using 2D DMA data transfers.

have to load and access the input array in the L1D cache once instead of twice plus function call overhead for the first method.

5.1.7 Merge Algorithm Components

The next significant performance increase was obtained by identifying redundant transformation cycles in the algorithm. In our original implementation we performed the following sequence of operations:

- For all rows: read in row, FFT, and write the result to external memory,
- For all columns: read a column, FFT, and write the result to external memory,
- For all columns: read a column, convolve with the Gaussian kernel, and write the result to external memory
- For all columns: read in column, IFFT, and write the result to external memory,
- For all rows: read a row, IFFT, and write the result to external memory.

The remainder of SSMR calculation is then performed. We can take advantage of the independence of each column of image data by merging some of the preceding steps and thus eliminating several stages of data transfers. As soon as we have performed the FFT of a column, we can continue processing this column, multiplying it with the kernel, and then immediately perform an IFFT of the column. The processing stages then become:

- For all rows: read in a row, FFT, and write the result to external memory.
- For all columns: read in a column, FFT, multiply with the Gaussian kernel, IFFT, and write the result to external memory.
- For all rows: read in a row, IFFT, and write the result to external memory.

This saves four read and write transfers to external memory. Table 5.5 shows the results of implementing this optimization. The “fwdprocesscols” and “invprocesscols” items are now merged into the “processcols” item. Additional optimizations were also performed to reduce the “rtxeq” time. This includes moving all tables into L2 memory and performing a 1-dimensional DMA transfer for the final output values. The total execution time of the algorithm is down to 83.06 ms (12.04 fps). This is now approaching real-time performance.

5.1.8 Minimize Data Transfer Overhead

We then focused on formulating and applying a method to minimize the overhead of transferring data between external and internal memory. Instead of using processor cycles to perform this function, we used the DMA capability within the processor to perform all external-to-internal memory transfers. We were already using this function to perform 2-dimensional column transfers as mentioned in Section 5.1.5 and 1-dimensional array trans-

	Time (ms)
retinex	83.06 (12.04 fps)
fwdprocessrows	17.42
fftrows	9.8
logorig	3.25
processcols	41.14
multkernel	0.16
invprocessrows	34.11
rtxeq	5.3

Table 5.5: Performance results after merging algorithm stages. Since the forward and inverse column execution times are effectively merged together, the time to process columns is now in item “processcols”

fers for the final output values of the algorithm as discussed in Section 5.1.7. We now add additional DMA transfers for the row data transfers of FFT data and for the logarithm of the input image data. Storing the logarithm of the input data requires 256-KBytes, far larger than the memory available in the L2 memory, so these values must be kept in external memory.

Performing DMA transfers and waiting for completion obviously reduces the effectiveness of using DMA. To avoid this we implemented a double buffering scheme to move from a data I/O-limited algorithm to a execution cycle-limited algorithm. As noted earlier, DMA allows data transfers to occur independently or in the background of any processor activity. We developed an algorithm and implemented a series of buffers so that as we process one buffer, we simultaneously transfer in the next data to be processed. This double buffering scheme was used for all DMA transfers and removed the requirement to wait for any DMA transfer. Without having to wait, reading in more than one unit of transfer (e.g. two rows or two columns) did not improve performance.

5.1.9 Use Cache-optimized FFTs

After all of the previous optimizations, we returned to trying to improve the FFT. We identified and applied a more efficient form of the FFT algorithm, a cache-optimized ($SP \times SP$) algorithm that allows the use of mixed radix FFTs that can be calculated in multiple passes. A 256-point FFT only needs one pass and can be effectively calculated using the cache-optimized FFT in radix-4 mode. The benchmark equations for the cache-optimized FFT suggested that we could obtain better performance from this version versus the radix-2 form. The number of cycles C to compute the FFT using this equation is given by:

$$C = (3\lceil \log_4(n-1) \rceil n) + (21\lceil \log_4(n-1) \rceil) + (2n) + 44 \quad (5.10)$$

where C is the number of cycles, \log_4 is the base 4 logarithm, and n is the length of the complex input array. For a 256-point FFT $C = 2923$ cycles, or $19.5 \mu s$, corresponding to a 30% increase in FFT performance. To forward transform the 256 rows of an image 256×256 image takes ≈ 5 ms.

Implementing the double buffering scheme and changing the FFT algorithm for the C6711 allowed us to achieve our final C6711 SSMR execution time of 48.33 ms or 20.7 fps. Table 5.6 shows the timing for the individual components of the algorithm. A sample output image frame from a video taken of a bookcase is displayed in Figure 5.1. The input image is shown on the left while the Retinex enhanced image is shown on the right. The enhanced image has greatly improved contrast and sharpness. Details that are indistinguishable in the original are easily noticed in the enhanced image.

	Time (ms)
retinex	48.33 (20.7 fps)
fwdprocessrows	12.83
fftrows	6.98
logorig	3.20
processcols	20.55
multkernel	0.16
invprocessrows	14.92
rtxeq	5.59

Table 5.6: Final SSMR performance results using the C6711 DSP.

5.2 Map Optimized SSMR to C6713

To improve and compare performance we mapped the same optimized SSMR code developed for the C6711 onto the C6713. Considering the similarity in architectures this should provide a near linear increase in performance corresponding to the increase in clock speeds between the devices. Thus performance should improve by 50% (225/150) and the expected frame rate should be close to 31 fps. The larger L2 memory on the C6713 is not used because all of the memory allocated in the current implementation fit in 64-KBytes, and the extra 192-KBytes of L2 SRAM on the C6713 are not large enough to move any of the significant data structures into on-chip memory.

After porting the code to the C6713, the algorithm ran successfully and we obtained a frame rate of only 28 fps. The 35% increase is sub-linear. This occurs because the C6713 EVM has a slower EMIF clock that controls the transfer rate to external memory. The C6711 EVM uses a 100 MHz EMIF clock while the C6713 EVM uses a 90 MHz clock. This reduces the external data transfer rate to the extent that the processor must now wait for DMA transfers to complete.

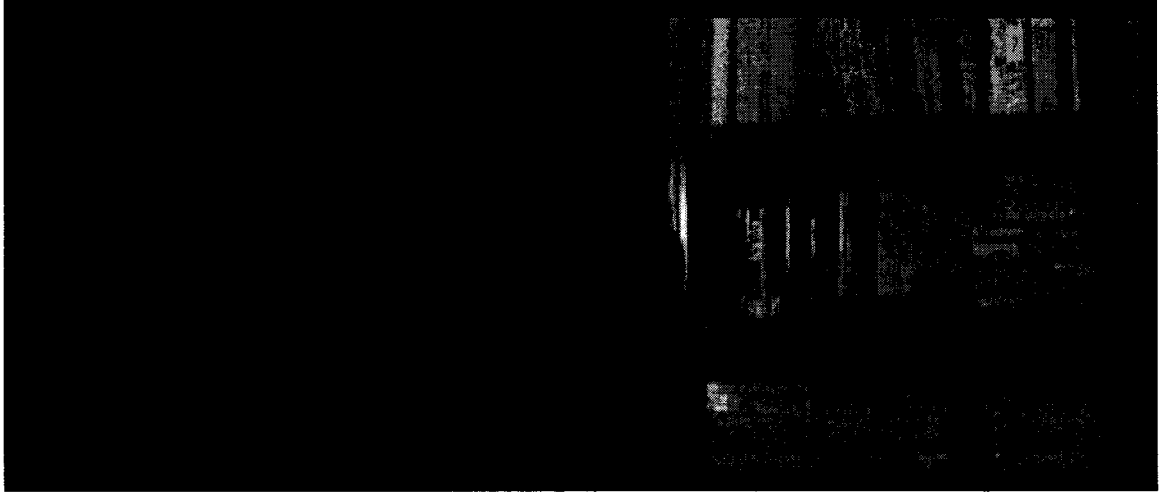


Figure 5.1: Capture Video Frame with input from camera on the left, and Retinex output on the right. Retinex parameters are $\alpha = 175$, $\beta = 135$, and $\sigma = 80$ — note that we are nearly reaching the noise limit of the camera.

5.3 Map Optimized SSMR to DM642

Although either of the C671X platforms would perform adequately for many applications, it is obvious that neither has the performance capability to meet real-time multi-spectral, multi-scale Retinex processing requirements. So next, we ported the SSMR algorithm to the DM642 platform. Although the DM642 uses different image capture and display drivers, DMA mechanisms, and FFT algorithms than the C6711/C6713, the core structures and methods developed and implemented on the C6711 remained the same. Directly comparing DM642 MIPS with the C6711 shows a potential four-fold increase in performance. However, other factors such as extra computations to handle fixed point arithmetic, and different processor specific instructions, libraries, and EMIF bus speeds affect performance. These modifications do not allow a direct comparison with C67X performance but we should anticipate approximately 70 to 90 fps performance.

Fixed-point arithmetic limits the dynamic range of the DM642 to $2^{31} - 1$. This is sufficient for some portions of the algorithm. For example, the input to a 256 point radix-4 FFT is processed in 4-stages where each stage gives 2 bits of growth. Our 8-bit image data input will then only grow to a maximum of 16 bits for one forward transform. Since we generate a 2-dimensional Fourier transform, a second 256-point FFT is also performed. This increases the growth to 32 bits which still fits in a standard integer data type. However, the now spatial frequency domain image data is then multiplied with a kernel. The largest numbers from the FFT operation are on the order of 10^8 . The smallest numbers from the normalized spatial frequency Gaussian kernel are truncated⁴ at 10^{-6} . Thus we must process values on the order of 10^{14} which, without scaling, is well beyond the capability of 32-bit fixed point representation.

To perform scaling we invoke a few simple arithmetic conventions. For example, to multiply an integer number I by 0.6913 (which equals $\log(2)$) one could perform

$$R = ((I * 6913) + 5,000)/10,000 \quad (5.11)$$

where 5,000 is added to perform rounding. If $I = 46$, floating-point multiplication yields 31.7998 while our fixed-point method yields $R = 32$. Because a shift left operation is equivalent to division by 2, we can improve the efficiency of this operation by dividing by a number that is a power of 2. Using 8192 (2^{13}) in our previous example our new multiplier becomes $0.6913 * 8192 = 5663.1296$. We could chose 5663 or 5664 depending upon which

⁴Significant digits beyond 10^{-6} are truncated without affecting image quality.

value is more accurate. Choosing 5663, our scaling equation above becomes

$$R = ((I * 5663) + 4,096)/8192 \quad (5.12)$$

and if $I = 46$ again, then $R = 32$. When scaling one must also be careful with proper selection of the storage classes used to hold intermediate results. Recall from Section 5.1.2 that we exploit the symmetry of the Gaussian to save memory space, so to compute the kernel values in the spatial frequency domain we multiply two of the properly indexed array values together. We scale each individual array value by 2^{19} in order to retain as much resolution as possible, so the final spatial frequency domain kernel values are on the order of 2^{38} . Multiplying by the maximum spatial frequency domain image values ($\approx 2^{24}$) results in values on the order of 2^{62} . Fortunately the TI compiler supports 64-bit signed and unsigned integer (long-long) data types. An alternative, but less efficient, method to minimize the size of internal values is to generate the inverse of the spatial frequency domain kernel values and use division instead of multiplication. The division operation is implemented on the DM642 by repeatedly issuing a conditional subtract operation (SUBC) instruction. After carefully balancing scaling and truncation tradeoffs a fixed-point version of the algorithm was implemented with the log values scaled by 2^{20} and Gaussian kernel table values are scaled by 2^{19} . These values maximize the retained precision without causing overflow in intermediate or final output calculations.

5.3.1 Apply Intrinsic

Another algorithm optimization implemented at this stage was to use intrinsics, originally mentioned in Section 4.2, at strategic points within our code. For example, to clamp final

Retinex output data to values between 0 and 255, we used two intrinsics, `min2` which returns the lesser of two inputs and `max2` which returns the greater of two inputs, and formed an instruction similar to `min2(max2(output_value,0),255)`. Measuring the performance⁵ of this instruction using STS objects results in 8.8 ns per pixel (2.24 μ s per 256 \times 256 image). As a comparison, to clamp the Retinex output using a standard if-then-else expression (if `output_value < 0 output_value = 0`, else if `output_value > 255 output_value = 255`) requires 27.4 ns per pixel (7.01 μ s per image). The instruction using intrinsics is over 3 times faster. After implementing the proper scaling operations and embedding intrinsics, we achieved an execution time of 17.89 ms (55.89 fps) for the SSMR on the DM642. This is still below our anticipated 70 to 90 fps.

5.3.2 Modify the Architecture

We determined that I/O was again limiting performance. The faster DM642 processor, even performing the additional scaling calculations, executes the algorithm quicker, thus at various points in the code the processor now has to wait for DMA transfers to complete. We eliminated this by making an architectural change on the DM642 EVM. The default EMIF bus rate is 133 MHz. We were able to increase the EMIF bus rate to a chip maximum 200 MHz, effectively over-clocking the SDRAM, by strapping the appropriate resistors onto the DM642 EVM module and changing memory access timing parameters. Implementing this modification increased SSMR performance to 69.15 fps effectively meeting our anticipated performance.

⁵This measurement was performed on the C6416 processor, but the ratio remains the same for the other processors.

5.4 Multi-Spectral Multi-Scale Retinex Optimizations

The performance achieved for the SSMR on the DM642 platform provided a baseline to pursue real-time multi-spectral, multi-scale Retinex (MSR) performance. Expanding from a single scale to multiple (three) scales primarily involves two additional computational requirements — (1) performing the additional convolutions and (2) weighting and combining the convolution results. We implemented the same technique previously developed for the SSMR except we pre-compute a *series* of Gaussian kernels directly in the spatial frequency domain and store the values in tables. The range of σ was constrained to values between from 5 to 260 in steps of 5. Each scale would then use a pointer to the appropriate table of the associated σ value. Since σ is static for each scale, the pointers are set prior to calling the Retinex function. The total size of all the Gaussian tables is now 52-KBytes. We could not keep this number of tables in memory on the C671X processors.

5.4.1 Reuse Transformed Input Image

Since the same input image data is convolved with each kernel, the optimum stage to perform this function is as each column is read from external memory and transformed. The sequence of operations at the convolution stage then becomes

- Read a column, FFT, multiply with kernel 1, IFFT, and DMA result to external memory.
- Multiply the same column with kernel 2, IFFT, and DMA result to external memory.
- Multiply the same column with kernel 3, IFFT, and DMA result to external memory.

This not only reuses the same spatial frequency domain image data, but also allows DMA transfers to be overlapped with processor activity. Double buffering is still implemented on column reads to ensure that column data is always present in local memory for processing. We improved the buffering scheme by accumulating the first spatial frequency domain column earlier during the first stage of row processing. The amount of memory needed to hold the convolved image data is now 1.5-MBytes — three times the previous requirement of 512-KBytes. After the convolution stage, the image data for each scale must then be transferred back into local memory, inverse transformed, weighted, and combined with the other scales. Again, we use DMA to retrieve the data back into local memory, and double buffering to perform this transfer in the background.

5.4.2 Reduce Computations

One major optimization idea we developed and applied for weighting and combining the scales is to rearrange the Retinex equation to reduce the number of operations that must be performed. It has been shown that using equal weighting factors provides good Retinex enhancement in many conditions [32]. We exploit this fact by distributing the weighting factors in the Retinex equation

$$R_i(x_1, x_2) = \sum_{k=1}^K W_k (\log(I_i(x_1, x_2)) - \log(I_i(x_1, x_2) * F_k(x_1, x_2))) \quad (5.13)$$

$$= \sum_{k=1}^K W_k (\log(I_i(x_1, x_2))) - \sum_{k=1}^K W_k (\log(I_i(x_1, x_2) * F_k(x_1, x_2))) \quad (5.14)$$

and noting that if the W_k s are equal ($W_k = W$) and $\sum_{k=1}^K W_k = 1$ then our equation becomes

$$R_i(x_1, x_2) = (\log(I_i(x_1, x_2))) - W \sum_{k=1}^K (\log(I_i(x_1, x_2) * F_k(x_1, x_2))) \quad (5.15)$$

This saves two logarithm computations, two subtractions, and two multiplications *per pixel*.

An additional reduction in calculations is gained by combining the proper weighting factors into the tables already used for the two pre-computed log tables discussed in Section 5.1.4. To pre-compute the second log table (the log table combined with β only) values, if two or three scales are used, then simply divide these values by the associated number of scales, 2 or 3, respectively.

The next requirement is to add to the multi-scale algorithm the capability to process in real-time multiple (three) spectral bands, i.e. color video. This addition is not quite as simple as just executing the same multi-scale algorithm on each band, particularly when embedding optimizations to improve performance. First, to perform color processing the image data should be in the RGB color space. The video decoders and encoders on the EVMs only work in the $Y' C_B C_R$ color space. For monochromatic processing we only have to extract the luma component from the $Y' C_B C_R$ input stream. As discussed in Section 4.1.2, on the C671X and DM642 EVMs the $Y' C_B C_R$ data is stored in planar format so only a pointer is required to address the Y' component. On the C6416 EVM the $Y' C_B C_R$ data is stored in interleaved format so the Y' component must be extracted from the frame data. This is easily accomplished by using 2-dimensional DMA calls discussed in Section 5.1.5. However whether the image data is in planar or interleaved format, the Y' data does not need to be converted as it does if color processing is to be performed.

