DSP Implementation of the Retinex Image Enhancement Algorithm

Glenn Hines[a], Zia-ur Rahman[b], Daniel Jobson[a],Glenn Woodell[a]

[a]NASA Langley Research Center, Hampton, VA 23681;
[b]College of William & Mary, Department of Applied Science, Williamsburg, VA 23187

ABSTRACT

The Retinex is a general-purpose image enhancement algorithm that is used to produce good visual representations of scenes. It performs a non-linear spatial/spectral transform that synthesizes strong local contrast enhancement and color constancy. A real-time, video frame rate implementation of the Retinex is required to meet the needs of various potential users. Retinex processing contains a relatively large number of complex computations, thus to achieve real-time performance using current technologies requires specialized hardware and software. In this paper we discuss the design and development of a digital signal processor (DSP) implementation of the Retinex. The target processor is a Texas Instruments TMS320C6711 floating point DSP. NTSC video is captured using a dedicated frame-grabber card, Retinex processed, and displayed on a standard monitor. We discuss the optimizations used to achieve real-time performance of the Retinex and also describe our future plans on using alternative architectures.

Keywords: image enhancement, digital signal processing, retinex

1. INTRODUCTION

Many digital image processing (IP) operations are inherently computationally intensive, where large data volumes must be stored, processed and transferred between processor and memory. Many IP algorithms require orthogonal data access or must process multiple image lines simultaneously, exacerbating the data processing problem. When video is considered, the processing requirements become enormous. Most general-purpose computers do not have the proper architecture or operating system to efficiently process image data (although the gap is rapidly closing). Several specialized hardware architectures and technologies have been developed that are a better match for IP requirements. Application specific integrated circuits (ASICs) are one-of-a-kind custom devices that often provide the performance required for IP but at the expense of long development times and high cost. Field programmable gate arrays (FPGAs) are an attractive alternative that offer relative ease of programming, high performance and completely reconfigurability to support custom applications. Digital signal processors (DSPs) are inexpensive, easy to program — usually in common high level languages such as C — and offer good performance. Several other esoteric technologies, such as array processors, are also available, but for quick, low cost, prototyping, DSPs are a suitable and sufficient design choice.

The IP algorithm that is implemented and evaluated in this research effort is the Retinex. The Retinex is an image enhancement algorithm that is used for producing good visual representations of scenes. It performs a non-linear spatial/spectral transform that synthesizes strong local contrast enhancement and color constancy.[1, 2] The algorithm is based on the last version of Land's model[3] for human vision's lightness and color constancy. Jobson et al. have extended the Retinex into a general purpose image enhancement algorithm that provides simultaneous dynamic range compression, color consistency, and color and lightness rendition.[1, 2] The algorithm was initially targeted to process multi-spectral satellite imagery but has found applicability in very diverse areas such as medical radiography, forensic investigations, consumer photography, and aviation safety.[4] It is offered in the commercially available software package PhotoFlair by TruView.[5] Several improvements have been added to the original version of the Retinex including the use of multiple scales,[2] the addition of color restoration,[2] and the use of white balancing as a post-processing technique.[6] As the potential utility and complexity of the Retinex expands, so do the computational requirements of the algorithm. In particular, many new applications require the use of real-time video, thus increasing the processing requirements considerably.

Until now, no real-time digital implementation of the Retinex has been achieved. Real-time video frame rates have been defined as 15 to 30 frames per second (fps) by several human factors studies[7–10] with 30 fps the de facto standard. We have chosen a specialized processor, Texas Instruments (TI) TMS320C6711 DSP, for implementation and performance evaluation. This processor is offered on a low-cost evaluation board (DSK) and the board can be targeted using TI's software tools and utilities such as Code Composer Studio. We will briefly describe the Retinex algorithm, give an overview of the DSP hardware and software, and discuss the optimizations used to achieve a real-time version of the Retinex algorithm.

2. RETINEX

The Retinex is a member of the class of center/surround functions where the center is defined as each pixel value and the surround is defined as a Gaussian function. Expressed mathematically, the single-scale, monochromatic Retinex is defined by

$$R(x_1,x_2) = \alpha\big(\log(I(x_1,x_2)) - \log(I(x_1,x_2) * F(x_1,x_2))\big) - \beta$$

where I is the input image, R is the Retinex output image, log is the natural logarithm function, and α and β are scaling factors and offset parameters respectively, that transform and control the output of the log function. The $*$ symbol represents convolution. F is a Gaussian filter (surround or kernel) defined by

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

where σ is the standard deviation of the filter and controls the amount of spatial detail that is retained, and κ is a normalization factor that keeps the area under the Gaussian curve equal to 1.

