Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems (PDCS 2011)
December 14 - 16, 2011 Dallas, USA

COLOR IMAGE EDGE DETECTION BASED ON QUANTITY OF COLOR INFORMATION AND ITS IMPLEMENTATION ON THE GPU *

Jingxiu Zhao
School of Computer Science
Qufu Normal University
Rizhao 276826, China
email: jingxiuzhao@126.com

Yonghong Xiang, Laurence Dawson and Iain Stewart
School of Engineering and Computing Sciences
Durham University
Durham, United Kingdom, DH1 3LE
email: {yonghong.xiang, l.j.dawson, i.a.stewart}@dur.ac.uk

ABSTRACT
In this paper, we present a new method for quantifying color information so as to detect edges in color images. Our method uses the volume of a pixel in the HSI color space, allied with noise reduction, thresholding and edge thinning. We implement our algorithm using NVIDIA Compute Unified Device Architecture (CUDA) for direct execution on Graphics Processing Units (GPUs). Our experimental results show that: compared to traditional edge detection methods, our method can improve the accuracy of edge detection and withstand greater levels of noise in images; and our GPU implementation achieves speedups over related CUDA implementations.

KEY WORDS
color image edge detection; HSI color space; GPU

1 Introduction

Edge detection is one of the most fundamental operations in computer vision, image processing and pattern recognition systems. As other tasks (such as contour detection [1], image segmentation, object recognition and classification, image registration, and so on) can depend upon edge characterization, it is crucial that the process of edge detection should result in a precise characterization of the image features but via a reduced image of relatively small size. Hence, edge detection must be both reliable and efficient [2].

Edge detection differs according to whether an image is color or not. Novak and Shafer [3] found that about 90% of edges in color images are also edges in terms of their gray values. However, the remaining 10% of edges in color images can not be so characterized. Thus, many color edge detection algorithms have been proposed (see [4, 5, 6] for examples).

We propose a new color edge detection algorithm based on the *quantity of color information* (*QCI*) in the HSI color space. To detect as many edges as possible, we incorporate the QCI into an existing method which concentrates on the luminance as the color information. By doing this, we can detect edges that can not be detected by using the luminance alone. As the Canny edge detection algorithm is considered as standard and fundamental [7, 8, 9], we mainly compare our algorithm with the Canny algorithm. Experimental results show that our algorithm performs better than the Canny and Prewitt edge detection algorithms in that it can sustain significantly more noise and sometimes detect more edges. Moreover, our algorithm runs much faster than the Canny algorithm. By using NVIDIA Compute Unified Device Architecture (CUDA) [10] and optimizing memory usage, we implement our algorithm on a Graphics Processing Unit (GPU) and we obtain an implementation that is around 20 times faster than the CPU Intel Performance Primitives (IPP) Canny algorithm and around 5 times faster than the existing CUDA Canny algorithm [11]. Throughout the paper, we direct the readers to the webpage [10] for Nvidia CUDA related content.

This paper is organized as follows. In the next section we give a general introduction to edge detection techniques, GPU and CUDA, and some related work before describing the main steps of our algorithm in Section 3. We describe our parallel implementation in Section 4, and in Section 5 we detail our results and provide an analysis. The final section concludes the paper.

2 Edge Detection and GPU Applications

2.1 Introduction to Edge Detection

An edge in a monochrome image is defined as an intensity discontinuity, while in a color image, the additional variation in color must be considered; for example, the correlation among the color channels. There are a number of color edge detection methods. They can be divided into two groups: 1. techniques extended from monochrome edge detection; 2. vector space approaches, including vector gradient operators, directional operators, compound edge detectors, entropy operators, second derivative operators, vector order statistic operators and difference vector operators [12, 13]. Numerous kernels have been proposed for finding edges; for example, the Roberts kernels, Kirsch compass kernels, Prewitt kernels and Sobel kernels. By us-

*CORRESPONDING AUTHOR: YONGHONG XIANG. THIS WORK IS SUPPORTED BY THE UK EPSRC GRANT (NO. EP/G010587/1) AND THE NATURAL SCIENCE FOUNDATION OF SHANDONG PROVINCE UNDER GRANT NO. ZR2009GM009 AND NO. ZR2010FQ004.