For color processing, the $Y'C_B C_R$ input data must be converted [53] into the RGB 888 color space⁶ and the processed RGB data must be converted back into the $Y'C_B C_R$ color space for output into the video encoders. The following equations from Poynton [53] are used to convert between $Y'C_B C_R$ and gamma-corrected⁷ 8-bit computer⁸ RGB ($R'G'B'$):

$$R' = 1.1644(Y' - 16) + 1.5960(C_R - 128) \quad (5.16)$$

$$G' = 1.1644(Y' - 16) - 0.3918(C_B - 128) - 0.8129(C_R - 128) \quad (5.17)$$

$$B' = 1.1644(Y' - 16) + 2.0172(C_B - 128) \quad (5.18)$$

then converting into fixed-point format using a scaling factor of 2^{13} , the conversion equations above become

$$R' = ((9539(Y' - 16) + 13075(C_R - 128) + 4096) \gg 13) \quad (5.19)$$

$$G' = ((9539(Y' - 16) - 3209(C_B - 128) - 6660(C_R - 128) + 4096) \gg 13) \quad (5.20)$$

$$B' = ((9539(Y' - 16) + 16525(C_B - 128) + 4096) \gg 13). \quad (5.21)$$

To encode 8-bit $Y'C_B C_R$ from $R'G'B'$ we use the following equations:

$$Y' = 0.2568R' + 0.5041G' + 0.0979B' + 16 \quad (5.22)$$

$$C_B = -0.1482R' - 0.2910G' + 0.4392B' + 128 \quad (5.23)$$

$$C_R = 0.4392R' - 0.3678G' - 0.0714B' + 128. \quad (5.24)$$

⁶In RGB 888, each pixel is represented by an 8-bit red, green, and blue component

⁷Gamma-correction refers to the non-linear transfer function applied to RGB values in most imaging systems. This is used to mimic perceptual response [53]

⁸Computer RGB uses the full 8-bit range with black at code 0 and white at code 255.

Again, converting into fixed-point format using a scaling factor of 2^{13} yields

$$Y' = ((2104R' + 4130G' + 802B' + 4096) \gg 13) + 16 \quad (5.25)$$

$$C_B = ((-1214R' - 2384G' + 3598B' + 4096) \gg 13) + 128 \quad (5.26)$$

$$C_R = ((3598R' - 3013G' - 585B' + 4096) \gg 13) + 128. \quad (5.27)$$

All RGB and $Y'C_BC_R$ values should be clamped between 0 and 255, and 16 and 235 respectively. In practice we simplify these equations by eliminating the redundant calculations.

5.4.3 Buffer Across Spectral Bands

Another technique we developed to maintain our I/O performance is to modify our row double-buffering scheme to buffer data across spectral bands. This modification is done only on the row output processing stage since we need data simultaneously from all three bands during this stage. When processing a row of data for the red spectral band, instead of performing a DMA of the next row of red spectral data, we DMA the next row of green spectral data. Similarly, when processing the green band, we DMA the next blue band, and when processing blue band, we DMA next red band. So our buffering sequence becomes

- DMA the red band
- loop start: DMA the next green band; process the red band
- DMA the next blue band; process the green band
- DMA the next red band; process the blue band; combine bands; end loop.

After all three channels for a row are processed, i.e. the blue band is complete, the bands are combined and converted to $Y'C_B C_R$. This optimization continues to maximize the processing load by keeping data transfers in the background.

5.4.4 Allocate Log Values in L2 Memory

The additional transfer buffers and tables used in the optimizations discussed so far are statically allocated in L2 memory. All of these data structures easily fit in the 256-KBytes of L2 memory on the DM642 with a nominal allocation of ≈ 175 -KBytes used in our implementation. However the DM642 L2 memory is still not large enough to hold all of the processed image data at any stage of the algorithm. As mentioned in Section 3.4, the C6416 is not only faster but has a larger L2 memory of size 1-MByte. We exploit this feature by keeping all of the logarithm of the original image data, 768-KBytes, in L2 memory. This uses nearly all of the L2 memory with a total allocation 1,011,904 Bytes, but by keeping this data local we eliminate all of the associated DMA transfers and thus improve performance.

5.4.5 MSR Performance Results

To measure the performance of the MSR we used both the DM642, with EMIF bus speeds of 133 MHz and 200 MHz, and the C6416 processors on their respective EVMs in our test-bed outlined in Chapter 5. The graphs in Figures 5.2, 5.3, and 5.4 show the performance obtained on the processors for the Retinex with 1 to 3 scales and 1 to 3 spectral bands. The vertical lines are the cutoff points for real-time performance based on 15 fps and 30 fps. The same data is shown in tabular form in Table 5.7. Execution time is shown in milliseconds, and in frames per second in parenthesis. The values for the Gaussian surrounds, σ , are 5

for 1 scale, 5 and 80 for 2 scales, and 5, 80 and 200 for 3 scales. The gain α and offset β values are 250 and -100 respectively.

For 1 spectral band, implementations of the algorithm on both processors meet the 15 fps, and 30 fps real-time requirements for all scales. For 2 spectral bands implementations on both processors again meet the 15 fps target. With 2 or 3 scales, only the C6416 meets the 30 fps target. The DM642 with a 200 MHz EMIF only meets this target for 1 scale. For 3 spectral bands, only the implementation on the C6416 with 1 scale meets the 30 fps target. Performance for 3 bands, 3 scales is 20.25 fps. For the 200 MHz EMIF DM642, 3 band 3 scale performance is at 13 fps, just missing the 15 fps target. Interestingly, although all implementations on each processor performed linearly, the slopes progressively decrease from the plots for the C6416 to the 200 MHz DM642, and to the 133 MHz DM642 respectively on all three graphs. This may be due to the fact that more data is kept local to the processor for the C6416. When there are more Retinex computations, there is more data to be transferred, and so the algorithm becomes more I/O driven, degrading performance at a faster rate than if it was more controlled by processing cycles as it is for the C6416.

For comparison purposes we placed STS objects in the code on the C6416 to measure the execution time of the different stages of algorithm like we did earlier for the C6711. Table 5.8 shows the best single-scale, monochrome Retinex performance on the C6711 and on the C6416 DSPs. Note the significant decrease in the time required to process the FFT. The specific FFT used from the DSPLib for the C6416 is the mixed-radix 16×32 -bit FFT⁹.

⁹The 16×32 refers to the bit width of the coefficients, and the input and output data, respectively.

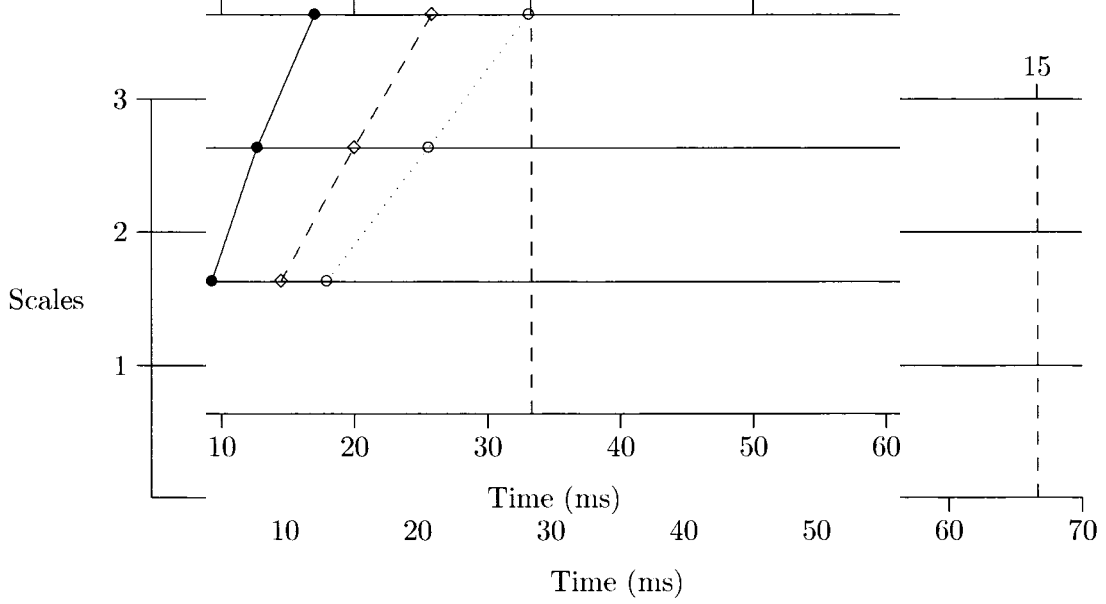


Figure 5.2: Retinex performance in time (bottom axis) and frames per second (top axis) to process 1 spectral band of image data on DM642 with 133 MHz EMIF (dotted line), DM642 with 200 MHz EMIF (dashed line), and C6416 (full line).

The benchmark number of cycles to compute this FFT is given by [83]:

$$C = (13n/8 + 24)\lceil \log_4(n) - 1 \rceil n + (n + 8)1.5 + 27. \quad (5.28)$$

For $n = 256$, the length of the FFT, $C = 1743$ cycles. This corresponds to $1.743\mu\text{s}$ on the 1 GHz C6416. So based on the benchmark equation, to forward transform the 256 rows of a $256 \times$ image takes $\approx 446\mu\text{s}$. Our measured FFT time is $516\mu\text{s}$ nearly meeting the benchmark. Also note the significant decrease in time for “rtxeq” is due to the use of intrinsics, loop index and equation simplifications, and the increase in processor speed. Finally, we also note the increase in time to multiply by the kernel. This occurs because of the scaling operations performed at this stage of the algorithm and the required use of

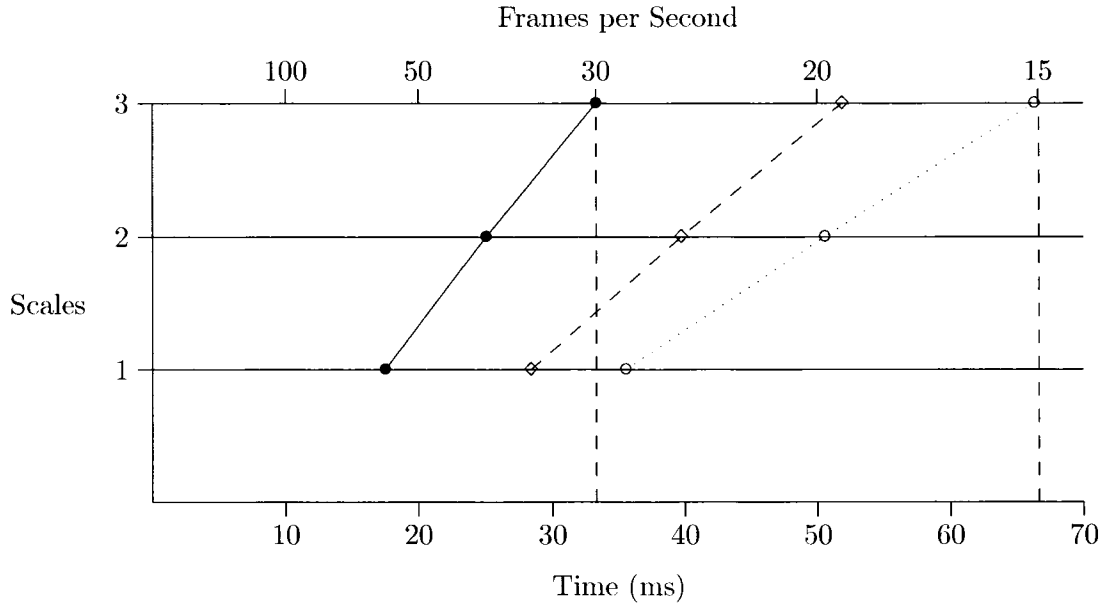


Figure 5.3: Retinex performance in time (bottom axis) and frames per second (top axis) to process 2 spectral bands of image data on DM642 with 133 MHz EMIF (dotted line), DM642 with 200 MHz EMIF (dashed line), and C6416 (full line).

inefficient long-long data types to hold intermediate values. As one final execution time measure we also tested the algorithm without any internal measurement instrumentation and only for 1 scale and 1 band. With these simplifications we obtained an execution time of 8.9 ms (112.36 fps).

We also measured CPU load for the C6416. Unlike the previous Retinex timing measures which only encompass the Retinex task, this is a global measure which includes frame acquisition. Table 5.9 shows the values obtained under different Retinex configurations. For the lower computational requirement configurations (1 spectral-band, or 2 spectral bands and 1 or 2 scales, or 3 spectral bands and 1 scale) the processor is underutilized. Only a small percentage of these unused execution cycles are spent waiting for DMA to complete

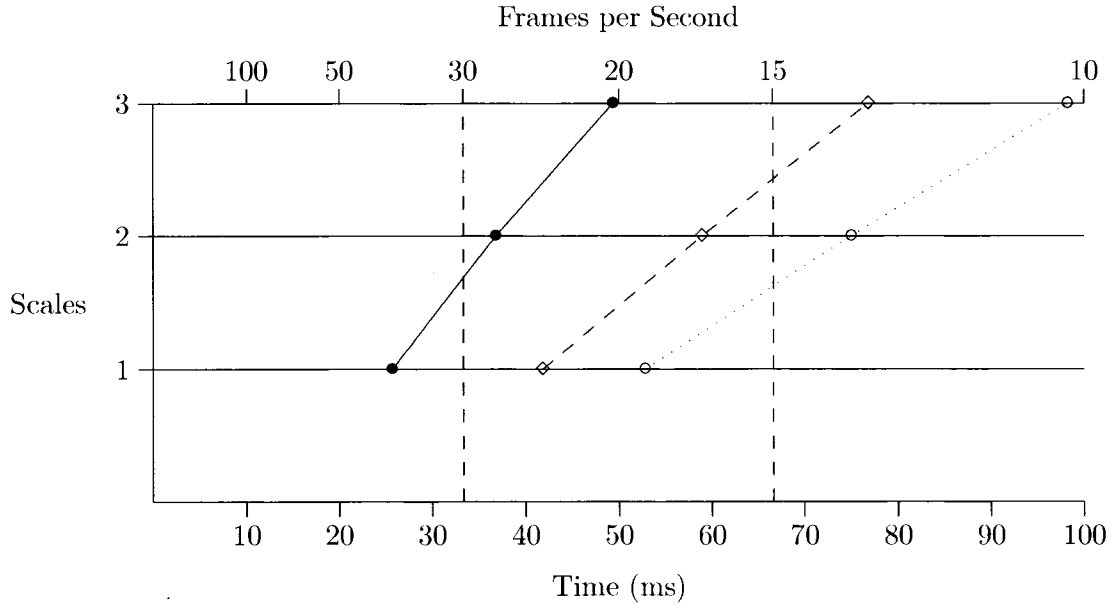


Figure 5.4: Retinex performance in time (bottom axis) and frames per second (top axis) to process 3 spectral bands of image data on DM642 with 133 MHz EMIF (dotted line), DM642 with 200 MHz EMIF (dashed line), and C6416 (full line).

due to the highly optimized code at this point. Only one or two DMA wait statements have to be inserted into the algorithm to achieve correct operation, and this is only for the single band, single scale case. The majority of the unused execution cycles are spent simply waiting for the next frame from the input camera.

To visible demonstrate the performance of the real-time algorithm we processed a video of an outside scene at NASA LaRC in Hampton, Virginia. The video was taken on November the 8th, 2005 between 5:15 PM and 5:30 PM using a standard Sony TRV-20 videocamera. Sunset on this day was at 5:02 PM. For presentation in this dissertation, we have extracted 3 snapshots from the processed video. The first snapshot, shown in Figure 5.5 is taken 40 seconds into the video. The second snapshot, shown in Figure 5.6, is taken 6 minutes and

Retinex Execution Time Table		Bands		
		1	2	3
1 scale	DM642/133 MHz EMIF	17.89 (55.9)	35.54 (28.1)	52.77 (18.9)
	DM642/200 MHz EMIF	14.46 (69.1)	28.39 (35.2)	41.84 (23.9)
	C6416	9.24 (108.2)	17.5 (57.1)	25.66 (38.9)
2 scales	DM642/133 MHz EMIF	25.54 (39.1)	50.55 (19.8)	75.04 (13.3)
	DM642/200 MHz EMIF	19.98 (50.1)	39.74 (25.2)	58.96 (16.9)
	C6416	12.68 (78.9)	25.06 (38.9)	36.83 (27.1)
3 scales	DM642/133 MHz EMIF	33.11 (30.2)	66.32 (15.1)	98.25 (10.2)
	DM642/200 MHz EMIF	25.79 (38.8)	51.85 (19.3)	76.86 (13.0)
	C6416	17.03 (58.7)	33.11 (30.2)	49.37 (20.3)

Table 5.7: Measured Retinex performance on DM642 and C6416 processors. The 133 and 200 refer to the clock speed of the EMIF bus. Measurement units are in both milliseconds, and frames per second in parentheses.