Color constancy is also a direct result of the center/surround form of the algorithm. As an approximation, the intensity value, I, can be expressed as the product of an illuminant component i, and a reflectance component ρ

$$I(x_1,x_2) = i(x_1,x_2)\rho(x_1,x_2)$$

ignoring α and β we can write

$$R(x_1,x_2) = \log\big(i(x_1,x_2)\rho(x_1,x_2)\big) - \log\big((i(x_1,x_2)\rho(x_1,x_2)) * F(x_1,x_2)\big).$$

Since the illumination i varies slowly across the scene, we assume it can be represented by a constant, I_o. Thus, we can rewrite R as

$$R(x_1,x_2) = \log\left(\frac{I_o\rho(x_1,x_2)}{I_o\rho(x_1,x_2) * F(x_1,x_2)}\right)$$

which is independent of the illuminant I_o.[1]

Several extensions of the basic Retinex have been defined. This includes the multi-spectral, multi-scale Retinex (MSR) with color restoration (MSRCR),[2] and recently the addition of post-processing with a white balance technique for improved color restoration.[6] For this effort we consider the single-scale version of the algorithm. A visual example of the processing performed by the single-scale Retinex image is shown in Figure 1.

3. DSP

The TMS320C6711B (6711) DSP is a 32-bit floating point processor offering up to 900 million floating point operations per second (MFLOPs) performance at a clock rate of 150 MHz (6.67 ns cycle time). Speed grades up to 200 MHz are available. As shown in Figure 2 the processor is divided into three main components: the CPU (or core), memory, and peripherals.

The CPU is based on the advanced very-long-instruction-word (VLIW)[11] architecture developed by TI. It has eight independent functional units and the 256-bit wide VLIW architecture allows up to eight 32-bit instructions to be supplied to the units on every clock cycle. Control measures are built-in to vary what is

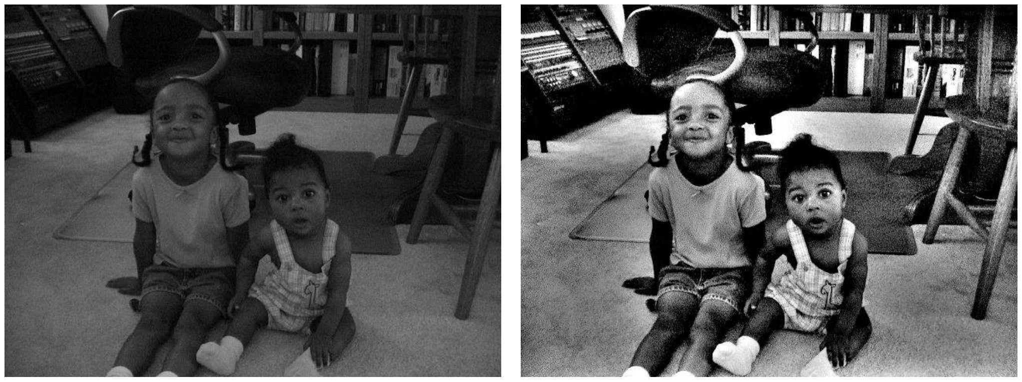

Figure 1. On the left is a low contrast JPEG image, on the right is a single-scale Retinex processed image — note the increase in contrast and sharpness even with a single scale.

executed by each functional unit. 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 formed for a total of up to 300 Million MACs per second. Each of the two register files contain sixteen 32-bit registers for a total of 32 general-purpose registers. Like MIPS processors, the CPU uses a load/store architecture, where all instructions operate on registers.