DOI: 10.2316/P.2011.757-077

ing gradient operators the edges are detected by looking for the maximum in the first derivative of the color or intensity of the images. Second derivative operators search for zero-crossings in the second derivative of the color or intensity of the image to find edges [14]. First derivative operators are very sensitive to noise, while the second derivative operators lead to better performance in a noisy environment.

After selecting a suitable color space [15], primary edge detection steps include: (1) suppressing noise by image smoothing; and (2) localizing edges by determining which local maxima correspond to edges and which to noise (thresholding). A Gaussian filter is widely used to remove noise. The Gaussian operator is isotropic and therefore smoothes the image in all directions [16]. One problem with derivative-based edge detection is that the output may be thick, and a further edge thinning [17] step is necessary.

2.2 GPU and CUDA

NVIDIA CUDA [10] is a general purpose parallel programming model for developing applications specifically for execution on NVIDIA GPUs. CUDA provides several memory types and allows developers to write code in C/C++ completely bypassing traditional graphics interfaces such as the shader language Cg. This model of programming allows developers to exploit data parallelism in a wide range of domains whilst abstracting the underlying architecture for ease of use and portability.

The typical architecture of a CUDA GPU consists of a scalable array of streaming multiprocessors (SM), each containing a subset of Streaming Processors (SP). Methods to be executed in parallel on the GPU are known as kernels. When a kernel method is called, the execution is distributed over a set of blocks each with their own subset of parallel threads.

CUDA applications are able to support fine-grained parallelism. In contrast CPU applications often apply coarse-grained parallelism. However, through Streaming SIMD Extensions (SSE), modern Intel CPUs can support an SIMD instruction set to potentially improve performance in data-parallel regions such as image processing, digest, hashing and encoding. The two levels of granularity are directly interoperable which can lead to a wider variety of parallel solutions.

A physical restriction of the GPU is the amount of memory available on-chip. As a result the programming model presents a set of different memory types to the developer. Each thread within a block has access to a set of fast local registers and on-chip shared memory. Registers are the fastest form of storage; however, the total number per block is limited and exceeding this can result in a lower occupancy of the GPU. For inter-block communication the slower shared memory can be used. This shared memory is unique to each block and inaccessible to all others. Moving off-chip, threads have access to texture memory, constant memory and global memory (DRAM). Constant and texture memory generally have the fastest access times due to on-chip caching. However with the increased cache size presented by Fermi (a recent GPU computing architecture) [10] this is not always the case. As noted by NVIDIA in the Fermi tuning guide, when using older cards, texture memory can potentially reduce the load time when loads are not coalesced. However, newer Fermi cards cache global memory loads in the high bandwidth L1 cache which has a faster access time than texture cache. With this in mind we chose to use global and shared memory to take advantage of the higher bandwidth cache.

2.3 Related Work

Luo and Duraiswami implemented a parallel version of the Canny edge detection algorithm using CUDA [11]. Their work is similar to a previous parallel implementation; however, it uses CUDA as opposed to NVIDIA Cg and extends the functionality to include hysteresis thresholding. Sections of the implementation such as Gaussian blurring were built upon NVIDIA SDK examples which provide a good level of code reusability. The results of [11] show a modest speedup compared to highly optimised Intel SSE code and significant speedups compared to naive Matlab code. However it is worth noting that their code was tested using first generation CUDA hardware. Luo and Duraiswami note that hysteresis thresholding occupies over 75% of the overall runtime. This is due to the function to connect edges between thread blocks being called multiple times and without this stage the algorithm performs around 4 times faster. They note that the runtime can vary depending on the complexity of the image and number of edges.