	C6711 Time (ms)	C6416 Time (ms)
retinex	48.33 (20.7 fps)	9.24 (108.23 fps)
fwdprocessrows	12.83	1.3
fftrows	6.98	516 μ s
logorig	3.20	141 μ s
processcols	20.55	6.43
multkernel	0.16	2.18
invprocessrows	14.92	1.49
rtxeq	5.59	571 μ s

Table 5.8: Comparison of final SSMR performance using the C6711 and the C6416 DSPs.

28 seconds into the video. The third snapshot, shown in Figure 5.7, is taken 14 minutes and 28 seconds into the video. The unprocessed video frames are on the left. They show the scene as captured by the video camera. The progressive darkening of these images is due to the sunset. The real-time Retinex enhanced frames using the C6416 EVM are on the right. The processed frame in the first snapshot shows a moderate enhancement over the unprocessed scene. Note the non-linear dynamic range compression performed by the enhancement. The very dark areas are enhanced without severe blooming around the bright

Spectral Bands	Scales	C6416 CPU Load
1	1	31.24%
1	2	39.97%
1	3	51.22%
2	1	52.37%
2	2	71.48%
2	3	100.00%
3	1	73.77%
3	2	100.00%
3	3	100.00%

Table 5.9: C6416 CPU Loading for different Retinex configurations.

car lights. In the unprocessed frame of the second snapshot, the colors are nearly completely indeterminable and objects are becoming difficult to distinguish. The processed frame of the second snapshot retains most of the contrast and brightness of the first processed frame. Colors are still clearly perceptible and objects are still defined. For example, the vehicle that is nearly unseen in the unprocessed image is clearly seen in the processed image. The unprocessed frame of the third snapshot is almost completely dark. The processed frame of the third snapshot is nearly reaching the noise limit of the camera, but still provides significant information about the scene. Objects such as the wind tunnel spheres, that are not discernable in the unprocessed frame are clearly perceived in the processed frame.

The aim of our research was to achieve real-time multi-scale, multi-spectral Retinex image enhancement. We started by developing, implementing and analyzing several algorithm optimizations, and using the C6711 DSP we achieved 20.7 fps performance of the single-scale, monochrome Retinex. Building upon this effort, we continued to optimize and refine the algorithm and configuration of the architecture, and using the C6416 DSP we were able to achieve 20.3 fps performance of the multi-scale, multi-spectral Retinex.



Figure 5.5: First snapshot taken 40 seconds into the video recorded at NASA LaRC. The frame as captured by the camera is on the left and the real-time Retinex processed frame is on the right.

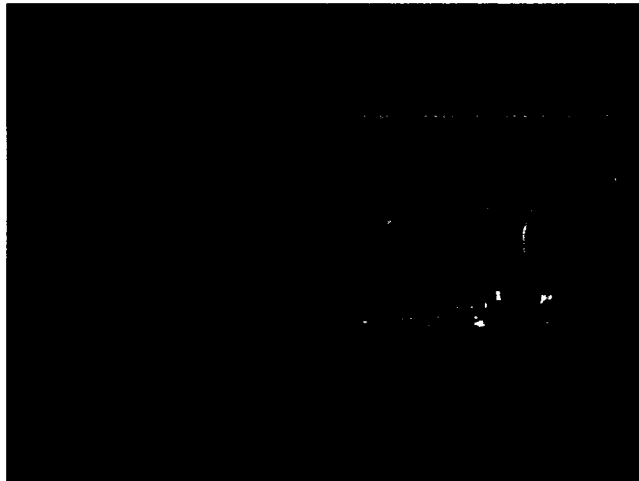


Figure 5.6: Second snapshot taken 6 minutes and 28 seconds into the video. Colors are nearly completely indeterminable and objects are difficult to distinguish in the unprocessed image. Colors and objects are still clear in the processed frame.



Figure 5.7: Third snapshot taken 14 minutes 28 seconds into the video. The only distinguishable object in the unprocessed frame is the tail-lights on the vehicle. Although noisy, the real-time Retinex processed image still clearly shows most of the major objects in the first snapshot including spheres, tree lines, and parked vehicles.

Chapter 6

Enhanced Vision System Case Study

6.1 Background

The real-time Retinex can be used to enable a wide variety of applications. We have chosen a NASA LaRC developed Enhanced Vision System (EVS) to demonstrate the performance of the real-time Retinex in an actual system. The EVS is a new aviation safety technology that is used to provide enhanced images of the flight environment to assist pilots flying in low visibility conditions such as rain, snow, fog, or haze [98]. During August and September of 2005, the EVS, and many other new technologies, were demonstrated during flight tests on the NASA 757 as part of the Follow-On Radar, Enhanced and Synthetic Vision Systems Integration Technology Evaluation (FORESITE) program.

The EVS contains a long-wave infrared (LWIR), a short-wave infrared (SWIR), and a visible-band camera, all mounted in an enclosure that is flown beneath a NASA 757 aircraft.

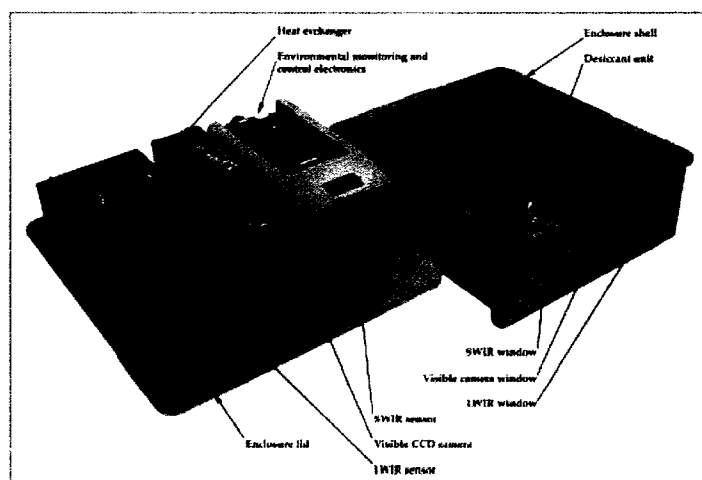


Figure 6.1: The EVS LWIR, SWIR, and visible-band cameras mounted to a baseplate, and the enclosure shell. Inaccurate bore-sighting can cause image registration problems.

Figure 6.1 shows the cameras mounted to a baseplate and the enclosure shell. Figure 6.2 shows the enclosure installed on the aircraft. Figure 6.3 shows the aircraft during a runway approach with the simulated shaded area depicting the field of view (FOV) of the cameras. The LWIR is a Lockheed Sanders LTC500 thermal imager and senses radiation in the 7.5–14 μm band. It can image background scenery, terrain features and obstacles at night and in other low visibility conditions. The SWIR is a Merlin Near-Infrared (NIR) camera that senses in the 0.9–1.68 μm region and is optimal for detecting peak radiance from runway and taxiway lights even in poor visibility conditions. The visible-band camera is a Bowtech BP-L3C-II CCD that detects the 0.4–0.78 μm band and covers imaging runway markings, skyline and city lights in good visibility conditions. A frame from each of the three video streams generated by the cameras in clear weather conditions is shown in Figure 6.4.

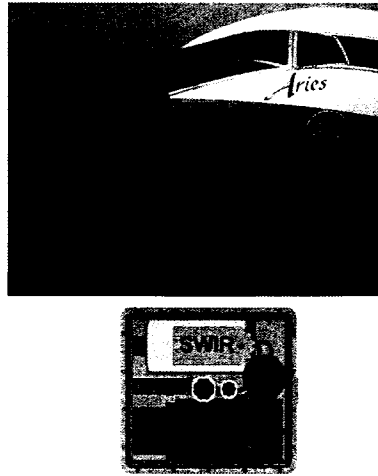


Figure 6.2: EVS camera enclosure mounted forward-looking underneath the NASA 757.

6.2 Image Processing Functions

The image processing architecture for the EVS is outlined in the top of Figure 6.5 [26]. The analog National Television System Committee (NTSC) RS-170 outputs of the SWIR and LWIR cameras are routed from the EVS camera enclosure (mounted beneath the NASA 757) to the processing board through a video distribution box. The processing board is situated in a pallet within the NASA 757 approximately 120 feet away from the EVS camera enclosure. Similarly, the digital RS422 outputs of the cameras are transferred to the processing board using optical fibers. We do not use these outputs, but for future implementations they may have a better signal-to-noise ratio than the analog outputs.

The functions performed by the processing components are shown in the bottom of Figure 6.5. The multi-spectral data streams from the EVS cameras must be resized, enhanced, registered, and fused into a single image stream. The images are resized into dimensions that are a power-of-two to fit the input requirement for the FFT (see Section 5.1.1). Methods

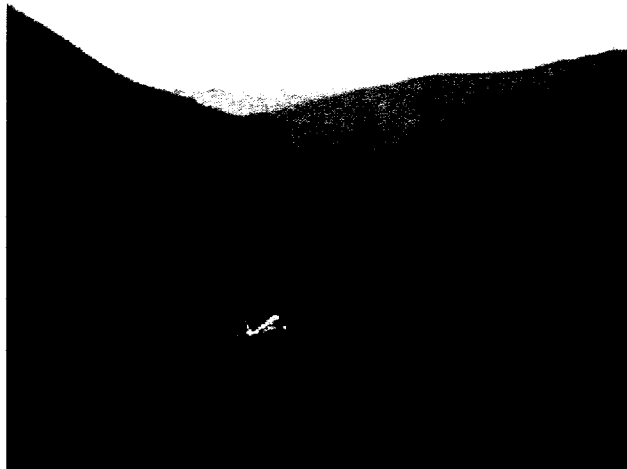


Figure 6.3: The EVS acquires data during the entire flight but take-off and landing phases are critical. The simulated shaded area depicts the field of view (FOV) of the cameras.

for resizing are discussed in Sections 6.3 and 6.4. Enhancement is performed to improve the information content of the images particularly in poor visibility conditions. For enhancement we use the real-time Retinex. The Retinex provides an ideal solution for enhancing EVS imagery because of its superb performance in improving low-contrast, dimly-lit images.

Registration is used to remove field of view (FOV) and spatial resolution differences between the cameras, and to correct bore-sighting inaccuracies [23]. Table 6.1 gives characteristics of the sensors that are relevant to registration. Registration is performed by first manually selecting a set of control points based on corresponding features in a LWIR and SWIR frame acquired at the same time. The control points are analyzed using multiple linear regression to approximate the coefficients of an affine transform which is applied to the LWIR image. The transformed image is then resampled using bilinear interpolation to align the registered LWIR image data to the same grid as the reference SWIR image. The same transform can then be used on all other LWIR frames since the optical parameters

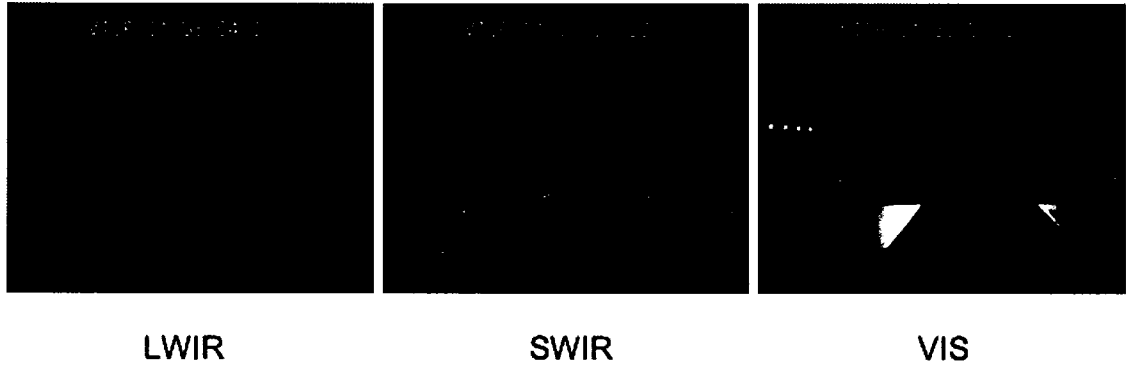


Figure 6.4: Examples of the imagery generated by each camera in good weather conditions. The images from cameras must be registered, enhanced, fused and displayed to the pilot in real-time.

and the camera alignment are assumed to remain constant during flight. Appendix A gives a more detailed discussion of the registration procedure.

	SWIR	LWIR	CCD
Image Dimensions (pixels)	$320H \times 240V$	$320H \times 240V$	$542H \times 497V$
Optics FOV	$34^\circ H \times 25^\circ V$	$39^\circ H \times 29^\circ V$	$34^\circ H \times 25^\circ V$
Detector Readout Frame Rate	60 Hz	60 Hz	30 Hz (interlaced)

Table 6.1: Sensor Specifications

The two enhanced and registered video streams from the SWIR and LWIR cameras are then fused into a composite video stream that contains more information than either input spectral band. This also provides the additional benefit of producing a single output to observe instead of multiple images from multiple video sources. The Retinex could be used as a fusion engine for this application since the algorithm performs nearly symmetric processing on multi-spectral data. Multiple camera inputs could be distributed onto these multi-spectral processing chains and fused using the weighting and summation properties of the Retinex [57]. However, for EVS processing the image streams are fused by effectively

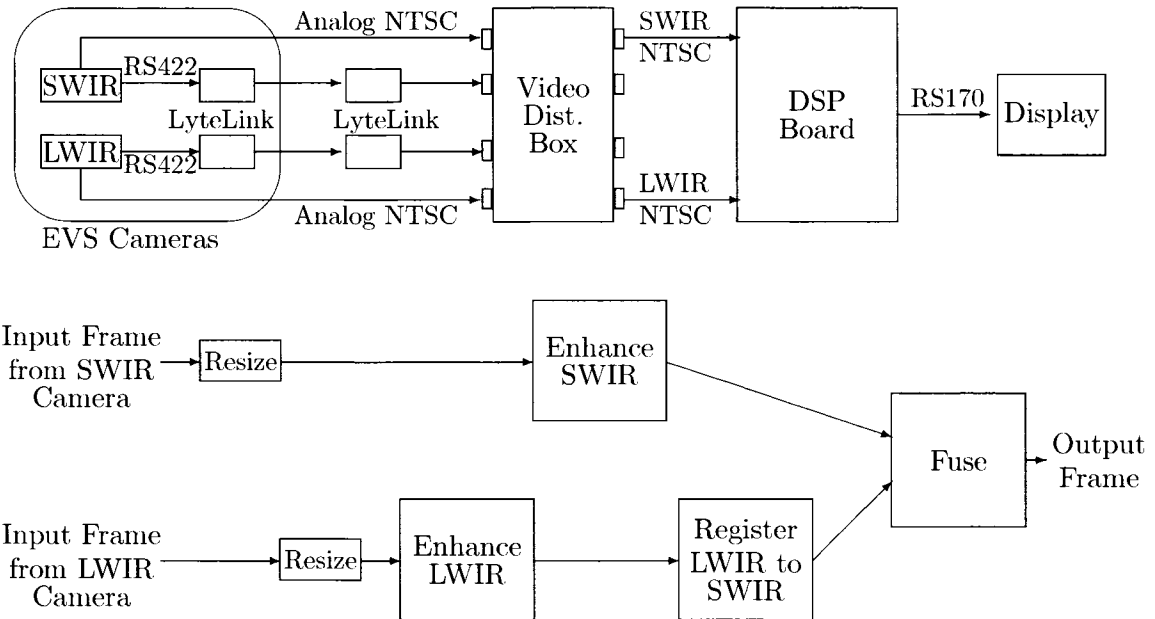


Figure 6.5: Image processing architecture and functions of the EVS. Analog NTSC camera outputs are currently processed. The SWIR data is used as the baseline for registration since it has the smallest field of view.

performing a weighted sum of the two processed outputs since a different Retinex is applied to each channel. Pixels are summed on an inter-frame basis. Other methods such as interleaving frames or fields causes sever flicker. The fused data stream is output as a standard composite NTSC signal into a display.

6.3 Additional Requirements

Several other EVS parameters complete our baseline requirements and constraints for real-time Retinex processing. First, our initial performance goal is to achieve a display rate of 15 fps, instead of the de facto standard of 30 fps for real-time video. We can use this reduced rate because our final processed output will be sent to a pilot's display and several human factor studies have shown that an update rate of 15 fps is more than sufficient to avoid flicker,

accurately portray motion [100], and not cause pilot induced oscillations (PIOs) [34, 40] of the aircraft. A PIO can occur when a pilot views and reacts to an instrument or display that is updated too slowly (< 12 fps) [2, 44].

Second, the cockpit displays are low-resolution (320×240). This significantly reduces the amount of image data that must be processed. Fitting the closest power-of-two input requirement for the FFT to this frame size dictates that we process a 256×256 portion of each frame. Only 20 percent of the horizontal component of the image is lost and the vertical component is zero-padded to fill 256 pixels.