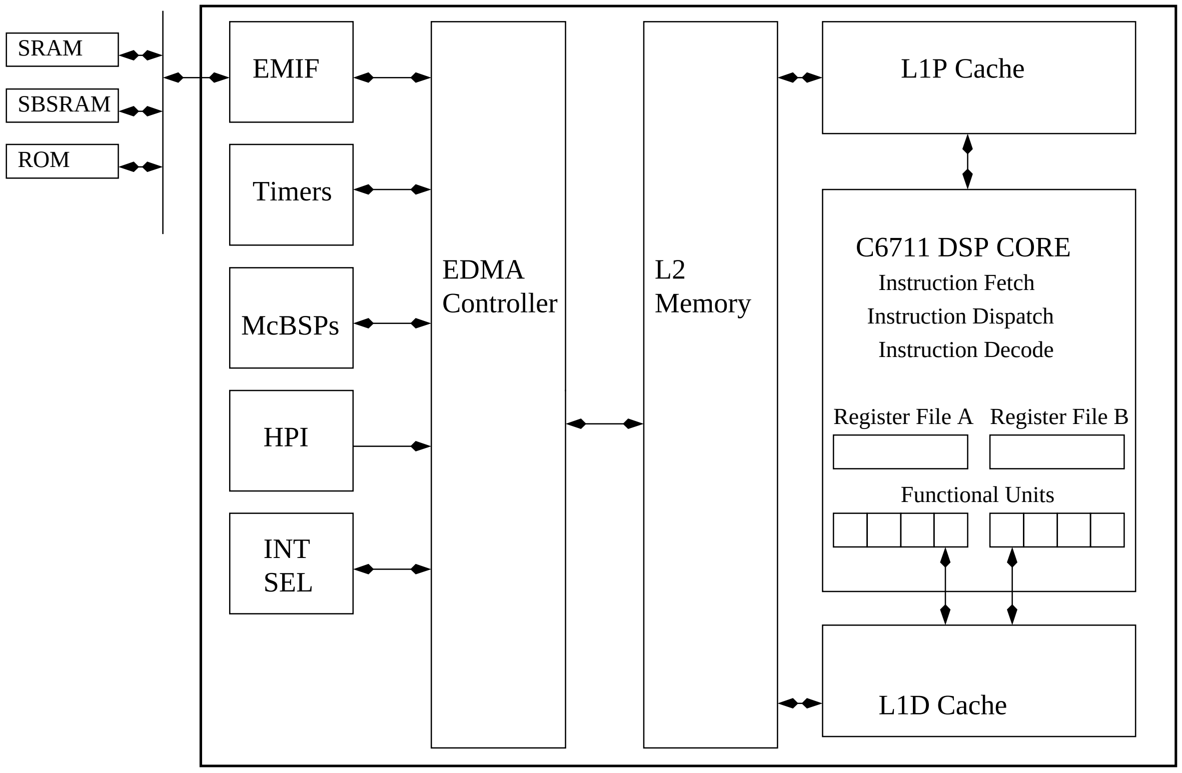

Figure 2. Basic DSP Components: CPU, L1 Data Cache, L1 Program Cache, L2 memory (SRAM/Cache) and EDMA

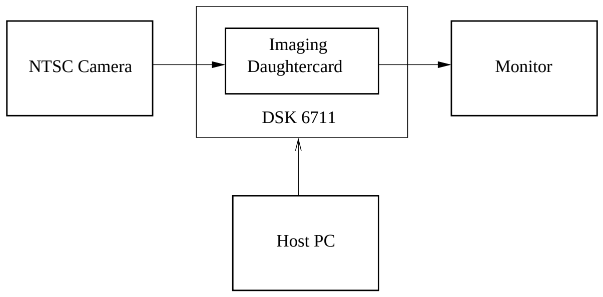

Figure 3. Testbed — The PC only provides setup information to the DSK/DSP; after initiation the DSP executes autonomously

The internal processor memory consists of a two-level cache.[12] The Level 1 program cache (L1P) is a 32-Kbit direct mapped cache and the Level 1 data cache (L1D) is a 32-Kbit 2-way set associative cache. The Level 2 memory/cache (L2) is a 512-Kbit memory space that can be configured as mapped memory, cache, or combinations of the two. The peripherals include a multichannel enhanced direct memory access (EDMA) controller, multichannel serial ports, a host port interface, two 32-bit general purpose timers, and an external memory interface (EMIF) that supports synchronous dynamic random access memory (SDRAM), synchronous burst SRAM (SBSRAM) and other asynchronous devices.[13]

4. TEST SETUP

The testbed for real-time Retinex video processing consists of a host PC, a camera, a monitor, an evaluation DSP board, a frame-grabber/display daughter-card, and associated software tools. The host PC is a standard Pentium PC running a Windows (2000) OS. The camera generates NTSC/PAL composite video that is fed into the daughter-card. The RGB output of the daughter-card is fed into the CRT monitor for display. Figure 3 is an outline of the system.

The 6711 is integrated on the DSP evaluation board (DSK). To support the DSP, the DSK has 16-MBytes of 100 MHz SDRAM, a parallel port controller to interface to a standard parallel port on a host PC, 128-KBytes of programmable and erasable flash memory, an 8-bit memory-mapped I/O port, and expansion memory and peripheral connectors for daughter-board support.

The daughter-board is an imaging daughter-card (IDC) that is used for video capture, display, and data formatting.[14] The IDC contains an NTSC/PAL digital video decoder chip (TVP5022), an NTSC/PAL digital video encoder chip (TVP3026), a Xilinx FPGA and 16-Mbits of SDRAM for frame capture. A female RCA connector and a 15-pin female VGA connector are also on the IDC, for composite video input and RGB monitor output, respectively.

Figure 4 is a block diagram of the video capture subsystem.[15] Input image data from the NTSC camera is 4:2:2 sampled[7] by the TVP5022 chip. This chip also controls all video input timing, in particular the vertical synchronization signal that generates a CPU interrupt once per frame. The FPGA separates the 4:2:2 digital stream into a luminance component and two chrominance components (standard YCrCb format[7]) and then writes the data as two separate fields in three separate blocks into the capture frame buffer. The capture buffers are mapped into the DSPs memory address space as read-only and are accessed via the EMIF. A triple buffering scheme is used to avoid delaying the executing application. At any time the FPGA controls two of the buffers while the user application has access to the third. The "active" buffer is the buffer currently receiving data from the TVP5022. The "last active" buffer is the last buffer that was filled by the TVP5022. The "user" buffer is the buffer currently being read by the 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 be returned. If