Building upon the work of Luo and Duraiswami, Ogawa et al. extend the implementation of the Canny edge detection algorithm [18]. They note that as the hysteresis step is called a fixed number of times, some edges which span over blocks may not be fully traversed. Their implementation solves this by introducing a stack onto which weak edges are pushed and when all edges have been traversed, the algorithm will terminate. Their results show a speedup of around 50 times for large images; however it is unclear as to whether they first compare with niave CPU code or SSE optimised code. It would be reasonable to assume that the runtime of their improved method will also vary largely depending on the input image. The implementations of Luo and Duraiswami and Ogawa et al. only support gray input and to the best of our knowledge there have been no color edge detection algorithms implemented using CUDA.

NPP (NVIDIA Performance Primitives) is a closed-source CUDA library targeted specifically at video and image processing (although the scope is set to increase with time). NPP allows developers to easily port existing sections of Intel Performance Primitives (IPP) C/C++ code to corresponding GPU functions. We will not be using the NPP library, as it is not an open source library and modifications cannot be made.

3 Main Steps of Our Algorithm

Our algorithm is divided into 4 steps.

Step 1. Noise removal and color space transformation. The following Gaussian filter is used to remove noise.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution.

We choose to use the HSI (Hue-Saturation-Intensity) color space as it represents colors similarly to how the human eye senses colors, and it is one of the most commonly used color spaces in image processing. The HSI color space can be plotted as a cylinder (see Fig. 1) [19]. As shown in Fig. 1, for a given color P in the HSI color space, H (hue) is given by an angular value ranging from 0 to 360 degrees, S (saturation) is given by a radial distance from the cylinder center line, and I (intensity) is the height along the cylinder axis. The following three formulas convert a color in RGB (R,G,B) to a color (H,S,I) in the HSI color space:

$$\begin{aligned} I &= \frac{R+G+B}{3} \\ S &= 1 - \frac{\min(R,G,B)}{I} \\ H &= \begin{cases} \theta & \text{if } G \geq B \\ 2\pi - \theta & \text{otherwise} \end{cases} \end{aligned}$$

where $\theta = arccos\left[\frac{\frac{1}{2}[(R-G)+(R-B)]}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right]$.

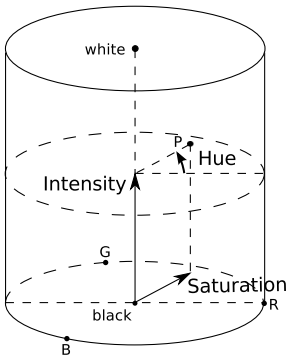

Figure 1. HSI color space.

Step 2. Gradient finding. This is an important step for our algorithm, and we thus obtain better edge maps than some of the existing algorithms.

Let $P = Img(i,j)$ be a pixel at position (i,j) in image Img, and its color information be (H_P, S_P, I_P). Define its volume $V_P = \pi * S_P^2 * I_P * (H_P/360)$. As well as volume information, color intensity information is also considered in our algorithm. Replace each pixel's color information in Img by the corresponding volume (resp. intensity) information, and denote it as Img_V (resp. Img_I). We use Prewitt kernels to calculate the magnitude of the volume information and the intensity information.

Let Img_P be the 3×3 area centered at pixel P in Img_V (resp. Img_I), and let $G_x = D_x \times Img_P$ and $G_y = D_y \times Img_P$ be the magnitude in the x direction and the y direction for P in Img_V (resp. in Img_I), where D_x is the x directional Prewitt kernel, and D_y is the y directional one. The magnitude of the volume information (resp. intensity information) for the pixel P is defined as: $M1_P = \sqrt{G_x^2 + G_y^2}$.

Another two direction operators are applied in our algorithm to obtain more accurate edges: one in the $45°$ direction; and one in the $135°$ direction. Similarly, the magnitude of the volume (resp. intensity) information is defined as: $M2_P = \sqrt{G_{45°}^2 + G_{135°}^2}$. A pixel is considered as on an edge if either $M1_P$ or $M2_P$ is above a corresponding threshold.