Third, only the SWIR and LWIR cameras are targeted for processing by the current EVS sponsors. The visible band camera is only used to provide context. Use of the visible band data in conjunction with the infrared cameras to improve the information provided to a pilot is an open research topic. For now, processing only the monochrome SWIR and LWIR cameras reduces the number of bands that have to be processed from 5 (1 each for the LWIR and SWIR, and 3 for the visible-band camera) to 2. Only processing the SWIR and LWIR cameras also enables the use of the SSMR version of the Retinex since it provides good enhancement of single-band infrared imagery with the additional benefit of minimizing computational requirements.

Several environmental parameters are defined for the EVS. The space allocated is approximately 17 wide by 8 inches deep by 3 inches high. This is enough space to hold a standard PCI board, thus allowing a board-level (vs. chip-level) solution, but eliminates multiple board or cluster solutions. The operational temperature range falls within the standard commercial temperature range of 0 to 70 degrees C. The maximum power allocated for image processing is approximately 5 watts with a standard input voltage of 5

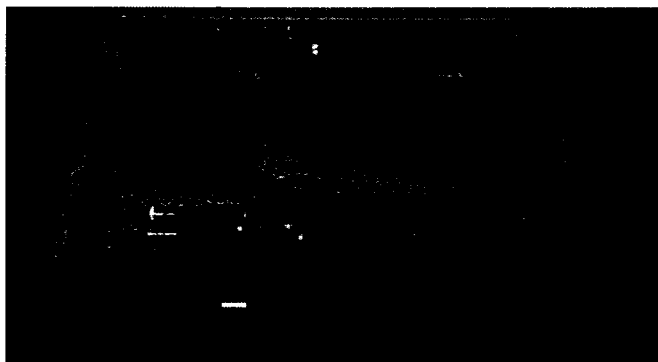


Figure 6.6: DM642 EVM, signal splitter boards, and power supply in flight box.



Figure 6.7: Flight box in flight pallet on NASA 757.

volts and current limited to 1 amp. If other input voltages are required DC/DC converters can be used within the space allocated. Waivers for additional power can also be requested since the NASA 757 has many power resources. However, general aviation aircraft have significantly fewer resources and it is beneficial to limit our resource allocation for potential use in these environments also.

Each EVM discussed in Section 4.1 easily fits within the physical constraints of the EVS, however only the DM642 EVM has two video inputs to accept the two infrared camera outputs. The DM642 EVM was flight hardened¹ and the board was encased in a rack-mountable box with interfaces and switches extended to the front and rear panels. A power supply and signal break-out cards are also enclosed in the box. Figure 6.6 shows the DM642 EVM and other devices in the flight box and Figure 6.7 shows the box in the flight pallet on the NASA 757.

A new method to update parameters was developed for the DM642 EVM because a host PC with a JTAG emulator was not available for continuous use during flight test to perform RTDX based updates. Instead of using the JTAG port, our new method uses the Ethernet

¹Flight-hardening means that components are secured to prevent being shaken loose during flight.

port on the DM642 for communication with an external PC thus eliminating the need for a JTAG emulator. A new task was written to process messages received via Ethernet using the mailbox module in DSP/BIOS. The mailbox module provides a set of functions that are used to pass synchronized messages from one task to another on the same processor. In this case, the parameter update messages are passed from the Ethernet task to the main frame processing task discussed in Section 4.6.

6.4 Results

The EVS was tested during FORESITE flight demonstrations in August and September of 2005. All flights were performed in good weather and although this was not ideal for testing the performance of the EVS, this still enabled a thorough evaluation of the functionality of the EVS components including the real-time Retinex. As mentioned in Section 6.2 we have to individually resize and enhance the monochrome output images of the SWIR and LWIR cameras, register the LWIR to the SWIR, and then fuse the two channels together. Since both cameras are flown upside-down underneath the NASA 757, the images must be rotated 180° for normal viewing. This is usually performed using embedded routines in the cameras but unfortunately, the camera integrators were unable to rotate and place the corresponding gamma look-up tables in ROM for the LWIR camera. We decided to perform the rotation of the LWIR image within our image processing routines on the DSP. We modified our Retinex routine to read in the LWIR image data starting at the end of the image data and proceeding to the first pixel. This causes a 180° rotation of the image. Our sequence of tasks is as follows:

- resize the LWIR input image to 256×256 pixels,
- Rotate and Retinex the LWIR image,
- resize the SWIR input image to 256×256 pixels,
- Retinex the SWIR image,
- register the enhanced LWIR image to the enhanced SWIR image,
- interpolate the LWIR image to the SWIR grid,
- fuse and output the final processed image.

Higher quality imagery is achieved by enhancing the LWIR image before performing registration, instead of registering first, since registration may eliminate part of the original image when it is transformed.

Our algorithm performed the above sequence of tasks on the DM642 at 33.89 fps. Sample input frames² from the SWIR and LWIR cameras are shown in Figure 6.8 and Figure 6.9, respectively. The LWIR input is actually received from the LWIR camera rotated (upside down) 180° , but is shown right-side up for viewing purposes. The same SWIR frame after SSMR enhancement is shown in Figure 6.10. It is easy to see the improved contrast and brightness in the image. Similarly, a frame of the enhanced and registered LWIR channel is shown in Figure 6.11. Registration can be seen by noting the large vertical shift downward at the top of the image. Both of the SWIR and LWIR enhanced frames shown are captured as intermediate results for demonstration purposes and not the final output product of our

²We wrote a small utility to send image data from the DSP to the host to capture frames at various stages of processing.

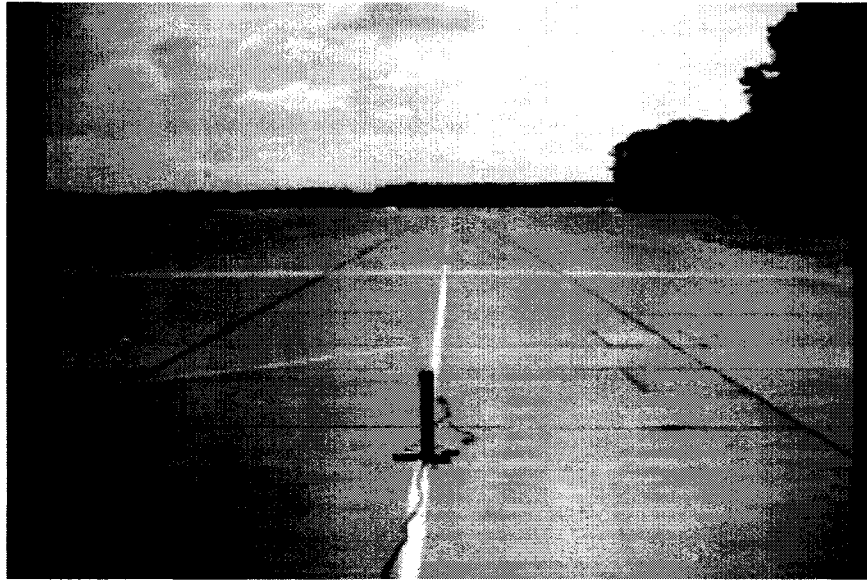


Figure 6.8: A frame from the EVS SWIR camera before processing. The faint vertical lines were part of the input image and probably caused by subsampling in the video distribution system.

processing. The final fused output is shown in Figure 6.12. This image has significantly better contrast, brightness, and sharpness than any of the original inputs, and provides a single enhanced output for the pilot to view. Enhancement and registration parameters were determined empirically.

Our fused output image is actually a 512×512 image, but we are only processing 256×256 pixels per image. The CCD arrays for both the imagers are approximately 320×240 pixels, but the NTSC composite inputs received are upsampled to 640×480 through pixel replication (horizontally) and line duplication (vertically). We used this information and modified our core Retinex routine to generate a 512×512 image by 2:1 subsampling the horizontal and vertical components of our input images. This process retains the majority of the original resolution of the cameras.

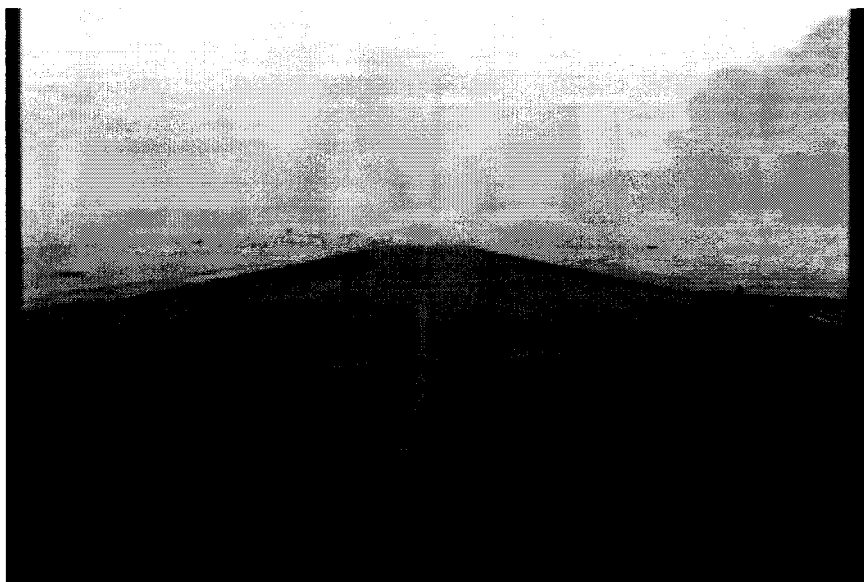


Figure 6.9: A frame from the EVS LWIR camera before processing. The LWIR camera output is actually rotated 180° from what is shown.

An additional interesting addendum to this process was the requirement to store the algorithm in non-volatile flash memory so that the algorithm would automatically execute at system power-up. As discussed earlier, an Ethernet client was added to the code to facilitate communication with a host to update Retinex parameters. This expanded the size of the executable beyond the flash page boundary so we developed a new multi-page bootloader algorithm to implement this feature. Development of this algorithm is discussed in Appendix C. This information will be used in a new TI application report on bootloaders for their C6X processors.

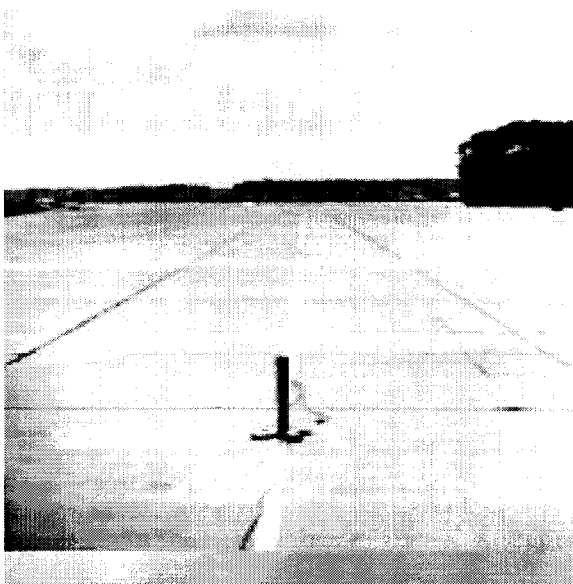


Figure 6.10: SWIR frame after enhancement.



Figure 6.11: LWIR frame after enhancement and registration to the SWIR image.

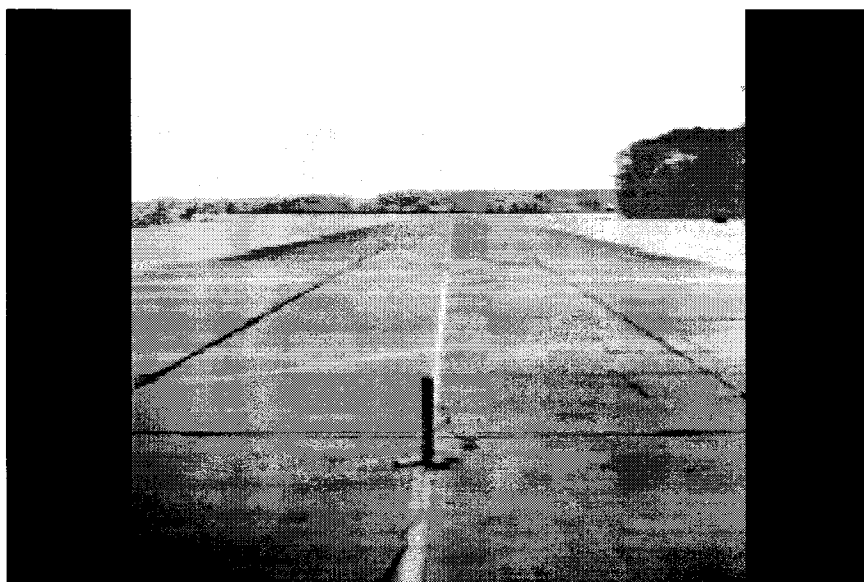


Figure 6.12: Enhanced, registered and fused output image.

Chapter 7

Future Research

As with most research topics, there is always the question of “how can we make it better?”. For our real-time Retinex, improvements can basically be categorized as (1) increasing the performance of the algorithm on DSPs to provide 30 fps MSR performance, (2) processing larger format images, and (3) migrating to a multi-processor environment. It would also be beneficial to integrate additions that augment the Retinex, such as color restoration or white balance techniques, into our real-time version of the algorithm, but the three primary areas listed above should be solved first.

7.1 Luma-only Retinex

Before addressing these issues we briefly digress to discuss a method that can immediately provide a near Retinex quality enhancement at full 30 fps performance for certain applications. This alternative version of the Retinex is called the luma-only Retinex (LOR). In the LOR algorithm, only the luma, Y' , component of an image is processed. The chroma components are left unchanged and passed directly from input to output. The enhancement

quality of this algorithm is very good because the majority of spatial detail is contained in the luma component of an image.

Processing only the luma component eliminates all of the $Y'C_B C_R$ to RGB input and output conversions. As discussed in Section 4, the DM642 EVM stores image data in planar form and the C6416 EVM stores image interleaved. Thus for the DM642 only a pointer to the Y' component is required to access the input and to generate the output. The Y' must still be extracted from the input image data and embedded in the output data on the C6416, but this is performed very efficiently. Since only the Y' component is processed using three scales, the performance is analogous to that shown for the DM642 and C6416 for 1 band and 3 scales — 38 fps for the DM642 at 200 MHz and 58 fps on the C6416.

7.2 Improving Current Performance

To improve our full, real-time Retinex to meet 30 fps performance on DSPs would require moderate speed-ups in processor performance (by ≈ 33 percent) and either a similar speed-up in EMIF bus rates and external memory access times or a L2 memory large enough to remove at least some of the DMA requirements. Our final algorithm execution time is driven by processor cycles, not I/O bandwidth. However, as we showed in migrating our code from the C6711 to the C6713, when processor clock speed is increased, the I/O bandwidth needs to improve also or it will become the bottleneck on performance. Having a larger L2 memory and placing larger segments of data there implicitly improves I/O bandwidth because it removes the requirement to transfer that particular data. We demonstrated this by keeping the logarithm of the input image in the L2 memory on the C6416.

Extending this concept, to store the FFT of a color input image requires 1.5-MBytes of memory with 512-KBytes per band. Retaining our current technique of storing the image logarithm data locally implies that we would need a L2 memory size of between 2 and 3-MBytes, well within the reach of next generation DSPs. Storing the FFT in L2 memory would eliminate the transfer of the FFT of each row to external memory, and the 2-dimension read of external memory to form the column data. Just looking at the rows and ignoring function call overhead, theoretically to DMA a row requires 1.92 microseconds through a 64-bit wide EMIF bus clocked at 133 MHz. To DMA 256 rows requires 4.93 ms. Measured 2-D transfer times were on the order of twice the row transfer time or ≈ 10 ms. Placing the FFT data in L2 memory would not directly achieve a 15 ms increase in performance because these transfer are currently performed in the background. However it does mean that any additional processor improvements would then be immediately effective, thus with a commensurate increase in processor performance, 30 fps MSR would be easily achievable. An alternative idea is to attempt to store all of the convolved data, but this would require 512-KBytes per scale per band equating to 4.5-MBytes for the MSR. Local DSP memories on this order are probably years away.

7.3 Processing Larger Format Images

Processing larger format images exacerbates the issues address above. First, significantly more processing cycles are required. Using the calculation of the FFT as an example, Table 7.1 shows the number of cycles and the associated processing time for different FFT sizes executing on the C6711, DM642, and the C6416 processors. The benchmark equation

for the out-of-place cache-optimized mixed radix FFT executing on the C6711 originally given in Section 5.1.9 is repeated here as

$$C = (3\lceil\log_4(n-1)\rceil n) + (21\lceil\log_4(n-1)\rceil + (2n) + 44. \quad (7.1)$$

and the benchmark equation for an extended-precision, mixed radix 16×32 FFT with rounding, and digit reversal executing on the DM642 and C6416 is repeated here as

$$C = (13n/8 + 24)(\lceil\log_4(n) - 1\rceil) + (n + 8)1.5 + 27. \quad (7.2)$$

The processing time for FFTs ranging in size from 256 to 2048 are shown in Table 7.1.