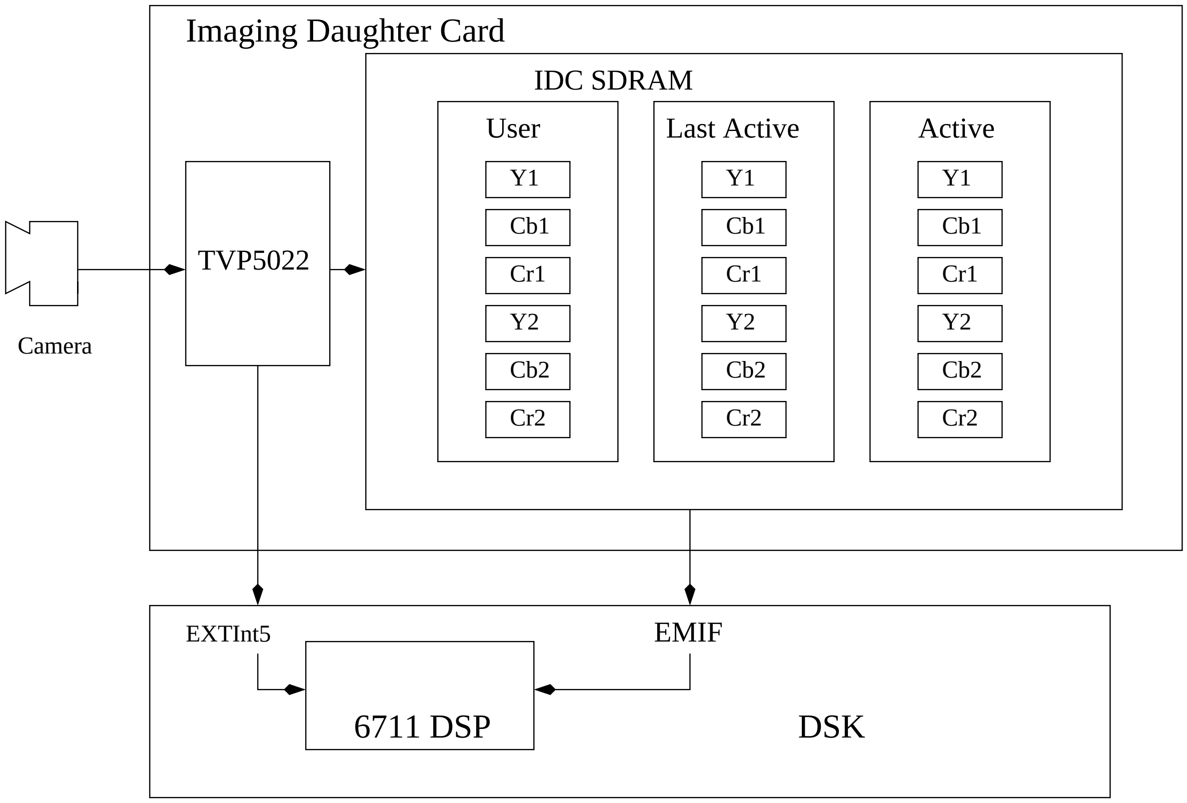

Figure 4: The Video Capture System — EXTInt5 is triggered by every other VSYNC (field)

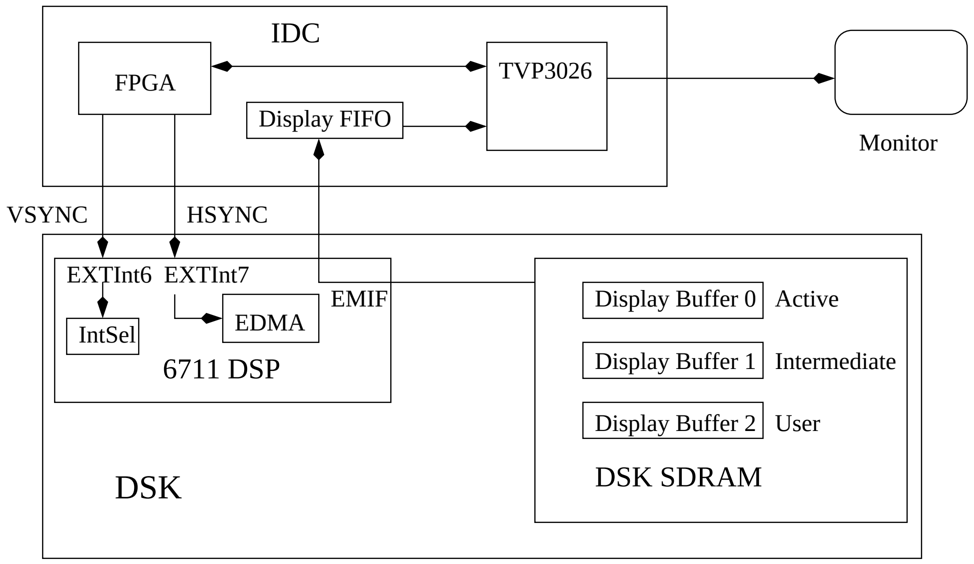

Figure 5: The Video Display Subsystem: the FPGA generates frame and line interrupt signals to the DSP.

the application executes slower, then captured frames will be overwritten. There are set of API functions that abstract accessing these buffers to the programmer.

