Using the CPU and GPU for Real-Time Video Enhancement on a Mobile Computer

Asheer Bachoo
Optronic Sensor Systems
Defence, Peace, Safety and Security
Council for Scientific and Industrial Research
Pretoria, South Africa
Email: abachoo@csir.co.za

***Abstract*—Real-time video enhancement is generally achieved using costly specialized hardware that have specific functions and outputs. Commercial off-the-shelf hardware, such as desktop computers with Graphics Processing Units (GPUs), are also commonly used as cost effective solutions for real-time video processing. In the past, the limitations in computer hardware meant that real-time video enhancement was mainly done on desktop GPUs with minimal use of the Central Processing Unit (CPU). These algorithms were simple and easily parallelizable in nature, which enabled them to achieve real-time performance. However, complex enhancement algorithms also require the sequential processing of data and this cannot be easily achieved in real-time on a GPU. In this paper, the current advances in mobile CPU and GPU hardware are used to implement video enhancement algorithms in a new way on a mobile computer. Both the CPU and GPU are used effectively to achieve real-time performance for complex image enhancement algorithms that require both sequential and parallel processing operations. Results are presented for histogram equalization, local adaptive histogram equalization, contrast enhancement using tone mapping and exposure fusion of multiple 8-bit grey scale videos of size up to 1600$\times$1200 pixels.**

I. Introduction

Real-time execution of video and image processing algorithms can improve operator feedback in decision critical situations. For example, enhancement of low contrast video scenes in high risk areas will improve the visual appearance of dangerous objects that may otherwise be hidden. Different types of hardware can be used to achieve real-time video enhancement. Field Programmable Gate Arrays (FPGAs) and specialized digital signal processing chips are widely used in digital signal processing for real-time applications. Desktop computers are used by most scientists and researchers due to their low cost and programming flexibility. The CPUs of desktop computers currently have multiple cores for high performance computing. In recent years, the rendering pipeline of the GPU has been made accessible to programmers through the introduction of GPU programming languages. It employs the Single Instruction Multiple Data (SIMD) technique to achieve data and computing parallelism with huge performance gains in certain applications.

Bing-jian *et al.* use an FPGA to enhance the contrast of infrared images at 25 frames per second [1]. Their test images had a size of 128$\times$128 (8-bit pixel depth). The method they implemented was plateau histogram equalization. A novel image enhancement algorithm using an FPGA, which compensates for the under- and over-exposed image regions, is proposed by Iakovidou *et al.* [2]. Processing at approximately 32 frames per second is achieved for a 2 megapixel image. Babenko and Shah present a GPU library for image processing on the GPU [3]. They provide results for the homography transform between 3-dimensional views and claim a 600 times speedup on the GPU when compared to the CPU implementation. Implementations of various image processing algorithms are also mentioned e.g. edge detection, optical flow and image pyramid computation, but no meaningful results are presented apart from their claims of improved running times. Castano-Díez *et. al.* evaluate the GPU for 3D image processing using the Compute Unified Device Architecture (CUDA) [4]. They report speedups over the CPU in the range of 5 to 67 times.

As seen from the above, previous video processing algorithms on computers are typically data parallel in nature and can easily be ported to the GPU hardware. However, complex algorithms generally require the sequential processing of data for particular information and these operations suffer a considerable performance decrease on modern GPUs. In this paper, implementations of four grey scale video enhancement algorithms are presented that require sequential processing of data: histogram equalization (HE), local histogram equalization using interpolation (LHEI), a multiscale image enhancement (MSIE) algorithm and exposure fusion (EF). The algorithms execute in real-time on a mobile computer by exploiting the CPU and GPU processing capabilities. In the next section CPU and GPU processing is discussed. The implemented algorithms are then presented. In the last two sections experimental analysis and discussion followed by concluding remarks are provided.