Referencing the information in Table 7.1, Table 7.2 gives benchmark FFT performance

FFT Benchmarks	C6711 @ 150 MHz		DM642 @ 720 MHz		C6416 @ 1 GHz	
FFT Size	cycles	μs	cycles	μs	cycles	μs
256	2923	19.49	1743	2.42	1743	1.74
512	7296	48.64	4231	5.88	4231	4.23
1024	14464	96.43	8327	11.56	8327	8.33
2048	34965	233.10	19871	27.60	19871	19.88

Table 7.1: FFT Benchmarks for C6711, DM642 and C6416.

values for various input image sizes. As can be seen from this data, to perform a 512×256 FFT on the C6711 takes nearly the full 33.33 ms time allotted to process a frame at 30 fps. Initially looking at the data the DM642 and C6416 perform significantly better and seem to be potential solutions for 512×512 sized images. However, this is only the forward FFT of the input image. Subsequently three inverse FFTs (one for each scale) must be performed for each band, each taking the same time as the forward FFT. This drives the FFT processing time for a 512×512 image to 60.02 for the DM642 and 43.33 ms for the

FFT Image Benchmarks		C6711 @ 150 MHz	DM642 @ 720 MHz	C6416 @ 1 GHz
x-dim	y-dim	ms	ms	ms
256	256	9.98	1.23	0.89
512	256	29.89	3.63	2.61
512	512	49.81	6.02	4.33
1024	512	123.64	14.85	10.7
1024	1024	197.48	23.67	17.06
2048	1024	576.13	68.36	49.24
2048	2048	954.78	113.04	81.43

Table 7.2: FFT Processing Time Benchmarks using C6711 and DM642 for various sized images.

C6416 exceeding the 33.33 ms boundary. Again, this is just the time to process FFTs, other computations must be included to perform Retinex enhancement.

The second issue that the size of the images are considerably larger, thus requiring more memory for storage and more bandwidth for transfers. Table 7.3 shows typical FFT storage requirements and 64-bit, 133 MHz EMIF transfer times for various sized images. As shown, the FFT of a 512×512 image requires 2-MBytes for storage eliminating any possibility of keeping this data in current DSP L2 memory.

FFT Image Size		Memory Requirement	EMIF Transfer Time
x-dim	y-dim	MBytes	ms
256	256	0.5	4.93
512	256	1	9.86
512	512	2	19.72
1024	512	4	39.44
1024	1024	8	78.88
2048	1024	16	157.76
2048	2048	32	315.52

Table 7.3: FFT storage requirements and transfer times (based on row oriented data) for various sized images. Storage is based on complex image data stored as integers. Transfer times are based on a 64-bit EMIF bus clocked at 133 MHz.

Incremental increases in performance could also be achieved by modifying the FFT. Since

our image signal is real-valued we could use the imaginary part of the FFT input and exploit the symmetry of the frequency spectrum to compute either a $2N$ -point sequence using an N -point FFT or compute two, N -point FFTs simultaneously [7, 75, 63]. This technique can perform the FFTs $\approx 30\text{--}40\%$ faster [63] than the conventional method, but the overhead associated with interlacing the input and unscrambling the output reduces the effectiveness of this method. The FFT routine currently used could be rewritten to take advantage of alternative fast bit-reversal techniques such as those introduced by Zhang [104]. Pitas and Strintzis [52] discuss an interesting method to build up the column transform in steps while selectively processing rows to reduce the I/O operations between hard disk and internal memory. Although hard disk access is several orders of magnitude slower than external-to-internal memory transfers, an adaptation of this method could be used for external-to-L2 memory transfers.

There is no fundamental reason why we have to use the row-column method to decompose the 2-D DFT. We could possibly reduce the number of arithmetic operations performed by using other algorithms such as a vector-radix Fast Fourier algorithm [22], a polynomial transform FFT [48], or a fast 2-D Hartley transform [5]. Other techniques, such as using fast matrix transposition methods to reduce the number of I/O operations [16, 15], could also be explored. While all of these methods are worthwhile, revolutionary increases in performance will probably only be addressed through using alternative processing platforms.

7.4 Migrating to a Multiprocessor Environment

Another strategy is to map the MSR algorithm into a multiprocessor system [35, 6] and take advantage of the parallelism of the algorithm. Most multiprocessor systems in general, exceed our initial constraint of performing the MSR on a small, embeddable, low-power system. However as newer technologies emerge this may become a viable alternative. Even today, several relatively small multiprocessor boards are available from vendors such as MangoDSP, Sundance or Vitecmm.

A system that completely distributes the primary tasks of the MSR could resemble a design similar to that in Figure 7.1. The first level task splits the input image into its RGB spectral components. The next two levels perform forward row and column transforms, respectively. The output of this level is fed into three other tasks, each performing convolution of the now spatial frequency domain image data with the associated kernel. The next two levels perform inverse FFTs of the columns and rows respectively, for each convolved output. The next level combines the data for each scale, computes the log and subtracts this from the log of the original image. The final task combines the processed data from each band. Each task could be mapped to an individual processor or assigned to a pool of processors. Similar to our EMIF bus bandwidth issues, interprocessor communication and data sharing will need to be carefully balanced. The processors used to perform these tasks could be DSPs, FPGAs (see Appendix B), or a mixture of both. In a heterogeneous system, FPGAs could perform pre- and post-processing tasks, while DSPs perform the core FFTs and convolutions. In this dissertation we have established a core set of techniques that could easily be used to implement the Retinex in this multiprocessing environment.

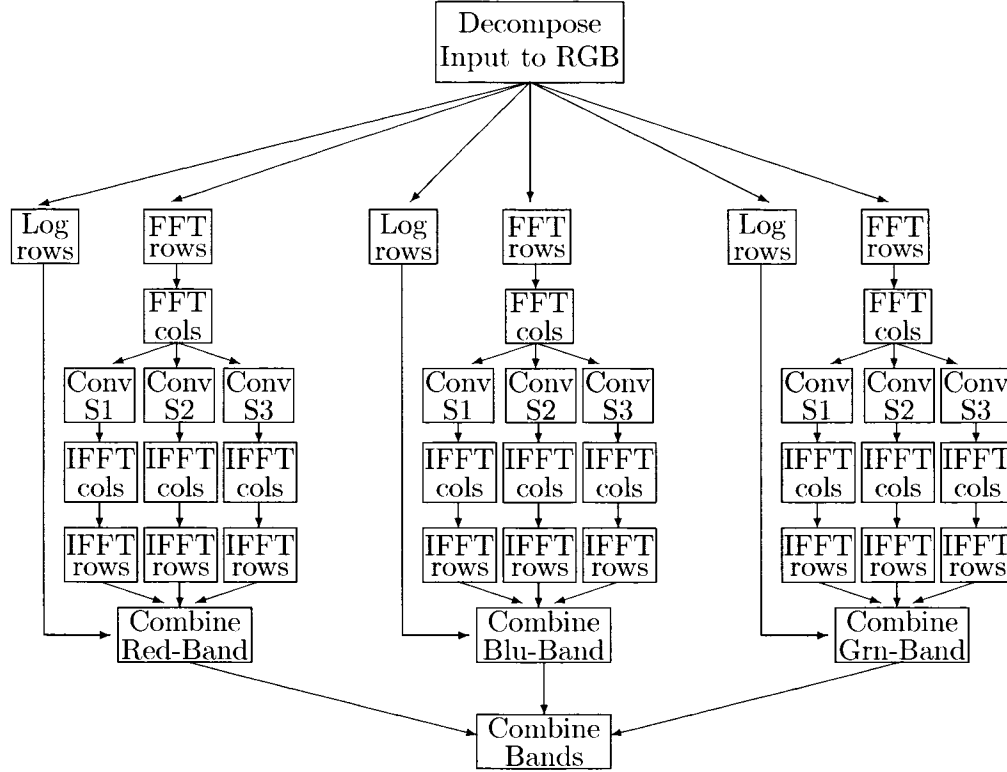


Figure 7.1: Data flow diagram of MSR tasks

Ultimately, it would be beneficial to develop an embeddable single chip-level implementation of the processing components of the algorithm. We would start by using the techniques we developed to place the MSR tasks described above into one or more FPGAs. Commercial tools are available from companies such as Celoxica, Accelchip, and Catalytic, that automatically convert C code developed for DSPs into VHDL, the current language of choice for FPGAs, and multi-FPGA boards are available from companies such as Sundance and Nallatech. Implementation in an FPGA would enable the full customization of our design and a direct migration path to an ASIC.

Chapter 8

Conclusions

In the last few years the multi-spectral, multi-scale Retinex has provided outstanding image enhancement of still imagery for numerous users. Literally thousands of images have been processed. The first versions of the multi-scale, color Retinex, coded on a Windows NT 200 MHz Pentium Pro PC and processing a 512×512 image, executed in ≈ 45 seconds — more than three orders of magnitude slower than required for real-time performance. Current PC implementations of the Retinex for a 512×512 image execute in ≈ 3 seconds, still two orders of magnitude too slow to be considered for real-time applications. It was my thesis that a real-time, 15 fps multi-scale, multi-spectral Retinex could be achieved on a single-processor embedded system through proper algorithm and architecture optimization. The summation of this dissertation is that we have successfully achieved this goal.

Throughout this research a series of optimizations were developed, investigated, and implemented on progressively faster DSPs, each with more capability. These techniques were discussed in Chapter Five. We began by focusing on the single-scale monochromatic Retinex targeting the floating-point C6711 and C6713 DSPs and achieved 20.7 fps and 28

fps performance, respectively. Although the C671Xs platforms did not allow us to obtain real-time MSR performance, the core algorithm structures and techniques developed for them, such as merging algorithm components to reduce I/O and performing effective DMA routines, were used repeatedly in future implementations. We then changed our hardware target to the fixed-point DM642 DSP. After modifying our single scale Retinex design into a fixed-point implementation and adding additional optimization techniques, we obtained 69 fps performance on this platform.

Using the knowledge gained from our previous experiences, we focused our research on the more computationally intensive multi-scale Retinex, while continuing to target the DM642 DSP and adding the more powerful C6416. We again developed and implemented additional optimizations into our core algorithm focusing on constructs specific to multi-scale, multi-band processing and taking advantage of the additional resources within the processors. This includes restructuring the mathematics of the algorithm to enable exploiting the pre-computation of additional parameters and modifying our buffering scheme to keep DMA processes from driving the algorithm computation time. Our best performance on the most computational intensive version of the Retinex (the MSR) was 20.25 fps using the C6416 platform. This exceeded our baseline target of 15 fps but still requires further exploration to meet 30 fps.

We applied our real-time algorithm in actual flight hardware during demonstrations at NASA LaRC enhancing, registering, and fusing two infrared video camera outputs. This was a significant achievement however, the accomplishment of this research extends beyond this one application. It provides a new tool for image enhancement to a broad range of users and will provide the basis for further academic research. Future implementations can

use the core techniques we have developed and demonstrated and will hopefully achieve even better performance through the use of multi-processor systems, FPGAs, or ASICs.

Appendix A

Multi-Image Registration

Coupling infrared sensors with visible band sensors — for frame of reference or for additional spectral information — and properly processing the multiple information streams has the potential to provide valuable information in night and/or poor visibility conditions. In Chapter 6, we discussed an EVS that is being developed to test this concept. A set of images consisting of an image from each of the cameras of the EVS taken during one time-aligned frame is fused into a single image that contains more information than any individual spectral band. This process is then repeated for all the image frames making up a video sequence. To properly perform fusion it is critical to ensure that the information from each sensor refers to the same features in the environment [8, 43]. The different sensors of the EVS have different acquisition lattices and optics, therefore they capture information in data structures that are substantially different from each other. Thus, the images must first be registered before any fusion is performed. Several authors have addressed image registration problems with innovative, but often complex, general solutions [42, 60, 41]. In this appendix, we describe two straightforward solutions for registering EVS images.

A.1 Background

Image registration is the task of aligning images taken at different times, from different sensors, or from different viewpoints so that all corresponding points in the images match. A transform must be defined that relates the points in one image to their corresponding points in another. This transform depends upon the characteristics of the differences between the images being registered, and is computed with respect to a reference or baseline image. The images that are to be matched to the reference are called the sensed, or, distorted image.

More particularly, image registration is defined as a mapping between two or more images both spatially (geometrically) and with respect to intensity. Expressed mathematically we have:

$$I_2 = g(I_1(f(x_1, x_2))), \quad (\text{A.1})$$

where I_1 and I_2 are two-dimensional images (indexed by x_1, x_2), $f : (x_1, x_2) \rightarrow (x'_1, x'_2)$ maps the indices of the distorted frame to match those of the reference frame, and g is a one-dimensional intensity or radiometric transform [9]. We assume that we do not need to make any radiometric adjustments, so $g = \mathcal{I}$, the identity transform. Hence we are concerned only with the spatial transformation, f . In generating a spatial transform for the EVS, our primary difficulty is the lack of fiducial markers within the images generated by the EVS sensors. The cameras are, however, assumed to be bore-sighted so they are expected to have a common center of alignment. The spatial transform should, then, properly align the images, but should not affect any characteristic differences that should be exposed by registration.

Spatial transforms may take on different forms depending upon the application. Simple, common transforms specified by analytic expressions include rigid-body, affine, projective or perspective, and polynomial [59, 47]. The distortions between the images of the EVS in general seem constrained to those correctable by affine transforms. They also appear to be characterizable by a global (versus local [21]) transform where a single transform correctly maps all the points on the distorted image to match the corresponding points on the reference image. An affine transform fulfills the requirements for the needed transform.

An affine transform can perform rotation, translation, scaling and shearing operations. It offers six degrees of freedom when selecting six unknown coefficients and solving a system of six linear equations. In general, it can perform triangle-to-triangle mappings. A general representation of an affine transform is $[y_1, y_2, 1] = [x_1, x_2, 1]T$ where

$$T = \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{bmatrix}, \quad (\text{A.2})$$

x_1 and x_2 reference the input coordinate system, y_1 and y_2 reference the output coordinate system, and a_{ij} are transform coefficients [102].

The forward mapping functions are

$$y_1 = a_{11}x_1 + a_{21}x_2 + a_{31} \text{ and} \quad (\text{A.3})$$

$$y_2 = a_{12}x_1 + a_{22}x_2 + a_{32}. \quad (\text{A.4})$$

Geometric, image-to-image registration can be summarized in three general steps:

1. Feature identification and matching is performed to establish a correspondence between features in the distorted image to those in the reference image;

2. A spatial transformation is selected and the transformation coefficients are computed based upon the feature matching criteria;
3. The distorted image is inverse-mapped using the computed transformation and re-sampled to register it with the reference image.

Feature identification and matching are often performed by selecting pixel locations called control points. Identification of control points can be accomplished in several different ways [18, 21]. Manual identification of control points is commonly performed. The images are displayed, normally side-by-side, and corresponding points usually based on features such as lines, edges, or contours are selected from both images.

The spatial transform coefficients that represent the unknown image distortions are determined from the control points. A minimum of three non-collinear control points are required to determine the six unknown coefficients of an affine transformation. Wolberg and Jensen [102, 31] describe several techniques to solve for unknown coefficients including pseudo-inverse solutions, least squares with ordinary and orthogonal polynomials, and weighted least squares with orthogonal polynomials.

Image resampling is the process of transforming a sampled image from one (input pixel grid) coordinate system to another (output pixel grid), where a sampled image is the digitization of the spatial coordinates of an image function $f(y_1, y_2)$ — a two-dimensional intensity function [102, 13, 20]. The two coordinate systems are related to each other by the mapping function of a spatial transformation.

To perform image resampling, initially, the output pixels are inverse mapped using the transformation function to a new grid which (usually) doesn't correspond to the input

grid. Thus an interpolation (image reconstruction) procedure is used to generate a continuous surface through the samples of the new grid. Then the input image is sampled (digitized) at these points to provide the discrete output pixel values of the process. Three common methods of interpolation are nearest neighbor, bilinear, and parametric cubic convolution [102, 50].

Table A.1 shows the relevant manufacturer characteristics of the sensors [98]. The images from the three sensors obviously need to be registered because of the differences in these characteristics. The solutions developed to resolve these differences are discussed in Section A.2.

	SWIR	LWIR	CCD
Image Dimensions (pixels)	$320H \times 240V$	$320H \times 240V$	$542H \times 497V$
Optics FOV	$34^\circ H \times 25^\circ V$	$39^\circ H \times 29^\circ V$	$34^\circ H \times 25^\circ V$
Detector Readout Frame Rate	60Hz (typical)	60Hz	30Hz (interlaced)

Table A.1: Sensor Specifications

The characteristics of the actual images obtained for registration differ from the initial manufacturer specifications because of data acquisition and storage to tape. First, all images have a nominal image size of 640 x 480 pixels corresponding to the NTSC format of the recorded images. However, the actual size of the images is quite different after the images are cropped so that the FOVs match the “visible” part of the images (see Section A.2). Second, ground test measurements of the cameras’ FOVs differed from the manufacturer provided values. These updated characteristics are shown in Table A.2, and need to be included in the computations for proper registration of the data streams.