Figure 5 is a block diagram of the video display subsystem.[15] The FPGA provides the video timing for the output. It generates a horizontal synchronization signal that triggers an EDMA event to copy one line of display data to the current display buffer. The TVP3026 chip then transmits this line to the RGB output port. The FPGA also generates a vertical synchronization signal that is used to post a semaphore indicating that the frame is complete. Like the video capture system, a triple buffering scheme is used so that the application will not have to wait for a new buffer. The exception is that the video buffers are now located in SDRAM on the DSK. Data is transferred in real-time from the DSK to the IDC via EDMA. The "active" buffer is where EDMA events move data from the buffer to the display FIFO. The "user" buffer is the buffer that the user application is currently rendering into and the "intermediate" buffer is the buffer the application will receive when it attempts to get the next buffer. If the application attempts to access buffers too fast, frames will be lost. If access is too slow, frames will be displayed repeatedly. Again there is a set of API functions available to the programmer.

A complete set of software development tools is available, including a C-compiler, assembly optimizer and a debugger for visibility into source code execution. All of these tools are incorporated into the Code Composer Studio (CCS) tool available from TI. Other rapid prototyping software tools are available such as a chip support library (CSL)[16] used to configure and control on-chip peripherals, an image data manager that offers abstraction of double buffering for DMA requests, and an image library containing general purpose image/video processing routines. There is also a DSP library (DSPLib)[17] that contains a collection of optimized DSP functions such as matrix operations and Fast Fourier Transforms (FFTs).[18]

General operation of the testbed system is as follows. C code to perform the Retinex algorithm is written using the CCS tools on the host PC. The code is compiled, assembled and linked into a common object file format (COFF) targeted for the 6711 on the DSK board. The output COFF file is downloaded from the host into the DSK through the parallel port. Execution is then initiated from the host. From this point on, the DSK operates independently of the host. The DSK, through the IDC, captures video frames from the camera and re-samples the 640×480 pixel input image to 320×240 pixel image used for processing. The sub-sampled image is Retinex processed and displayed adjacent to the unprocessed image for comparison.

5. OPTIMIZATIONS AND RESULTS

Our target performance range for real-time Retinex processing is 15–30 frames per second (fps) for NTSC video. The 30 fps rate is considered the de facto standard for real-time video, but lower frame rates are acceptable on the basis of avoiding flicker and the accurate portrayal of motion. Thus we choose a lower limit of 15 fps, or 66 ms processing time. We setup our L2 memory by splitting the memory into 32K of cache and 32K of SRAM. The 32K of SRAM is sufficient to store all the required variables for our current implementation. After initial setup of the testbed, several optimizations are performed to achieve at least the minimum frame rate.

5.1. Use the Convolution Theorem

In observing the Retinex equation, note that the input image is convolved with a Gaussian kernel. Good single scale Retinex (SSR) 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[18, 19]

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

where F and G are the spatial frequency domain representations of f and g respectively. We employ the 2-dimensional $M \times N$ forward and inverse Discrete Fourier Transforms (DFT),[19]

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

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

to rewrite the Retinex equation as:

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

where $I'(\mu,\nu)$ and $F'(\mu,\nu)$ represent the DFTs of $I(x_1,x_2)$ and $F(x_1,x_2)$ respectively, and $\mathcal{F}^{-1}$ represents the inverse DFT. Exploiting the separability of the DFT and the computational efficiency of the well known Fast Fourier Transform (FFT), we compute 2-dimensional transforms by applying 1-dimensional FFTs to the rows and columns of the image. The FFTs are obtained with the optimized DSPLib that restricts the number of input points to a power of two. So the 320 input frame is cropped and padded to a 256×256 image before Retinex processing. The specific algorithm used is the floating-point radix-2 FFT. The number of cycles that TI benchmarks to compute this operation is given by:

$$C = (2n\log_2 n) + 42$$

where C is the number of cycles, $\log_2$ is the base 2 logarithm, and n is the length of the complex input array. For a 256-point FFT on the 6711 operating at 150 MHz, this corresponds to 4138 cycles or 27.5 microseconds under ideal benchmark conditions. For a 256×256 image, the total number of 1-dimensional FFTs is 512, thus the total transform time is $\approx$ 7 milliseconds. Prior to implementation, we felt that this would be the tall pole for processing time. Table 1 summarizes the actual performance of the algorithm after implementation.

	Count	Time (ms)
processing	1	389.22
rescale	1	3.52
copy	1	0.63
retinex	1	385.05
fftrows	1	21.00
fftcols	1	158.46
ifftcols	1	157.65
ifftrows	1	17.04
fft	256	9.94
convolve	1	12.39
reteq	1	14.46

Table 1: Initial performance results from basic implementation of Retinex.