Step 3. Thresholding. The threshold value is set as the average volume magnitude value multiplied by a constant. For the constant, we first set a random value, and then adjust it until we get a satisfiable edge map. Different images will have different constant value.

Step 4. Edge thinning. One problem with a derivative-based edge detection algorithm is that the output edges may be thicker (several pixels in width) than a connected unit-width line around the boundary of an object. It is necessary to thin the edges. In this paper, we use one of the algorithms as implemented in [20], which is a modified version of the algorithm in [21].

4 CUDA Implementation

This section aims to describe how the four steps of our algorithm (described in Section 3) are mapped to the GPU using CUDA and C++ bindings. For the input image we chose to use the file format BMP due to the ease of loading and saving images without the need for third party libraries. Image data is stored in the CUDA vector type *uchar4* which allows us to access the individual RGB components of a pixel without the need for bit shifting techniques which are commonly used in OpenGL to pack data into a single integer value. This method requires the same memory size allocation; however, it reduces the program complexity. At each stage of the algorithm global memory accessed has been coalesced for maximum bandwidth. To meet the block and thread size restrictions of CUDA we chose to use square images. However, if needed the solution could be modified to pad images. Finally our solution is designed to utilise the advancements made by Compute 2.0 devices such as an increased shared memory size and processing of threads by full warp; as a result it is incompatible with older GPUs.

Step 1. Gaussian Blurring and Color Space Transformation. The first step of our algorithm applies a Gaussian blur to the input image whilst preserving the three color channels. We initially set the standard deviation $\sigma = 1$

for the Gaussian filter, and the user can change it at run time. As this step is a fairly common operation in image processing NVIDIA have provided optimised CUDA code as part of the source development kit (SDK). The example provided is designed to reduce idle threads, allows multiple pixel processing per thread and is optimised for memory coalescence. NVIDIA describe how the Gaussian blur technique can be expressed as the product of two one-dimensional filters. This allows developers to replace a single image convolution kernel with two kernels targeting separate planes for reduced complexity. As threads at the edges of the block will require additional information from neighbouring blocks, a block apron is used which overlaps pixel values. As we convolve the image with a small filter radius we chose to implement two separable filters as opposed to using the fast Fourier transform library. This approach was also used by Luo and Duraiswami for the first stage of their parallel Canny edge detection algorithm [11].

The separable convolution example provided only supports blurring of gray images and is optimised for a single channel. For a color image the three channels can be blurred independently and recombined after the last blur. This basic approach would allow us to use the source provided without modification. However by combining the three iterations into one single pass we can take advantage of the results existing in shared memory.

At the end of the second convolution we calculate the color space transformation. By combining the two stages of the algorithm we are able to remove unnecessary access to slow global memory. After the color space transformation is complete the results are written back to global memory. This ensures that all blocks are synchronised for the next stage of the algorithm. Another optimisation made to the first two stages of the algorithm was implementing a set of pre-processor macros to replace repeating independent sections of calculation. This allowed us to carefully control and reduce the total number of registers used by each thread block and retain a higher occupancy. The final optimisation tested was to replace the inverse cosine function with a faster approximation method. However this performed worse than the compiler optimised fast-math alternative and was not included.