The algorithms operate on a set of three, time-aligned images where each image is acquired by an individual camera of the EVS. Each of the video streams is recorded, or

	SWIR	LWIR	CCD
Image Dimensions (pixels)	$640H \times 480V$	$640H \times 480V$	$640H \times 480V$
Optics FOV	$31.5^\circ H \times 23.5^\circ V$	$41^\circ H \times 30.75^\circ V$	$33.5^\circ H \times 25^\circ V$
Detector Readout Frame Rate	60Hz (typical)	60Hz	30Hz (interlaced)

Table A.2: Updated Sensor Specifications

post-processed, with video timecode information in each frame. The frames are time-aligned simply by finding the frames with matching time codes. This set of time-aligned frames is then used to obtain the registration parameters with respect to the baseline frame. All other frames of the video sequence can be processed with the same parameters. Each frame, including the ones from the color CCD sensor, is converted to grayscale before registration and further processing.

A.2 Registration algorithms

Our first solution for image registration is based solely on camera sensor specifications. The cameras were assumed to be properly bore-sighted at installation thus the only distortion parameters to account for in registration are the differences in FOVs and resolutions. This algorithm, called the SS (sensor specifications) algorithm, performs registration by first equalizing the FOVs and then resampling the distorted image to match reference resolutions. Based upon the lessons learned from the SS algorithm, a geometric image-to-image registration algorithm was implemented. Both of these algorithms are discussed below. For each of the algorithms, we use the SWIR image as the baseline since it has the “worst” image parameters (the smallest FOV and poorest spatial resolution). The size of an image can be modified through interpolation but we cannot increase the FOV.

A.2.1 SS algorithm

The first step of the SS algorithm is to equalize the instantaneous FOVs (IFOV)s of the sensors. The FOV is the angular extent of the full image on the sensor and the IFOV is the angular extent on an individual detector element, i.e., the solid angle through which a detector element is sensitive to radiation.

From Figures A.1, A.2, and A.3, we observe that the visible portion of the images is actually smaller than the full image capture window. The FOVs listed in Table A.2 are assumed to correspond to the visible portion and not the capture window. Thus, the first stage of processing is to crop the images to the visible portions. The second stage of processing is to ensure that the two images are representing the same portion of the scene. Since the FOVs of the SWIR and the LWIR sensors differ — LWIR has the greater FOV and hence captures a wider swath of the scene — the LWIR image needs to be cropped so that it encompasses the same FOV as that encompassed in the SWIR image. The dimensions of the cropped LWIR images — the number of columns and rows — are determined by a simple scaling operation. The horizontal and vertical IFOVs of the LWIR image are obtained using

$$\text{IFOV-LWIR-HORIZONTAL} = \frac{\text{FOV-LWIR-HORIZONTAL}}{\text{LWIR-COLS}} \quad (\text{A.5})$$

and

$$\text{IFOV-LWIR-VERTICAL} = \frac{\text{FOV-LWIR-VERTICAL}}{\text{LWIR-ROWS}}, \quad (\text{A.6})$$

respectively. The number of cropped columns and rows for the LWIR image is then deter-

mined by

$$\mathbf{columns} = \frac{\text{FOV-SWIR-HORIZONTAL}}{\text{IFOV-LWIR-HORIZONTAL}} \quad (\text{A.7})$$

$$\mathbf{rows} = \frac{\text{FOV-SWIR-VERTICAL}}{\text{IFOV-LWIR-VERTICAL}}. \quad (\text{A.8})$$

After cropping, the SWIR and LWIR FOVs are equal, but since the dimensions of the cropped LWIR are different from the dimensions of the SWIR, the IFOVs of the LWIR and the SWIR images are still different. To make the IFOVs the same we must resample the cropped LWIR image so that it is the same size as the SWIR image. This entails: (1) computing an expansion factor that will make the dimensions of the cropped LWIR image greater than the dimensions of the SWIR (2) pixel replicating the cropped LWIR based on the expansion factor and (3) downsampling the expanded LWIR image to the SWIR dimensions. We use the bi-linear interpolation method [10]. Nearest neighbor interpolation can also be selected if desired but bilinear interpolation is more spatially accurate and results in images that are slightly smoother.

A similar sequence of operations is performed between the SWIR image and the visible image. If the FOVs are the same, as in Table A.1 then the visible image is simply down-sampled to match the SWIR resolution. The initial results from the SS algorithm clearly indicated that the distortions present in the images were not excessive, but they also were not limited to FOV and resolution differences.

A.2.2 MLR algorithm

Based on the results obtained from the SS algorithm a more general, geometric image to image registration algorithm is implemented. The distortions between the images seem to be due to sensor translation, (slight) rotation, scale change, and, possibly, shear. An affine

transform is, thus, used to model the spatial transformation. Control points are manually selected for identifying and matching corresponding features between the reference and distorted images. Since we assume that the sensors do not change alignment over time, we only need to register one baseline set of images that can subsequently be used for the rest of the image frames.

We use point mapping without feedback [9] to approximate the global affine transformation. The first stage of the MLR algorithm is to select a minimum of three non-collinear control points from two input images. More points can be chosen to make the coefficients more representative of the distortions throughout the overall image if the points are well distributed. Global distortion representation is also improved by choosing pixels on the perimeter if possible. The control points are then analyzed using multiple linear regression [65, 101] to approximate the coefficients of the affine transform. Residuals to determine the accuracy of the regression model obtained are calculated. The defined affine transform provides a mapping between the baseline and distorted images. The distorted image is then resampled using the transform parameters to create the registered image. Bilinear interpolation is used for resampling.

A.3 Results

To demonstrate the performance of the algorithms we processed a set of videos taken by the EVS cameras during a flight test at Patrick Henry airport in Newport News, Virginia. The video sequence was taken as the NASA 757 aircraft approached a runway, and was digitized using a Canopus Video Board. Three images (one from each camera) time-aligned

at 00:26:14:18 were used for registration. As stated earlier, the SWIR image is used as the baseline for registration since it has the poorest spatial resolution and FOV. The SWIR, LWIR and visible images are shown in Figures A.1, A.2 and A.3 respectively. To provide a similarity metric to validate the performance of the registration algorithms we display the absolute difference of the reference and corrected images. This provides a visible validation of the registration process since features such as runway edges should align if registration is performed correctly.

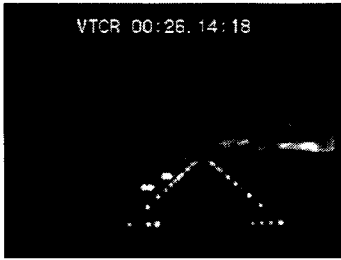


Figure A.1: Original SWIR



Figure A.2: Original LWIR

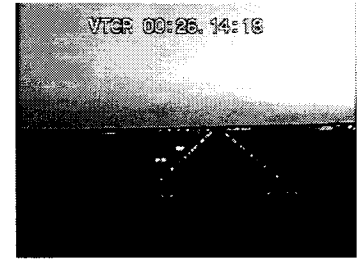


Figure A.3: Original Visible

A.3.1 SS algorithm

Applying the SS algorithm with the SWIR image as the baseline, and the LWIR and visible images as distorted images yields the “registered” SWIR, LWIR and visible images shown in Figures A.4, A.5 and A.6 respectively. The FOV of the LWIR image has been made smaller to match the FOV of the SWIR image. This change in FOV can clearly be seen in the horizontal direction of Figure A.5, by observing that the blurred artifact (which is an antenna in the FOV of the camera) in the upper left corner of the original LWIR is now almost completely removed in the registered image. In the vertical direction, the decrease in FOV is noted by the missing timecode at the top and the missing ground features at the bottom of Figure A.5 that are in the original image. The IFOVs have also been matched

though resampling. The general effects of resampling can be seen by observing the expansion of image features from Figure A.2 to Figure A.5. The FOV of the original visible image in Figure A.3 has been made slightly smaller, again to match the FOV of the SWIR, in Figure A.6. Since the FOVs nearly match and the image dimensions are the same, there is only a small expansion to match IFOVs, hence the registered image features are only slightly increased from the original.

Figure A.7 is the differenced SWIR and SS registered LWIR, and Figure A.8 is the differenced SS registered LWIR and SS registered visible image. The misalignment between the images after registration can clearly be seen in Figure A.7 by observing the difference in the outline of the runway from the LWIR component of the image, and the runway lights from the SWIR image. There is at least a large translation and a small rotation difference between the SWIR and registered LWIR. Similarly, the misalignment between the registered LWIR and visible images differenced in Figure A.8 can also be seen by noting the difference in the outline of the runway from the LWIR image, and the runway lights from the visible image. Again, there is an obvious translation between the images. Figures A.7 and A.8 clearly display the misalignment between the images thus indicating that differences in sensor design characteristics are not the only cause of distortion between the images.

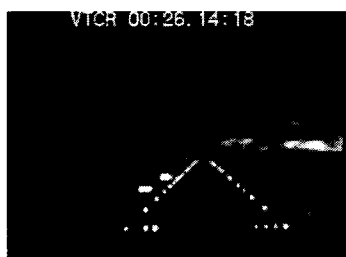


Figure A.4: Cropped SWIR

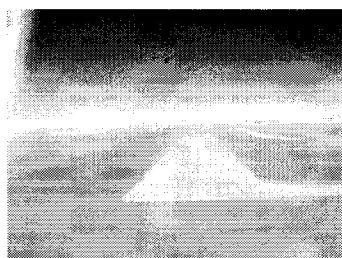


Figure A.5: SS Reg. LWIR

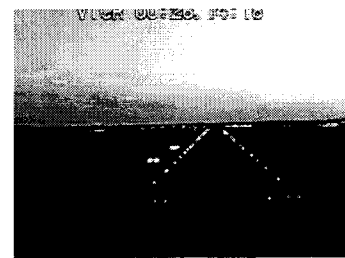


Figure A.6: SS Reg. visible



Figure A.7: SWIR and SS Registered LWIR

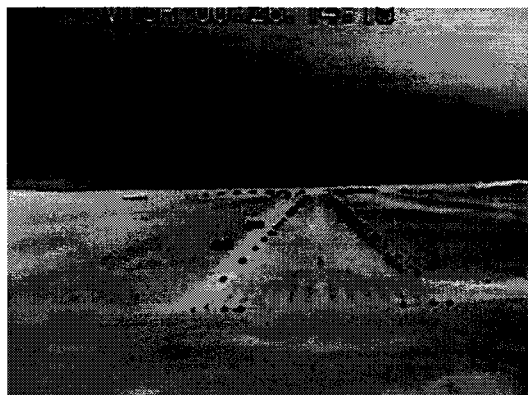


Figure A.8: SS Registered LWIR and visible

A.3.2 MLR algorithm

First we applied the MLR algorithm to the original (uncropped) SWIR and visible images, again using the SWIR as the baseline. Due to the lack of features around the perimeter of the SWIR image we used the runway lights as control points. Note that we are only using three control points for demonstration purposes. Figure A.9 repeats the original SWIR image for reference. Figure A.10 shows the registered visible image and Figure A.11 is the differenced SWIR and registered visible image. The coefficients obtained are given in Table A.3.

	b_0	b_1	b_2
x'	-0.546156	1.021212	-0.004578
y'	-20.440557	-0.007477	0.972837

Table A.3: Visible to SWIR MLR Coefficients

A close look at the runway and the runway lights in the two images shows that they are now registered. In particular, in the lower right corner of the SWIR image there are four runway lights lined up horizontally. In the visible image there are three runway lights in the same position, except the second light from the left is not visible. Figure A.11 shows

the four lights differenced in a horizontal line with the missing visible light filled in from the SWIR image. It is clear to see the warp performed during registration by observing the timecode size and location differences in the differenced images. The timecodes are the same size and at the same location in the original images. Figure A.9 and Figure A.10 could now be equally cropped to remove disjoint pixels around the perimeter to obtain the final images to be fused.

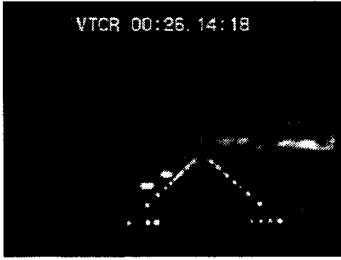


Figure A.9: Repeated SWIR

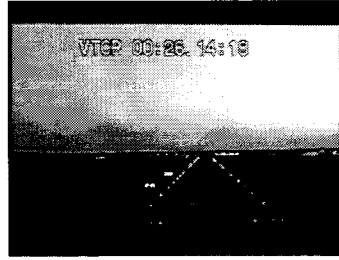


Figure A.10: MLR Reg. visible



Figure A.11: SWIR and MLR Reg. visible

Next we applied the MLR algorithm to the registered visible and LWIR images using the visible image as the baseline. Since the runway lights are not visible in the LWIR image, we use the intersecting lines at the bottom and top of the runway, and a stripe at the beginning of the runway towards the right in the LWIR image as control points. Figure A.12 repeats the MLR registered visible image for reference. Figure A.13 is the registered LWIR image and Figure A.14 is the differenced registered visible and the just registered LWIR images. The coefficients obtained are given in Table A.4.

	b_0	b_1	b_2
x'	8.347350	0.850628	0.037684
y'	9.637629	-0.012015	0.779082

Table A.4: LWIR to visible SWIR MLR Coefficients

As is evident in Figure A.14 the runway portion of the LWIR image is aligned with the

runway portion of the registered visible image. Also, the runway lights from the visible image border the perimeter of the LWIR runway. The one runway stripe selected as a control point is aligned. The taxiways on the right side of the image and the horizon across the image are also aligned. Again, any disjoint pixels around the perimeter could be removed by cropping. At this point all three original images are registered.



Figure A.12: Repeated MLR Registered visible



Figure A.13: MLR Reg. LWIR

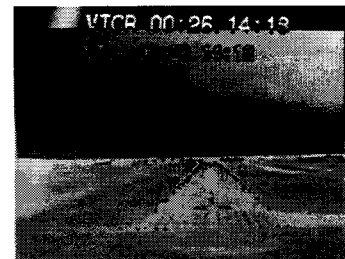


Figure A.14: MLR Reg. visible and LWIR

As a final test of the MLR algorithm we applied the same control point coefficients to a later frame in the video sequence. Figures A.15, A.16 and A.17 are the SWIR, LWIR and visible images at time 00:26:14:28, 10 seconds later in the sequence. Figure A.18 shows the MLR registered visible image. Figure A.19 is the differenced SWIR and MLR registered visible images. Figure A.20 is the MLR registered LWIR image. Figure A.21 is the differenced MLR registered visible and MLR registered LWIR image. As in the previous set of images, the registration can be observed by noting the alignment of the runway and runway lights in Figures A.19 and A.21.

A.3.3 Discussion

The images shown visually demonstrate the performance of the two algorithms on typical image data from the EVS. The registration inaccuracies of the SS algorithm are obvious.

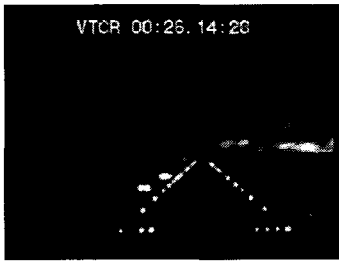


Figure A.15: Orig. SWIR at Time 26:14:28

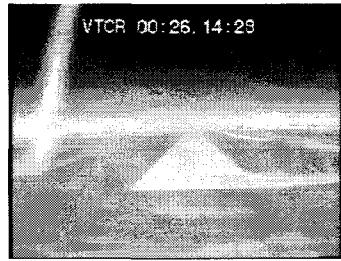


Figure A.16: Orig. LWIR at Time 26:14:28

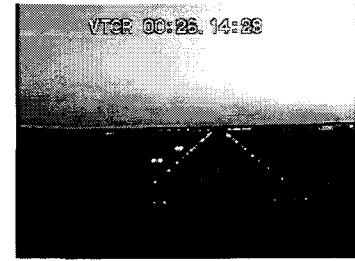


Figure A.17: Orig. visible at Time 26:14:28



Figure A.18: MLR Registered visible at Time 26:14:18



Figure A.19: SWIR and MLR Registered visible

The differing FOV and resolution specifications given do not take into account the other distortions within the images. With this much discrepancy there seems to be either a fundamental problem in the bore-sighting or alignment of the cameras, or the alignment is changing during flight. If the sensors were actually bore-sighted and aligned, the SS algorithm should be able to match the performance of the MLR algorithm and in addition, not require any manual intervention like selection of control points. True FOV values could be obtained from a thorough ground calibration, and non-interpolated pixels of the actual image dimensions could be obtained from raw digital data streams from the cameras.

The MLR algorithm provides better registration of the images than the SS algorithm configured with the current set of specifications and with the current EVS alignment. In



Figure A.20: MLR Registered LWIR at Time 26:14:18



Figure A.21: MLR Registered visible and LWIR

our examples the runway and runway lights are clearly aligned. The coefficients obtained with only three points indicates that there are rotation, translation, scale and possibly shear distortion components found between the images. These distortions can be seen by viewing the timecode warps at the top of the differenced images. The application of the same MLR control points to a set of time-aligned images later in the same video sequence produced the same level of registration. This indicates that we could successfully use the registration coefficients obtained from one set of time-aligned images to apply to, at least, a group of frames from the video sequence. If the alignment is not changing substantially during flight then all frames could be processed with the same transform.