II. Implementation for the CPU and GPU

Computing hardware is designed to have particular types of operation. For example, CPUs are good for sequential operations while data parallel operations are efficient on GPUs. For our implementations, sequential operations are first executed on the CPU (this is typical of algorithms that require an initial scan through the frame for computing particular image information e.g. the image histogram); thereafter, the fragment shader pipeline is utilized to process image pixels using parallel operations on the GPU [5], [6]. Information generated by the CPU is stored in textures that are transferred to the GPU for parallel pixel processing.

Copying of image data between the CPU and GPU consumes a large number of CPU clock cycles when using texture fetches. This problem is averted by using the pixel buffer object (PBO) provided by the OpenGL driver [6]. The PBO provides regions of GPU memory that are directly accessible through identifiers. It achieves fast data transfer across the CPU/GPU bus by using direct memory access (DMA). The texture type that is used to store the data generated by the CPU is 32-bit single precision floating point. It has a constant size for each particular algorithm and this ensures that the DMA data transfer will have a fixed execution time. For example, when performing 8-bit grey scale histogram equalization (discussed in Section III), the texture size is 256×1 pixels. Algorithms that rely on data computed from image blocks assume a texture size with the number of rows equivalent to the largest number of image blocks possible. This sort of information is derived by assuming a maximum image size and a minimum block size. The CPU and GPU code was optimized by reducing the number of arithmetic and boolean operations. For example, replacing certain division operations by multiplications (which execute much faster than division operations). Other operations were reduced by pre-computing frequently used variables and by using built-in hardware functions on the GPU.

The software implementation can be summarized as follows:

1) Important image information that requires sequential processing is generated using the CPU. The information is stored in a 32-bit floating point texture.
2) The information texture is transferred to the GPU, using the PBO for optimal data transfer, where it is used to transform each pixel value in parallel.

III. Implemented Algorithms

A. Histogram Equalization

Global histogram equalization computes a histogram for the entire image and then generates a new pixel mapping using this histogram. Given an image with L grey levels, the probability of occurrence of grey level g is:

$$p(g) = \frac{n_g}{n}, g = 0, 1, \ldots, L-1 \tag{1}$$

where n is the total number of pixels and n_g is the number of pixels having grey level g. The transformation function for general histogram equalization is

$$h(g) = (L-1)\left(\sum_{i=0}^{g} p(i)\right) \tag{2}$$

where $p(i)$ is the function described in Equation 1. The transformation function (Equation 2) is computed on the CPU and sent to the GPU for adjusting the pixel brightness values in parallel.

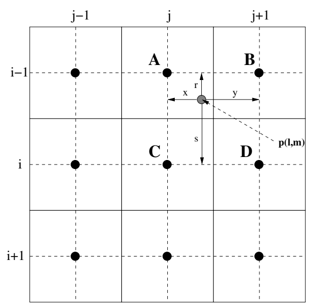


Fig. 1. *Bilinear interpolation for pixel blending.*

B. Local Histogram Equalization using Interpolation

Adaptive histogram equalization computes a local histogram for every image pixel. However, this is a computationally intensive task. It can also be performed efficiently using bilinear interpolation [7]. The input image is divided into square regions and the histogram mapping function for each region is computed. Each pixel is then transformed by using the mappings in surrounding regions to approximate its mapping as a bilinear interpolation. More specifically, the mapping for a pixel is derived from the four nearest square regions as a weighted sum of their mappings (shown in Figure 1). Block size is the only user input required.

Image blending at an interior pixel location (l, m) - a pixel location surrounded by four blocks - is achieved by using the histograms computed for blocks A, B, C and D (Figure 1). The mapping for an interior pixel at location (l, m) with grey value g is:

$$f(l,m) = \frac{s}{r+s}\left(\frac{y}{x+y}h_A(g(l,m)) + \frac{x}{x+y}h_B(g(l,m))\right) + \frac{r}{r+s}\left(\frac{y}{x+y}h_C(g(l,m)) + \frac{x}{x+y}h_D(g(l,m))\right) \tag{3}$$

where $h_k(g(l,m))$, $k \in \{A, B, C, D\}$, is a histogram transformation value (Equation 2) for grey scale value $g(l,m)$ using histogram h_k. Figure 1 shows how the parameters r, s, x and y are computed. Edge pixels with only two blocks in their proximity are mapped using a linear combination of the two image functions. Pixels at the corner of the image have only a single mapping function i.e. the image function of the closest block. The local histograms are computed on the CPU and stored in a texture together with block location information. This information and the original image are transferred to the GPU for processing where pixel blending is executed in parallel.