Step 2. Gradient Calculation. For the next step of the algorithm we calculate the gradient through convolution with the Prewitt operator. As we apply the operator in 4 directions, two of which are diagonal, the process is not linearly separable and requires a different approach to achieve efficient convolution. Stam [22] describes how peak convolution performance can be attained by using shared memory efficiently and gives insight into how his method could be modified to support color images across three channels (this approach was also used by the Gaussian blur kernels). Stam's solution also maps multiple pixels per thread to achieve higher occupancy and reduce the total number of apron pixels loaded. However, as his example uses gray images, the shared and global memory access patterns are

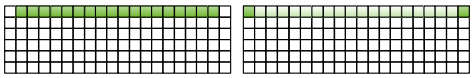

Figure 2. Part 1, load the top apron pixels

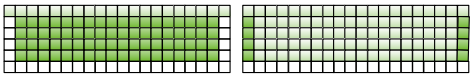

Figure 3. Part 2, load the centre and the side apron pixels

Figure 4. Part 3, load the bottom apron pixels

carefully aligned for smaller data types. Based on a modified version of the proposed solution we perform a fast efficient 2D convolution of the kernel that handles larger float datatypes and multiple color channels. Figs 2, 3 and 4 show the shared memory loading scheme. Shared memory is aligned across 32 memory banks satisfying the access pattern requirements to avoid bank conflicts. In tests we found our approach was around 2-3 times faster than naive global lookups and on older compute 1.0 hardware the penalty was much greater due to the lack of L1 global memory caching. Once the convolution is complete the values are again saved back to global memory in order to synchronise the algorithm.

Step 3. Average Gradient Calculation and Thresholding. In order to perform global edge thresholding the average gradient values must first be calculated. Using the results obtained from the previous convolution we implement a parallel reduction algorithm based upon the SDK example provided by NVIDIA. The algorithm is split into two stages (as CUDA does not support global synchronisation): first calculate the sum within each block; then calculate the global total of the blocks. Before saving the total average value to global memory the value is multiplied by a constant to obtain the thresholding value (the constant has been obtained by experiment).

With the global threshold values calculated, the next stage is the relatively simple task of applying this to each pixel. As each pixel is independent there is no need for any apron area surrounding the blocks and we assign each thread exactly one pixel. The first thread of each block then caches the previously computed threshold value into shared memory using a single 128-bit load. Each thread then loads the corresponding pixel data into local registers and performs a conflict-free broadcast access to the cached threshold. Each thread then decides if the pixel qualifies as an edge and writes this data back to global memory. At this point there is an option to either display the edges as they are or continue to process the data and thin the detected

256	32	4
128	16	2
64	8	1

Figure 5. LUT index matrix

edges.

Step 4. Edge Thinning. Using the edge thinning algorithm mentioned in Section 3, we are able to thin an edge based upon 3 conditions. Fortunately we can avoid the costly task of calculating if each pixel satisfies all of the conditions by pre-calculating the outcome of any possible situation and storing this data in a lookup table (LUT) in fast constant memory. As there are only 9 pixels in each window of interest, each of which can be 0 or 1, there are only 512 different outcomes to the edge thinning algorithm per pixel. The outcome of any pixel's calculation can be accessed in the LUT by calculating the index shown according to the matrix shown in Fig. 5. This is a commonly used optimisation in image processing and is possible in this circumstance due to the small kernel radius. A further optimisation noted by Matlab was to bypass the LUT for any pixel with no value as this will always return 0 [23]. On the GPU side we employ the same 2D shared memory loading scheme as was used in step 2. After the results from thresholding are loaded into shared memory, each thread calculates the index for the LUT for each pixel. The final step copies the value obtained from the LUT back into global memory resulting in a thinner edge. This process is repeated up to four times depending on desired output.

5 Results

In this section, we describe the results of our algorithm, timings from GPU experiments and compare our algorithm with other algorithms. For each algorithm the results shown are based on the criteria of selecting the best output by tuning the corresponding parameters. To evaluate our proposed technique, we compare our algorithm with the Canny and Prewitt edge detection algorithms for a set of color images of the standard image processing dimension 512×512.