A.4 Summary

Image registration is an essential prerequisite to subsequent image fusion. We have produced two algorithms to perform multi-image registration for the EVS. The SS algorithm uses EVS camera specifications and performs registration based solely on these parameters. The performance of this algorithm indicates that there is a severe inaccuracy in the boresighting

or alignment of the cameras. Correction of these issues should improve the performance of the algorithm and allow it to be used to automatically register all images across the cameras, or as validation of the MLR algorithm.

The MLR algorithm uses control point selection and linear regression to compute the coefficients of an affine spatial transformation. This transformation is then used to register the LWIR and visible images to the SWIR image. In addition, the MLR registration algorithm provides a means to generate a base set of coefficients for post processing of the full video stream across all cameras. We have subsequently used a set of baseline coefficients to process an entire 20 second video clip from each of the three cameras.

In addition, the coefficients obtained could also be used to back out the actual distortion values (translation amount, rotation angle, etc.) for feedback to the EVS designers. Improvements could also be made in the computation of the coefficients by using point selection with feedback or other more robust feature selection mechanisms. Manual control point selection can be improved by MSR enhancement of the images to emphasize and sharpen features prior to registration. This was done for another EVS data set and greatly improved the ability to select corresponding points. Most importantly, the actual boresighting and alignment can be checked against the values obtained from MLR and SS registration, and adjusted appropriately. This procedure could be performed both before, and after, EVS flight opportunities and used to verify and validate system alignment.

Appendix B

Field Programmable Gate Arrays

In the future we can capitalize on the lessons learned from mapping the real-time Retinex algorithm into a DSP architecture, and possibly use an alternative technology that will allow full customization of our design. One of these technologies is field programmable gate arrays (FPGAs). Architecture optimization usually implies performing the process of improving a system by properly allocating resources, such as memory or DMA channels, to improve execution speed or bandwidth. FPGAs redefine this term to apply at a much lower level of abstraction. Specifically, FPGAs are composed of a large matrix of logic cells, routing resources, and I/O blocks that must be selected, configured and interconnected. Figure B.1 is a block diagram of a typical FPGA architecture. A logic cell can be as simple as a transistor pair or 2-input nand gate, or as complex as a full microprocessor core. Logic cells are typically based on multiplexers and basic logic gates, or SRAM-based look-up tables (LUTs), and are generally used to implement combinatorial or sequential logic functions.

The routing resources implement the “field-programmable” portion of the FPGA definition. They are the interconnect fabric (wires) and electrical switches that are programmed

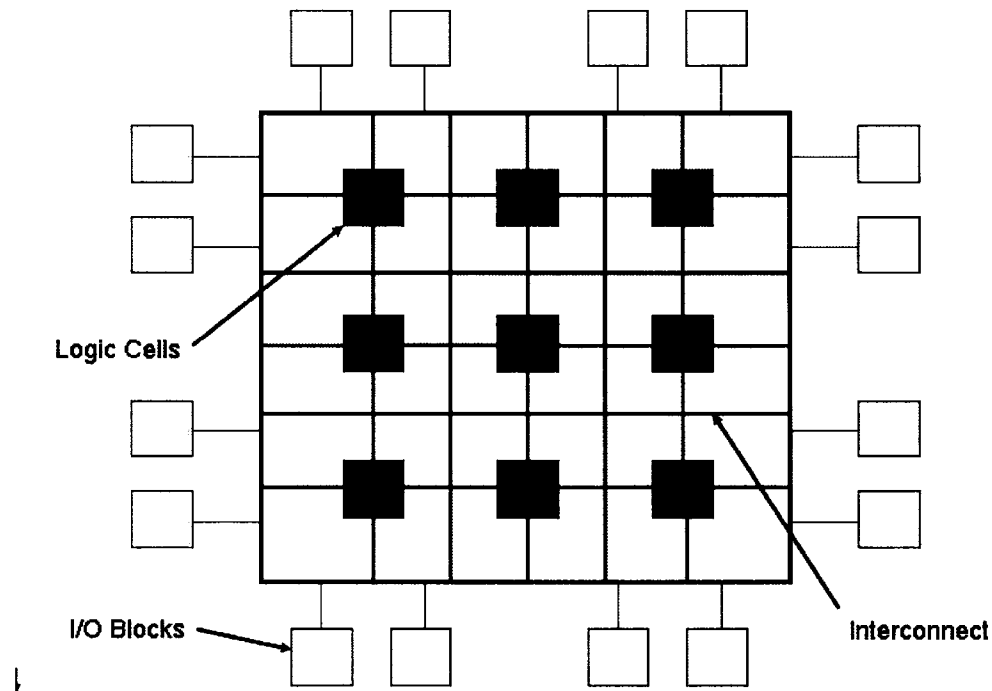


Figure B.1: High-level block diagram of a typical FPGA Architecture

and can (usually) be reprogrammed in-situ, i.e. after its manufactured or even during active operation. This concept leads to the idea of chip-level reconfigurable computing. Three primary programming technologies are used to implement the switches [58]: pass transistors controlled by the status of an SRAM bit, electrical programmable read only memory (EPROM) floating-gate transistors, or small antifuse switches electrically formed once by creating a low resistance path to ground. FPGAs that use write-once antifuses are tedious to use because the design must be complete and verified before programming, but they provide the benefits of low resistance and parasitic capacitance, high reliability and density, and can be relatively easily fabricated in a radiation-hardened foundry. Xilinx is large manufacturer of SRAM-based FPGAs, the Altera Max products are CPLDs, and

Actel is a major vendor of antifuse based FPGAs. The I/O blocks are special-purpose logic cells generally spread around the peripheral of the device that are used to buffer input and output signals. They can usually be configured to transfer input, output, or bi-directional signals.

The logical functionality of an FPGA is programmed into the device through a number of design stages [51]. First, a high-level design is entered as a structural design, normally through a schematic, or as a behavioral design using hardware description languages, such as VHDL or Verilog. Computer-aided electronic design automation tools exist for both, and often offer alternative entry methods, such as state-machine or waveform editors. Logic synthesis is performed next, where the high-level design is compiled into a netlist and translated into the available cells and technologies provided on the FPGA. Several issues are addressed during this stage, such as design size checks and redundancy elimination. After this, place and route is performed where cell placement is determined and the routing interconnect is defined. Finally, a configuration bit file is generated and downloaded into the device for programming. Because of the complexity of most FPGA architectures, functional and timing simulations are often performed concurrently and iteratively with the design stages. This allows the designer to correct errors before programming the device and is critical to ensure the successful implementation of antifuse-based devices. Test vectors that provide stimulus to both the simulator and actual device are also often generated to aid in debugging and to verify and validate behavior.

FPGA capacities in the late 1980's were on the order of thousands of usable gates [51]. They were often used as "glue logic", absorbing the functionality of a variety of miscellaneous logic, and performed functions like providing interfaces to external memories or

peripherals. Over the last few years the density and capabilities of FPGAs have increased tremendously. As an example, the XC2VP100 is the largest FPGA in the Xilinx Virtex-II Pro family of devices first introduced in January 2002. It contains 99,216 logic cells, 7,992-kbits of BRAM, 444 dedicated 18×18 multipliers, 12 digital clock managers (DCMs), 1164 user I/O pins, 2 PowerPC RISC processors, and 20 3.125 Gbps Rocket I/O serial transceivers. Each logic cell contains a 4-input LUT, a flip-flop and carry logic. BRAM is block RAM comprised of distributed and global dual-port SRAM. As FPGA densities have increased, so have the number of potential uses. They are now often used as co-processors, hardware accelerators, or custom, reconfigurable computing architectures. Several authors have suggested and implemented individual image processing functions [45, 46, 11, 12] as well as full platform and system solutions [1, 17, 38, 14, 68]. Xilinx and other vendors offer several DSP cores, such as 2-D 1024-point FFTs and YCrCb-to-RGB converters, that perform complex processing functions and can be easily inserted into a design.

We could design and map a new version of the algorithm into this technology taking advantage of its' capabilities. We may be able to properly utilize a single high-density FPGA to parallel process three spectral bands of image data. Otherwise, we could use a multi-FPGA platform that would allow pipelining the major components of MSCR processing. Bandwidth issues could be reduced since we could create and optimize internal bus widths to our data transfer requirements. High-level code could be written using VHDL and synthesized using Synplicity FPGA development tools. We also have access to other high-level tools, such as Matlab, that could be used for design, simulation, and test. An FPGA Retinex processing core could eventually be developed for widespread use in other Xilinx platforms.

Appendix C

DM642 EVM Flash Programming Guidelines

Many embedded applications require the need to execute automatically at system power-up after reset without outside intervention. This is often accomplished by storing application code in a non-volatile memory such as a read only memory (ROM) or flash memory. At power-up (or boot) the stored code is automatically copied into a runtime memory location in random access memory (RAM) and then the beginning program address is branched to to begin execution. We require this automatic start-up capability for our DM642 EVM based implementation used in the EVS system. The EVS system is required to work as an embedded, autonomous system. Power-up and power-down cycles are performed frequently during pre-flight check-outs and when the plane has stopped at other airport facilities. No operator is continually available to monitor the system and repeatedly reload code from a host, therefore loading and executing code autonomously is required.

The DM642, like all the other TI DSPs, has a set of facilities to support bootloading.

Three boot configuration modes are supported — no boot, ROM boot, and host boot. In no boot mode no action is performed at boot, and in host boot, an external host controls the boot process. In ROM boot mode after reset is released, the CPU is stalled until 1-KByte of memory is copied from the beginning of an external ROM to RAM address 0 using the EDMA controller. After this transfer is complete the CPU is released and starts to execute code at address 0.

Many applications will not fit in 1-KByte of memory. In this case, the code that is copied is usually a second-level bootloader that in turn, copies the rest of the application into RAM. The DM642 EVM has 4-MBytes of 8-bit wide flash. The flash is mapped into the 0x90000000 to 0x9007FFFF (lower CE1 space) address range of the DM642 using 19 address bits (A0-A18). This is smaller than the memory space available in the flash so an FPGA on the EVM is used to create 3 additional address lines extending the address range to 4-MBytes. These 3 lines effectively act as page bits dividing the address space into 0.5-MByte pages. Unfortunately, they default to 000 at power-on reset because the SRAM-based FPGA becomes unconfigured at reset and tri-states the output of all I/O. We discuss the ramifications of this next.

The size of most of the first executable files generated for our implementations were about 500-KBytes, and would fit on the first page of flash. After adding ethernet service components (and the large libraries required by them) to allow a user to perform parameter updates from any available laptop, our executable code size grew to \approx 677-KBytes. To place these executables in flash memory requires using a Flashburn utility provided by TI. This utility requires the file to be burned to be in one of several specific formats. We choose the hex format and used the TI supplied hex6x utility to perform the conversion.

TI supplies a sample second-level boot loader (`boot.asm`) assembly file that is used to load a users application. When the `boot.asm` file is included with the application, the `hex6x` conversion routine properly allocates the boot code (at address `0x00`) and the application code (defaulting to address `0x400`, immediately after the 1-KByte boot code). When burned into flash, the flashburn utility physically places the code in memory according to the values in the hex file and any offsets selected at burn time by the user.

The EVM board manufacturer, Spectrum Digital, supplies a default flash program that contains the configuration bits (of size `0x393D0` or ≈ 234 -KBytes) for the FPGA, and a program (`fpga_loader`) that loads the FPGA with these configuration bits. So the default setup would have `boot.asm` at address 0, the `fpga_loader` code at address `0x400` and the configuration bits at address `0x40000`. The boot code would be loaded into RAM at reset. After reset, it would copy the `fpga_loader` code into RAM and branch to the entry point of the `fpga_loader`, which subsequently loads the configuration bits into the FPGA.

The main issue is that the FPGA controls the addressing used for the flash (the page bits) and if it is not properly configured, the upper pages of flash cannot be accessed. So attempting to use flash above page 0 (above 512-KBytes) becomes a non-trivial issue. Our first attempts failed because our standard routine had been to erase flash, burn the FPGA configuration bits and our application (with `boot.asm` embedded in it), and then restart the system. This worked because our application fit on one page of flash. Now that our code was greater than one page, the burn failed because the address lines are all at zero and addresses above 512-KBytes are mapped into lower memory. In addition after getting data burned on more than one page, we needed a method to copy the upper pages of information into RAM and restart execution.

Our solution was to first, burn the `fpga_loader` with the default boot code and the FPGA configuration bits into the first page of flash and reboot thus configuring the FPGA. This gives us full access to all of flash memory. We then modified the default `fpga_loader` program so that at the end of the program execution it now (1) changes the FPGA page bits from 0 to 1 and (2) branches to a third-level bootloader at address `0x90000000` to load our application. To change the register that controls the page bits to 1 we use a function supplied in the `dm642` board support library: `evmdm642_rset(evmdm642_flashpage,1)`. To branch, we use three simple assembly language instructions in C code: `asm(" MVKL 0x90000000,A15");` `asm(" MVKL 0x90000000,A15");` and `asm(" BNOP,0x5");`.

Next we burned into memory the FPGA configuration bits at `0x90040000`, the modified `fpga_loader` with the default second-level boot code, and our application code embedded with the third-level boot code at address `0x90080000`. The address change is performed using an offset of `(0x80000)` in the Flashburn utility. When the modified `fpga_loader` branches to the address `0x90000000` with the page bits set to page 1, we are actually addressing address `0x80000` of the flash. The third-level bootloader simply loads into RAM our application from flash address `0xC0000`, branches to the start of the application code and begins execution.

Finally, here are a few miscellaneous notes on the discussion above. When burning the configuration bits, the default hex file already has the `0x40000` offset built in so nothing has to be done to place the data there in the Flashburn utility. Similarly the default and modified `fpga_loader` hex files instruct Flashburn to place the boot code at `0x00` and the application code at `0x400`. The application hex file is also built under the assumption that the boot code is placed at `0x00` and the application code is placed at `0x400`. We force the Flashburn tool to provide the offsets of `0x80000` and `0x80400` respectively.

Our current method executes the third-level bootloader out of slow flash memory. Although this only requires a few seconds, we could speed up the loading process by copying third-level bootloader (the first 1-KByte of memory at 0x80000) into RAM in the same way that the first bootloader does at power-up. Executing the third-bootloader out of RAM would then provide faster loading time. The information we developed for this guide will be used in a new TI application report on bootloaders for their C6X processors.

Bibliography

- [1] A. LYNN ABBOTT, PETER M. ATHANAS, AND ADIT TARMASER. Accelerating image filters using a custom-computing machine. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, John Schewel, editor, volume 2607, pages 62–70. Proceedings of SPIE, October 1995.
- [2] IRVING ASHKENAS, HENRY R. JEX, AND DUANE T. MCRUER. Pilot-induced oscillations: their cause and analysis. Technical Report Northrop Corp. 64-143, NASA Langley Research Center, June 1964.
- [3] ATEME SA. IEKC64x users manual. Technical report, ATEME, Bievres, France, 2003.
- [4] ANDREW BATEMAN AND IAIN PATERSON-STEPHENS. *The DSP Handbook: Algorithms, Applications, and Design Techniques*. Prentice Hall, 2002.
- [5] R. BRACEWELL, O. BUNEMAN, H. HAO, AND J. VILLASENOR. Fast two-dimensional Hartley transform. In *Proceedings of the IEEE*, volume 74, No. 9, pages 1282–1283, September 1986.
- [6] THOMAS BRAUNL. *Parallel Image Processing*. Springer, 2000.
- [7] E. ORAN BRIGHAM. *The Fast Fourier Transform*. Prentice-Hall, 1975.
- [8] RICHARD R. BROOKS AND S. S. IYENGAR. *Multi-Sensor Fusion: Fundamentals and Applications*. Prentice Hall, 1998.
- [9] LISA GOTTESFELD BROWN. A survey of image registration techniques. In *ACM Computing Surveys*, volume 24, No.4, December 1992.
- [10] HOWARD BURDICK. *Digital Imaging*. McGraw Hill, 1997.
- [11] CHRIS DICK. Computing multi-dimensional DFTs using Xilinx FPGAs. In *The 8th International Conference on Signal Processing Applications and Technology*, September 13-16 1998.
- [12] CHRIS DICK. Minimum multiplicative complexity implementation of the 2-D DCT using Xilinx FPGAs. In *Configurable Computing: Technology and Applications*. Proceedings of SPIE's Photonics East, November 1–6 1998.