C. Multiscale Image Enhancement

Tao and Asari [8] describe a multiscale approach for enhancing images. They perform dynamic range compression and then adaptive contrast enhancement. Luminance information $I(x, y)$ is first normalized i.e. $I_n(x, y) \in [0, 1]$. Thereafter, dynamic range compression is performed using the following

TABLE I
AVERAGE EXECUTION RATE FOR VIDEO ENHANCEMENT ALGORITHMS (IN FRAMES PER SECOND)

Video Size	Method					
	LHEI(32)	LHEI(64)	LHEI(128)	LHEI(256)	HE	MSIE
256×256	61.24	62.36	62.93	-	208.29	161.47
512×512	50.57	51.36	52.75	53.50	127.83	67.42
1024×1024	30.41	30.62	31.07	32.19	50.05	19.43
1360×1024	25.63	26.29	26.85	26.92	39.27	14.20
1600×1200	19.37	19.72	20.48	19.92	30.27	12.27

transform:

$$I_n{}' = \frac{I_n{}^{(0.75z+0.25)} + 0.4(1-I_n)(1-z) + I_n{}^{(2-z)}}{2} \tag{4}$$

where z is an image dependent parameter [8]. Multiple Gaussians G_i are then used to smooth the original image $I(x,y)$ at different scales, producing different blurred images $I_{G_i}(x,y)$. For each blurred image, a parameter E_i is computed as follows:

$$E_i(x,y) = r_i(x,y)^p = \left[\frac{I_{G_i}(x,y)}{I(x,y)}\right]^p \tag{5}$$

The parameter p is an image dependent parameter defined as:

$$p = \begin{cases} 3 & \text{for } \sigma \leq 3 \\ \frac{27-2\sigma}{7} & \text{for } 3 < \sigma < 10 \\ 1 & \text{for } \sigma \geq 10 \end{cases} \tag{6}$$

where σ is the global standard deviation in the original image. $I_n{}'$ is then enhanced at multiple scales

$$S_i(x,y) = 255 I_n{}'(x,y)^{E_i(x,y)} \tag{7}$$

The final enhanced image is produced as follows:

$$S(x,y) = \sum_i w_i S_i(x,y) \tag{8}$$

where w_i is a weight factor for each output. To achieve a high processing frame rate for this algorithm, the integral and squared integral image [9] are computed on the CPU. These are scan primitives that enable rapid computation of the mean and standard deviations for an image region. Another advantage is that a blurred image can be computed at several scales quite efficiently using the integral image for averaging pixels. The other parameters (p and z) are also easily computed on the CPU. Once the integral images and parameters are computed, they are transferred to the GPU for the image blurring and pixel tone mapping.

D. Exposure Fusion

Multiple different exposures of the same scene can be fused to enhance pixels that are under- or over-exposed in a single exposure. We use a variation of Goshtasby's algorithm [10]. n input frames with different exposures are provided as input to the fusion algorithm. Each frame is tiled and the Shannon entropy [11] is computed for each tile. Our algorithm selects the image block with the highest entropy and performs blending between all selected blocks using bilinear interpolation. This is similar to Equation 3.

TABLE II
AVERAGE EXECUTION RATE FOR EXPOSURE FUSION (IN FRAMES PER SECOND).