The "processing" value is the total time to process one frame. Since the average, maximum and minimum times were essentially the same for all parameters, only average values are reported. This value is the time associated with all activities to create one frame of the output display. The "rescale" value is the time taken to scale the input image size from 640×480 to 320×240 pixels. Horizontal scaling is performed by averaging adjacent pixels, while vertical scaling is performed by selecting only the rows of the even field. The "copy" value is the time taken to copy the scaled image to the output buffer for display. The "retinex" value is the total time to perform Retinex processing. The remaining times are portions of the "retinex" time. The "fftrows" and "fftcols" values are the times taken to perform the 1-dimensional transforms of all the rows or columns, respectively, of the input image. These are computed to transform the image from the spatial domain into the spatial frequency domain. This time includes reading each row or column, processing, and storing the data for later processing (more on this below). The "fftrows" time also includes taking the logarithm of the input values at this point for use in the Retinex equation later. The "ifftcols" and "ifftrows" values are the times to perform the inverse FFTs of the columns and rows respectively. These are computed to transform the spatial frequency domain image values to the spatial domain. The “fft” value is the time to perform 256, 256-point FFTs. This value is actually part of the “fftrows” value but is listed as a point of reference. The “convolve” value is the time taken to convolve by multiplication in the spatial frequency domain, the Gaussian kernel with the input image. The “reteq” value is the time taken to perform the computation of the Retinex equation. As noted earlier the logarithm of the original image has been pre-computed.

The total processing time is 389.22 ms which corresponds to 2.56 fps, far from our minimum target value of 15. Surprisingly, our initial total processing time is not driven by the FFT computations, but rather by the data transfer times. This can be seen by comparing the “fftrows” time and the “fftcols” time. If data transfers were not an issue these two times should essentially be the same. However, the “fftcolumn” time is almost *eight* times that of the “fftrows” time! Additional timers were added to the code within the “fftcolumns” to measure the column read and write times. To read a 256-point column required 60.03 ms and to write the same column (after processing) took 88.27 ms.

The primary cause of this discrepancy can be determined by examining the DSP architecture and the memory requirements of the algorithm. 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. While we do not have direct write access to the L1P or L1D caches, we do have access, and some control over, the next fastest access location: L2 memory. The 6711 has a 64-KByte L2 memory that can be configured as cache, SRAM, or a combination of the two. Unfortunately, this is nowhere near the capacity required to store all the image data for the following reasons. First, the input image size itself is 64-KBytes. Second, the FFTs require data to be in complex format, i.e. each point must have a real and imaginary (zero for our purposes) component and this doubles the data size. And third, the data must be in floating point (four byte) format. So the actual memory requirement just to store the image prepared for processing is 512-KBytes. Thus, the image data must be kept and fetched from external memory, i.e., memory that is off the chip.

To perform the FFT operation on a row of an image requires reading in all the contiguous pixels of the row from external memory into an input buffer, ideally located in internal L2 memory. On the 6711, and on most processors with a decent compiler, this is accompanied by reading in several points — or optimally the entire array — when the first point is accessed. Accessing the first point causes a cache miss, but since all other points in the row are already in L2, accessing the other points in the row is a cache hit. Strictly speaking, this is a L2 memory hit not a cache hit. To process a column requires accessing non-contiguous points with a stride difference equal to the number of columns. So, in essence, transferring each point from external memory to L2 generates a L2 memory miss thus severely degrading performance.

5.2. Enhanced Direct Memory Access (EDMA) Columns

Several authors[20] have suggested cache miss rate reduction techniques, many of which are incorporated into the compiler tools used within CCS. Even with these tools set to full optimization levels, the performance obtained is as stated above. Our problem is exacerbated by the wide stride length required for column access and also by the fact that we are only accessing and using the data once at this point. In order to improve the L2 memory transfer time for column-wise image data we must change the mechanism for access.

The 6711 contains an enhanced direct memory access controller (EDMA) to handle data transfers between the L2 controller and peripherals. Thus data can be transferred efficiently (and in the background to the processor) between L2 SRAM and external memory. The CSL contains a data module (DAT) that uses the EDMA hardware. The DAT module has a routine (DATcopy2d) to perform 2-dimensional transfers. One can specify 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 represent a column we can exploit the efficiency of this transfer to speed up column processing of the image. The result of using this method is shown in Table 2. As shown in the “processing” value, the total processing time is now down to 139.18 ms or 7.18 fps. This is a significant increase in the processing rate, but we are still 50 percent below our target. Several smaller optimizations such as exiting from loops early, using table lookups instead of direct log calculations, and merging α and β parameters into these tables gave incremental increases in performance to 123.24 ms.

	Count	Time (ms)
processing	1	139.18
rescale	1	3.64
copy	1	0.62
retinex	1	134.91
fftrows	1	19.05
fftcols	1	29.40
ifftcols	1	28.73
ifftrows	1	24.25
fft	256	9.90
convolve	1	9.84
reteq	1	16.21

Table 2. Performance results after using 2D data transfers. The 250 ms savings in processing time is the best gain in performance we achieved.

5.3. Merge Implementation Components

The next significant performance increase was gained through the identification of unnecessary transformation cycles. In our original implementation we performed the following sequence of operations.

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

We then continued with the remainder of Retinex calculation. We can take advantage of the independence of each column of image data by merging some of the preceding steps and thus, eliminate some of the data transfers. As soon as we have performed the FFT of a column, we can continue processing this column with the convolution operation and also IFFT the column. Our sequence above then becomes:

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

This saves four write/read transfers. Table 3 shows the results of implementing this optimization. The "ifftcols" value goes to zero because this function is now embedded in the "fftcols" routine.