- [13] NEIL ANTHONY DODGSON. Image resampling. Technical Report 261, University of Cambridge, United Kingdom, August 1992.
- [14] BRUCE A. DRAPER, J. ROSS BEVERIDGE, A. P. WILLEM BOEHM, CHARLES ROSS, AND MONICA CHAWATHE. Accelerated image processing on FPGAs. *IEEE Transactions on Image Processing*, 12(12), December 2003.
- [15] PIERRE DUHAMEL. A connection between bit reversal and matrix transposition: Hardware and software consequences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(11):1893–1896, November 1990.
- [16] J. EKLUNDG. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21:801–803, 1972.
- [17] LEE FERGUSON. Image processing using reconfigurable FPGAs. In *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, John Schewel, Peter M. Athanas, V. Michael Bove, Jr., and John Watson, editors, volume 2914, pages 110–121. Proceedings of SPIE, November 1996.
- [18] LELIA M. G. FONSECA AND B. S. MANJUNATH. Registration techniques for multisensor remotely sensed imagery. In *Photogrammetric Engineering and Remote Sensing*, volume 62, No. 9, pages 1046–1056, September 1996.
- [19] JACK G. GANSSLE. *The Art of Programming Embedded Systems*. Academic Press, 1992.
- [20] RAFAEL C. GONZALEZ AND RICHARD E. WOODS. *Digital Image Processing*. Addison-Wesley, 1993.
- [21] ARDESHIR GOSHTASBY. Image registration by local approximation methods. In *Image Vision Computing*, volume 6, pages 255–261, November 1988.
- [22] DAVID B. HARRIS, JAMES H. MCCLELLAN, DAVID S. K. CHAN, AND HANS W. SCHUESSLER. Vector radix fast fourier transform. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 548–551, 1977.
- [23] GLENN D. HINES, ZIA-UR RAHMAN, DANIEL J. JOBSON, AND GLENN A. WOODELL. Multi-sensor image registration for an enhanced vision system. In *Visual Information Processing XII. Proceedings of SPIE 5108*, Zia-ur Rahman, Robert A. Schowengerdt, and Stephen E. Reichenbach, editors, April 2003.
- [24] GLENN D. HINES, ZIA-UR RAHMAN, DANIEL J. JOBSON, AND GLENN A. WOOD-ELL. DSP implementation of the retinex image enhancement algorithm. In *Visual Information Processing XIII. Proceedings of SPIE 5438*, Zia-ur Rahman, Robert A. Schowengerdt, and Stephen E. Reichenbach, editors, April 2004.
- [25] GLENN D. HINES, ZIA-UR RAHMAN, DANIEL J. JOBSON, AND GLENN A. WOOD-ELL. Single-scale retinex using digital signal processors. In *Global Signal Processing Conference*, September 2004.

- [26] GLENN D. HINES, ZIA-UR RAHMAN, DANIEL J. JOBSON, GLENN A. WOODSELL, AND STEVEN D. HARRAH. Real-time enhanced vision system. In *Enhanced and Synthetic Vision, Proceedings of SPIE 5802*, Jacques G. Verly, editor, March 2005.
- [27] FRIEDRICH O. HUCK, CARL L. FALES, AND ZIA-UR RAHMAN. *Visual Communication: An Information Theory Approach*. Kluwer Academic, 1997.
- [28] IEEE. IEEE Std 1149.1 standard test access port and boundary-scan architecture. Technical Report SSYA002C, IEEE, New York, New York, 1993.
- [29] KEITH JACK. *Video Demystified*. Brooktree, 1993.
- [30] ANIL K. JAIN. *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.
- [31] JOHN R. JENSEN. *Introductory Digital Image Processing*. Prentice Hall, 1996.
- [32] DANIEL J. JOBSON, ZIA-UR RAHMAN, AND GLENN A. WOODSELL. A multi-scale Retinex for bridging the gap between color images and the human observation of scenes. *IEEE Transactions on Image Processing: Special Issue on Color Processing*, 6(7):965–976, July 1997.
- [33] DANIEL J. JOBSON, ZIA-UR RAHMAN, AND GLENN A. WOODSELL. Properties and performance of a center/surround retinex. *IEEE Trans. on Image Processing*, 6(3):451–462, March 1997.
- [34] GARY V. KELLOG AND CHARLES A WAGNER. Effects of update and refresh rates on flight simulation visual displays. Technical Report 100415, NASA Langley Research Center, February 1988.
- [35] JOSEF KITTLER AND MICHAEL J. B. DUFF. *Image Processing System Architectures*. Research Studies Press, 1985.
- [36] SUN YUAN KUNG. *VLSI Array Processors*. Prentice-Hall, 1988.
- [37] EDWARD LAND. An alternative technique for the computation of the designator in the retinex theory of color vision. In *Proceedings of the National Academy of Science*, volume 83, pages 3078–3080, 1986.
- [38] PHILLIP LAPLANTE AND WILLIAM GILREATH. Single instruction set architectures for image processing. In *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communication IV*, John Schewel, Philip B. James-Roxby, Herman Schmit, and John T. McHenry, editors, volume 4867, pages 20–29. Proceedings of SPIE, July 2002.
- [39] PHILLIP A. LAPLANTE AND ALEXANDER D. STOYENKO. *Real-Time Imaging: Theory, Techniques, and Applications*. IEEE Press, 1996.
- [40] JON C. LEACHTENAUER. *Electronic Image Display*. SPIE Press, 2004.

- [41] HUI LI, B. S. MANJUNATH, AND SANJIT K. MITRA. A contour-based approach to multisensor image registration. *IEEE Transactions on Image Processing*, 4, No. 3, March 1995.
- [42] HUI HENRY LI AND YI-TONG ZHOU. Automatic visual/ir image registration. In *Optical Engineering*, volume 35(2), pages 391–400, February 1996.
- [43] REN C. LUO AND MICHAEL G. KAY. *Multisensor Integration and Fusion for Intelligent Machines and Systems*. Ablex, Norwood, NJ, 1995.
- [44] DUANE T. MCRUER. Pilot-induced oscillations and human dynamic behavior. Technical Report NASA-CR-4683, NASA Langley Research Center, July 1995.
- [45] LES MINTZER. The FPGA as FFT processor. In *6th International Conference on Signal Processing Applications and Technology*, pages 1378–1382, October 1995.
- [46] LES MINTZER. Large FFT's in a single FPGA. In *7th International Conference on Signal Processing Applications and Technology*, volume 1, pages 895–899, October 7–10 1996.
- [47] KURT NOVAK. Rectification of digital imagery. In *Photogrammetric Engineering and Remote Sensing*, volume 58, No. 9, pages 399–344, March 1992.
- [48] HENRI J. NUSSBAUMER. *Fast Fourier Transform and Convolution Algorithms*. Springer Verlag, 1981.
- [49] ALAN V. OPPENHEIM AND RONALD F. SHAFER. *Digital Signal Processing*. Prentice-Hall, 1975.
- [50] STEPHEN K. PARK AND ROBERT A. SCHOWENGERDT. Image reconstruction by parametric cubic convolution. In *Computer Vision, Graphics, and Image Processing*, volume 23, pages 258–272, 1983.
- [51] DAVID PELLERIN AND MICHAEL HOLLEY. *Practical Design Using Programmable Logic*. Prentice Hall, 1991.
- [52] IOANNIS PITAS AND MICHAEL G. STRINTZIS. Algorithms for the reduction of the I-O operations in the calculation of the 2-D DFT. In *Signal Processing*, volume 12, pages 277–289, 1987.
- [53] CHARLES A. POYNTON. *Digital Video and HDTV Algorithms and Interfaces*. John Wiley & Sons, 2003.
- [54] WILLIAM K. PRATT. *Digital Image Processing*. John Wiley and Sons, 1991.
- [55] RAHMAN. see <http://dragon.larc.nasa.gov> for examples.
- [56] ZIA-UR RAHMAN, DANIEL J. JOBSON, AND GLENN A. WOODSELL. Retinex processing for automatic image enhancement. *Journal of Electronic Imaging*, 13, No. 1:100–110, January 2004.

- [57] ZIA-UR RAHMAN, DANIEL J. JOBSON, GLENN A. WOODELL, AND GLENN D. HINES. Multi-sensor fusion and enhancement using the retinex image enhancement algorithm. In *Visual Information Processing XI, Proceedings of SPIE 4736*, Zia-ur Rahman, Robert A. Schowengerdt, and Stephen E. Reichenbach, editors, April 2002.
- [58] ZORAN SALCIC AND ASIM SMAILAGIC. *Digital Systems Design and Prototyping Using Field Programmable Logic and Hardware Description Languages*. Kluwer Academic, 2000.
- [59] ROBERT SCHOWENGERDT. *Remote Sensing: Models and Methods for Image Processing*. Academic Press, 1997.
- [60] RAVI K. SHARMA AND MISHA PAVEL. Multisensor image registration. In *SID Digest, Society for Information Display*, volume XXVIII, pages 951–954, May 1997.
- [61] JULIUS O. SMITH. *Mathematics of the Discrete Fourier Transform (DFT)*. W3K, 2003.
- [62] MICHAEL JOHN SEBASTIAN SMITH. *Application-Specific Integrated Circuits*. Addison-Wesley, 1997.
- [63] STEVEN W. SMITH. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical, 1997.
- [64] WINTHROP W. SMITH AND JOANNE M. SMITH. *Handbook of Real-Time Fast Fourier Transforms*. IEEE Press, 1995.
- [65] GEORGE W. SNEDECOR. *Statistical Methods*. Iowa State University, 8 edition, 1989.
- [66] SPECTRUM DIGITAL. TMS320C6713 DSK technical reference. Technical Report 506735-0001, Spectrum Digital, Stafford, Texas, May 2003.
- [67] SPECTRUM DIGITAL. TMS320DM642 evaluation module technical reference. Technical Report 506845-0001, Spectrum Digital, Stafford, Texas, August 2003.
- [68] OLAF STORAASLI. Computing faster without CPUs: Scientific applications on a reconfigurable FPGA-based hypercomputer. In *6th Military and Aerospace Programmable Logic Devices Conference*, September 2003.
- [69] SABINE SUSSTRUNK, ROBERT BUCKLEY, AND STEVE SWEN. Standard RGB color spaces. In *The Seventh Color Imaging Conference: Color Science, Systems and Applications*. IS&T - The Society for Imaging Science and Technology, 1999.
- [70] TEXAS INSTRUMENTS. TVP3026 data manual video interface palette. Technical Report SLAS098B, Texas Instruments, Dallas, Texas, July 1996.
- [71] TEXAS INSTRUMENTS. TMS320C6000 technical brief. Technical Report SPRU197D, Texas Instruments, Dallas, Texas, February 1999.

- [72] TEXAS INSTRUMENTS. TMS320C6000 imaging developer's kit (IDK) video device driver user's guide. Technical Report SPRU499, Texas Instruments, Dallas, Texas, December 2000.
- [73] TEXAS INSTRUMENTS. TMS320C6711 floating-point digital signal processor data manual. Technical Report SPRS073D, Texas Instruments, Dallas, Texas, September 2000.
- [74] TEXAS INSTRUMENTS. TVP5022 data manual NTSC/PAL video decoder. Technical Report SLAS274, Texas Instruments, Dallas, Texas, July 2000.
- [75] TEXAS INSTRUMENTS. Implementing fast fourier transform algorithms of real-valued sequences with the TMS320 DSP platform. Technical Report SPRA291, Texas Instruments, Dallas, Texas, August 2001.
- [76] TEXAS INSTRUMENTS. TMS320C6000 imaging developer's kit (IDK) programmer's guide. Technical Report SPRU495A, Texas Instruments, Dallas, Texas, September 2001.
- [77] TEXAS INSTRUMENTS. TMS320C6000 imaging developer's kit (IDK) user's guide. Technical Report SPRU494a, Texas Instruments, Dallas, Texas, September 2001.
- [78] TEXAS INSTRUMENTS. TMS320C6000 peripherals reference guide. Technical Report SPRU190D, Texas Instruments, Dallas, Texas, February 2001.
- [79] TEXAS INSTRUMENTS. TMS320C64x technical overview. Technical Report SPRU395B, Texas Instruments, Dallas, Texas, January 2001.
- [80] TEXAS INSTRUMENTS. TMS320 DSP/BIOS user's guide. Technical Report SPRU423B, Texas Instruments, Dallas, Texas, November 2002.
- [81] TEXAS INSTRUMENTS. TMS320C6000 chip support library API user's guide. Technical Report SPRU401E, Texas Instruments, Dallas, Texas, December 2002.
- [82] TEXAS INSTRUMENTS. TMS320C6000 programmer's guide. Technical Report SPRU198G, Texas Instruments, Dallas, Texas, August 2002.
- [83] TEXAS INSTRUMENTS. TMS320C64x DSP library programmer's reference. Technical Report SPRU565A, Texas Instruments, Dallas, Texas, April 2002.
- [84] TEXAS INSTRUMENTS. TMS320C6713 floating-point digital signal processor data manual. Technical Report SPRS186B, Texas Instruments, Dallas, Texas, November 2002.
- [85] TEXAS INSTRUMENTS. The TMS320C67x FastRTS library programmer's reference. Technical Report SPRU100A, Texas Instruments, Dallas, Texas, October 2002.
- [86] TEXAS INSTRUMENTS. TMS320DM642 Video/Imaging fixed-point digital signal processor data manual. Technical Report SPRS200B, Texas Instruments, Dallas, Texas, July 2002.

- [87] TEXAS INSTRUMENTS. TMS320C6000 DSP cache user's guide. Technical Report SPRU656A, Texas Instruments, Dallas, Texas, May 2003.
- [88] TEXAS INSTRUMENTS. TMS320C621x/C671x DSP two-level internal memory reference guide. Technical Report SPRU609A, Texas Instruments, Dallas, Texas, November 2003.
- [89] TEXAS INSTRUMENTS. TMS320C64x DSP video Port/VCXO interpolated control (VIC) port reference guide. Technical Report SPRU629, Texas Instruments, Dallas, Texas, April 2003.
- [90] TEXAS INSTRUMENTS. TMS320C67x DSP library programmer's reference guide. Technical Report SPRU657, Texas Instruments, Dallas, Texas, February 2003.
- [91] TEXAS INSTRUMENTS. TVP5150A data manual. Technical Report SLES087, Texas Instruments, Dallas, Texas, September 2003.
- [92] TEXAS INSTRUMENTS. Migrating from TMS320C6211B/C6711/C6711B and C6713 to TMS320C6713B. Technical Report SPRA851G, Texas Instruments, Dallas, Texas, March 2004.
- [93] TEXAS INSTRUMENTS. TMS320C6000 optimizing compiler user's guide. Technical Report SPRU187L, Texas Instruments, Dallas, Texas, May 2004.
- [94] TEXAS INSTRUMENTS. The TMS320C6000 PLL controller reference guide. Technical Report SPRU233, Texas Instruments, Dallas, Texas, March 2004.
- [95] TEXAS INSTRUMENTS. TMS320C64x DSP two-level internal memory reference guide. Technical Report SPRU610A, Texas Instruments, Dallas, Texas, June 2004.
- [96] TEXAS INSTRUMENTS. TVP5146 data manual. Technical Report SLES084A, Texas Instruments, Dallas, Texas, November 2004.
- [97] TEXAS INSTRUMENTS. TMS320C6416T fixed-point digital signal processor data manual. Technical Report SPRS226H, Texas Instruments, Dallas, Texas, August 2005.
- [98] CARLO L. M. TIANA, J. RICHARD KERR, AND STEVEN D. HARRAH. Multispectral uncooled infrared enhanced vision system for flight test. In *Proceedings of SPIE*, volume 4363, April 2000.
- [99] TRUVIEW. see <http://www.truview.com>.
- [100] JOHN WATKINSON. *The Art of Digital Video*. Focal Press, 1990.
- [101] DICK R. WITTINK. *The Application of Regression Analysis*. Allyn and Bacon, 1988.
- [102] GEORGE WOLBERG. *Digital Image Warping*. IEEE Computer Society Press, 1990.

- [103] GLENN A. WOODDELL, DANIEL J. JOBSON, ZIA-UR RAHMAN, AND GLENN D. HINES. Enhanced images for checked and carry-on baggage and cargo screening. In *Sensors, and Command, Control, Communications and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense III, Proceedings of SPIE 5403*, April 2004.
- [104] ZHAO ZHANG AND XIAODONG ZHANG. Fast bit-reversals on uniprocessors and shared-memory multiprocessors. *SIAM Journal on Scientific Computing*, 22(6):2113–2134, 2001.



VITA

Glenn Derrick Hines

Glenn Derrick Hines was born in Portsmouth, Virginia on August 25, 1964. He graduated from I. C. Norcom High School, Portsmouth, Virginia, in 1982. He received his Bachelor of Science degree in Electrical Engineering from Old Dominion University in 1987, his Master of Science degree in Electrical Engineering from Old Dominion University in 1991, and his Master of Science degree in Computer Science from The College of William and Mary in 2002. Glenn defended his dissertation in January 2006 and will graduate with a Doctor of Philosophy degree in Computer Science from The College of William and Mary in May 2006. Glenn is employed as a senior electronics engineer and computer scientist at NASA Langley Research Center in Hampton, Virginia. He performs research in the area of image processing and is responsible for the development of aviation, spaceflight, and atmospheric research instruments. He is married to the former Sunita Etwaroo and has three children.