Video Size	Block Size			
	32	64	128	256
256×256	195.37	196.56	198.23	-
512×512	101.75	103.55	103.83	104.61
1024×1024	28.14	34.09	35.84	35.95
1360×1024	19.75	26.59	27.87	27.65
1600×1200	19.92	20.21	20.25	20.31

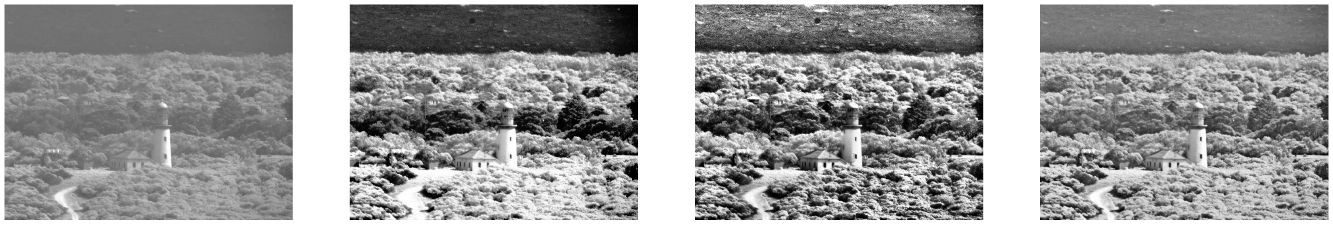
Image blending at an interior pixel location (l,m) - a pixel location surrounded by four blocks - is achieved by using the pixel values at location (l,m) in the frames having the exposures of the surrounding blocks. The mapping for an interior pixel is:

$$f_new(l,m) = \frac{s}{r+s}\left(\frac{y}{x+y}e_A(l,m) + \frac{x}{x+y}e_B(l,m)\right) + \frac{r}{r+s}\left(\frac{y}{x+y}e_C(l,m) + \frac{x}{x+y}e_D(l,m)\right) \tag{9}$$

where $e_k(l,m)$, $k \in \{A,B,C,D\}$, is a pixel value at location (l,m) in an acquired frame having exposure k. The entropies of the image blocks and block spatial information, computed on the CPU, are stored in a single texture when copied to the GPU for maximum entropy block selection and pixel blending.

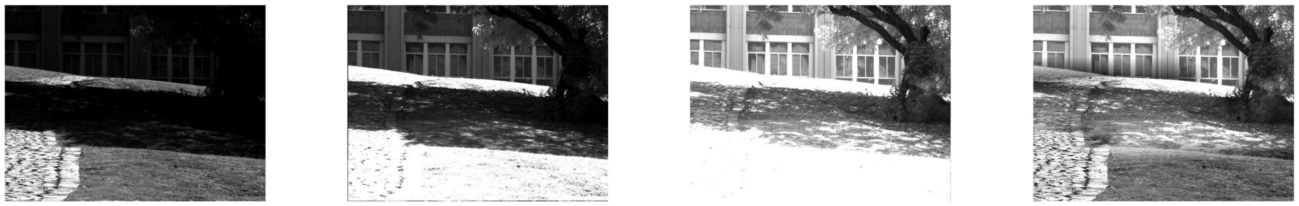
IV. EXPERIMENTAL RESULTS AND DISCUSSION

The algorithm implementations were tested on a Dell Latitude D830 Laptop with Intel Core2 Duo 2.2 GHz processor, 4GB memory and an entry level NVIDIA Quadro 140M NVS 256MB graphics card. The GNU/Linux (Debian) operating system was utilized. C++ and OpenGL Shading Language for GPU processing through OpenSceneGraph are also used. The algorithms were tested on 5 grey scale (8-bit pixel depth) videos with different dimensions. It must be pointed out that the algorithms operate independently of the video content i.e. the execution time will be approximately the same for video with the same dimensions, pixel depth and format. In order to mitigate the effects of disk read, it was ensured that data was cached so that there was minimal latency in data reading. Window sizes in {32, 64, 128, 256} pixels were used for LHEI and EF. The EF algorithm fuses 3 video frames (short, medium and long exposures). Three blur radii (scales) were supplied for MSIE by multiplying the image height by a ratio in {0.03, 0.1, 0.45}. Figures 2 and 3 show some enhanced videos.