As shown, this results in a savings of 40.84 ms. Our total processing time is now down to 87.40 ms or 11.44 fps. Again, after this smaller inefficiencies were eliminated such as placing FFT twiddle factors, bit reversal tables, and other various constants and variables in the L2 SRAM to speedup the computations that invoke them. This slightly decreased total processing time down to 85.09. As transfer times decreased and unnecessary operations were eliminated we began to focus more on the FFT operation itself.

	Count	Time (ms)
processing	1	87.40
rescale	1	3.60
copy	1	0.64
retinex	1	83.14
fftrows	1	17.42
fftcols	1	41.14
ifftcols	0	0
ifftrows	1	24.50
fft	256	9.80
convolve	1	2.18
reteq	1	5.30

Table 3. Performance results after merging implementation steps. Note the decrease in convolution time is a result of early exit from loops based on zero values remaining in the kernel array.

5.4. Use a Faster FFT and Double Buffering

The final major performance increase was obtained by using a more efficient form of the FFT algorithm, and by implementing a double buffering mechanism to improve processor utilization. The DSPLib offers a cache-optimized (SPxSP) FFT 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:

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

For a 256-point FFT $C = 2923$ cycles, or 19.5 microseconds, a 30% increase in performance. To transform the entire 256×256 image should take ≈ 5 ms.

We also implemented a double buffering scheme during this phase to improve processor utilization. As noted earlier the EDMAs of the 6711 allow data transfers to occur independently or in the background of any processor activity. Taking advantage of this we setup a series of buffers so that as we FFT process one buffer, we can simultaneously transfer a previously processed buffer. After careful setup and implementation of this scheme we achieved the results shown in Table 4.

	Count	Time (ms)
processing	1	56.00
rescale	1	3.69
copy	1	0.64
retinex	1	51.65
fftrows	1	12.94
fftcols	1	20.71
ifftcols	0	0
ifftrows	1	17.97
fft	256	7.23
convolve	1	2.26
reteq	1	5.45

Table 4. Performance Results after changing FFT routines and using double buffering. The processing time reduces to real-time rates.

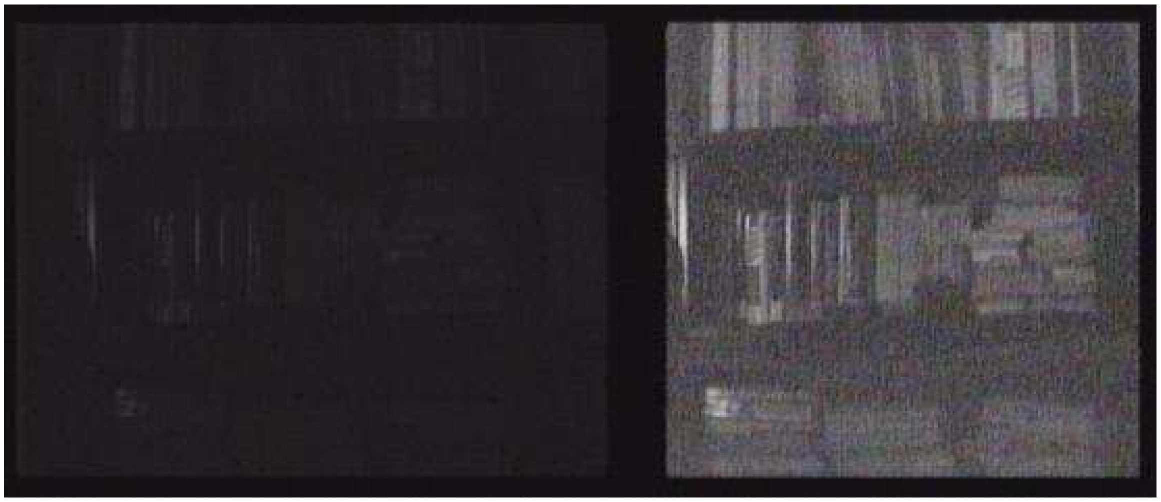

Figure 6. Capture Video Frame with input from camera on the left, and Retinex output on the right. Retinex parameters are $\alpha = 175$, $\beta = 135$, and $\sigma = 80$ — note that we are nearly reaching the noise limit of the camera.

This last improvement allowed us to meet our performance target. With 56 ms total processing time, the algorithm is processing 17.85 fps. Other minor changes, such as slightly modifying EDMA locations, have pushed the Retinex processing time down to 48.33 ms or 20.7 fps. A sample output image frame from a video taken of a bookcase is displayed in Figure 6. The input image is shown on the left while the Retinex enhanced image is shown on the right. The enhanced image has increased contrast and sharpness. Details indistinguishable in the original are easily noticed in the enhanced image.

6. CONCLUSIONS

We have successfully implemented a real-time (20 fps with full frame generation) version of the Retinex image enhancement algorithm using a 150 MHz 6711 digital signal processor on-board an evaluation circuit board. Video images were captured using a NTSC camera and frame-grabber daughter-card, processed, and displayed using a standard monitor. Initial performance was not constrained by FFT performance but by data transfer bottlenecks between internal and external memories. Appropriate use of DMA transfers, utilization of cache-optimized FFT routines, and restructuring and merging of major components of the implementation improved performance by nearly an order of magnitude.