(a) Original Video (b) HEQ (c) LHEQI (d) MSIE

Fig. 2. Video Enhancement



(a) Exposure 1 (b) Exposure 2 (c) Exposure 3 (d) Fused Video

Fig. 3. Exposure Fusion

The average execution rates for the algorithms, in frames per second, are shown in Table I and II. The size of the image blocks for LHEI appear in brackets. For a video size of 1024×1024 (approximately 1 million) pixels, most of the algorithms run at greater than 20 frames-per-second. The MSIE algorithm is generally slower than the others since a large number of statistics are computed and there is a significant amount of texture fetches. The same can be said of the EF algorithm, which has to process 3 video streams and then fuse them. Video frames with dimension 1360×1024 and 1600×1200 are those acquired using high-performance Prosilica video cameras. These high-resolution frames are processed at a minimum frame rate of close to 20 frames-per-second except for the MSIE algorithm (minimum frame rate of 12.27 frames-per-second).

The current performance is limited by GPU hardware. Utilizing a graphics card with a higher number of processing pipelines will increase the execution rate. For example, the frame rate almost doubles when using a NVIDIA 9600GT GPU. Our future work will consider looking at optimization techniques on multi-core CPUs and improved shader code for the GPU. More complex image partitioning, automatic block size selection and multiscale blending of image regions will also be considered. New measures for exposure selection will also be examined.

V. Conclusion

This paper presented new implementations of several video enhancement algorithms for real-time execution. With the exception of the computationally intensive MSIE method, all of the algorithms achieve a minimum frame rate of close to 20 frames per second. This is achieved using the sequential and parallel processing capabilities of low cost computer hardware. The high processing frame rates achieved for large video resolutions, using a mobile computer, introduces new possibilities in terms of mobility and testing in the real world.

References

[1] W. Bing-jian, L. Shang-qian, L. Qing, and Z. Hui-xin, "A real-time contrast enhancement algorithm for infrared images based on plateau histogram," *Infrared Physics and Technology*, vol. 48, pp. 77–82, 2006.

[2] C. Iakovidou, V. Vonikakis, and I. Andreadis, "FPGA implementation of a real-time biologically inspired image enhancement algorithm," *Journal of Real-Time Image Processing*, vol. 3, pp. 269–287, 2008.

[3] P. Babenko and M. Shah, "MinGPU: a minimum GPU library for computer vision," *Journal of Real-Time Image Processing*, vol. 3, pp. 255–268, 2008.

[4] Daniel Castano-Díez, Dominik Moser, Andreas Schoenegger, Sabine Pruggnaller, and Achilleas S. Frangakis, "Performance evaluation of image processing algorithms on the GPU," *Journal of Structural Biology*, vol. 164, pp. 153–160, 2008.

[5] R.J. Rost, *OpenGL(R) Shading Language (2nd Edition)*, Addison-Wesley Professional, 2006.

[6] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, Addison-Wesley Professional, 2007.

[7] SM Pizer, EP Amburn, JD Austin, R Cromartie, A Geselowitz, T Greer, BM ter Haar Romeny, JB Zimmerman, and K Zuiderveld, "Adaptive histogram equalization and its variations," *Computer Vision, Graphics and Image Processing*, vol. 39, pp. 355–368, 1987.

[8] L Tao and VK Asari, "Adaptive and integrated neighbourhood dependent approach for nonlinear enhancement of colour images," *Journal of Electronic Imaging*, vol. 14, no. 4, 2005.

[9] Franklin Crow, "Summed-area tables for texture mapping," in *SIGGRAPH '84*, 1984.

[10] Asheer Bachoo, "Real-time exposure fusion on a mobile computer," in *The Twentieth Annual Symposium of the Pattern Recognition Association of South Africa (PRASA)*, 2009.

[11] RC Gonzalez and RE Woods, *Digital image processing*, Addison-Wesley Publishing Company, 2002.