The discussed implementation has been for SSR processing a grayscale 256×256 image. Future plans are to expand the current implementation into a version that performs multi-scale color Retinex processing. Linear extrapolation implies that this level of Retinex processing requires at least six times the current processing power. Larger image formats should also be as closely supported as possible. If FFT algorithms that require power-of-two input dimensions are retained, then for a for 640×480 sized image, a 1024×512 sized FFT would be required. A closer size of 512×512 could be generated through proper cropping and padding with little loss of information. Other related enhancements such as color restoration could be added to the processing chain. Implementation of automatic parameter control could also be attempted. Obviously any additional functionality will also require more powerful processing.

Other processors are currently available with significantly better performance than the 150 MHz 6711. A TI6713 operating at 225 MHz has been obtained that theoretically should provide over 30 fps performance. Testing is currently in progress. A 1-GHz fixed point DSP has recently been announced: it should offer ample computational performance for real-time processing. Multiprocessor systems are also available that provide potential solutions. Also promising is the potential to use FPGAs. FPGAs now perform FFTs as efficiently as most processors. The chip architecture could be customized to perform both the Retinex algorithm and the pre/post processing required to extract and merge multiple spectral bands and scales.

Finally, future missions may require the use of Retinex processing in space. This would require the mapping of the algorithm to space qualified hardware. Several Actel FPGAs have spaceflight heritage, but these devices are write-once devices limiting their in-flight flexibility. Testing is performed primarily through simulation. Once a design is finalized their in-situ use becomes more appropriate. Most Xilinx FPGAs inherently are not radiation hardened because they are SRAM based devices. Other mitigation techniques have been investigated to make the devices more radiation tolerant. TI recently announced a radiation tolerant fixed-point DSP, the 6201. A multiprocessor board with this device would also offer a potential solution for spaceflight hardware.

7. ACKNOWLEDGMENTS

The authors wish to thank the Systems Engineering Competency at the NASA Langley Research Center and the Synthetic Vision Sensors element of the NASA Aviation Safety Program for the funding which made this work possible. In particular, Dr. Rahman's work was supported under NASA cooperative agreements NCC-1-01030 and NNL04AA02A. The authors would also like to thank Andrew Nesterov for his ideas on managing interrupts.

REFERENCES

1. D. J. Jobson, Z. Rahman, and G. A. Woodell, "Properties and performance of a center/surround retinex," *IEEE Trans. on Image Processing* **6**, pp. 451–462, March 1997.
2. D. J. Jobson, Z. Rahman, and G. A. Woodell, "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**, pp. 965–976, July 1997.
3. E. 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*, **83**, pp. 3078–3080, 1986.
4. Rahman. see http://dragon.larc.nasa.gov/fog_haze for examples.
5. TruView. see http://www.truview.com.
6. Z. Rahman, D. Jobson, and G. Woodell, "Retinex processing for automatic image enhancement," in *Journal of Electronic Imaging*, **13, No. 1**, pp. 100–110, January 2004.
7. J. Watkinson, *The Art of Digital Video*, Focal Press, 1990.
8. G. Kellog and C. Wagner, "Effects of update and refresh rates on flight simulation visual displays," Tech. Rep. 100415, NASA Langley Research Center, February 1988.
9. A. Hansen, W. Smith, and R.Rybacki, "Real-time synthetic vision cockpit display for general aviation," in *Proceedings of SPIE 3691*, April 1999.
10. J. Leachtenauer, *Electronic Image Display*, SPIE Press, 2004.
11. Texas Instruments, "TMS320C6000 technical brief," Tech. Rep. SPRU197D, Texas Instruments, Dallas, Texas, February 1999.
12. Texas Instruments, "TMS320C621x/C671x dsp two-level internal memory reference guide," Tech. Rep. SPRU609A, Texas Instruments, Dallas, Texas, November 2003.
13. Texas Instruments, "TMS320C6000 peripherals reference guide," Tech. Rep. SPRU190D, Texas Instruments, Dallas, Texas, February 2001.
14. Texas Instruments, "TMS320C6000 imaging developer's kit (idk) user's guide," Tech. Rep. SPRU494a, Texas Instruments, Dallas, Texas, September 2001.
15. Texas Instruments, "TMS320C6000 imaging developer's kit (idk) video device driver user's guide," Tech. Rep. SPRU499, Texas Instruments, Dallas, Texas, December 2000.
16. Texas Instruments, "TMS320C6000 chip support library api user's guide," Tech. Rep. SPRU401E, Texas Instruments, Dallas, Texas, December 2002.
17. Texas Instruments, "TMS320C67x dsp library programmer's reference guide," Tech. Rep. SPRU657, Texas Instruments, Dallas, Texas, February 2003.
18. O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, 1975.
19. R. Gonzalez and R. Woods, *Digital Image Processing*, Addison-Wesley, 1993.
20. D. Patterson and J. Hennessy, *Computer Organization and Design*, Morgan Kaufmann, 1998.