5.1 Noise Tolerance

To demonstrate our algorithms tolerance to noise we present the results of experiments using the image *Lemon.bmp*. In Fig. 6, the images in column one are: original lemon, +20% white noise lemon, and +40% white noise. Columns two to four contain the results of our algorithm, Canny and Prewitt, respectively. With no noise present we can see that our algorithm detects more useful edges than Prewitt and produces an edge map similar to Canny. When noise is introduced to the image, our algorithm maintains a near constant edge map whilst Canny and Prewitt begin to falter. When noise is added, Canny and Prewitt required significant input parameter changes whilst our algorithm required minimal input changes. We can observe that the hysteresis stage of the Canny edge detection algorithm begins to form incomplete edges due to an increased threshold to remove noise.

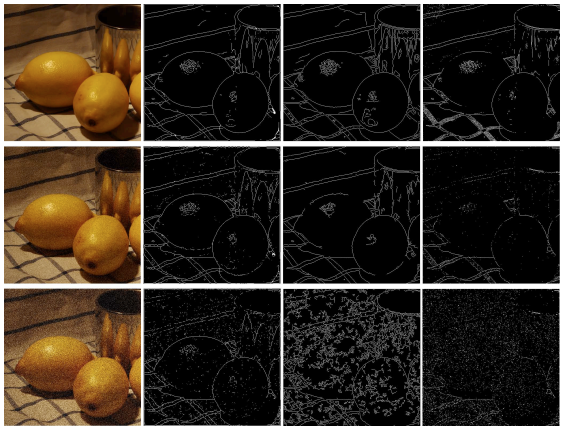

Figure 6. Compare edge maps for image lemon.

5.2 Improved Edge Detection

By using coefficients for intensity and volume, in some circumstances we are able to produce better edge maps. For example when using the image *Silicon.bmp* the results shown in Fig. 7 are significantly better than Canny and Prewitt. In this example we adjusted the parameters to allow more volume information than intensity to contribute to the edges. When using the image *Butterfly.bmp* we were able to produce high quality edge maps with less background edges than Canny and Prewitt (Fig. 8) (we focussed on the subject by using volume information, intensity information or both). Experimentally we found that our algorithm was able to match the edge map produced by Canny for most test images; for example, *Lemon* and the standard *Mandrill*. In some circumstances such as the previously mentioned Silicon and Butterfly we were able to improve upon the results. However in some instances such as the standard test image Lena, we found that it is not always the case that the additional volume information contributes to an improved edge map especially in regions of gradual intensity change such as shadows. However, we still detected high quality edges, and improve upon the edge maps of Prewitt.

5.3 GPU Implementation Results

This section presents results from our GPU experiments using an NVIDIA GTX 580 GPU and Intel i7 950 CPU. Using the five different test images discussed we recorded the average runtime over 1000 iterations for our algorithm, the aforementioned GPU CUDA Canny implementation and the optimised OpenCV CPU Canny implementation. As

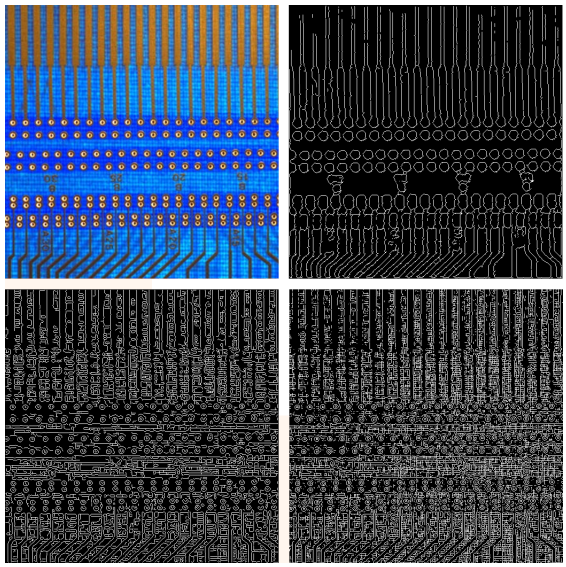

Figure 7. Edge maps for Silicon.bmp (top-left); ours (top-right), Canny's (bottom-left) and Prewitt's (bottom-right)

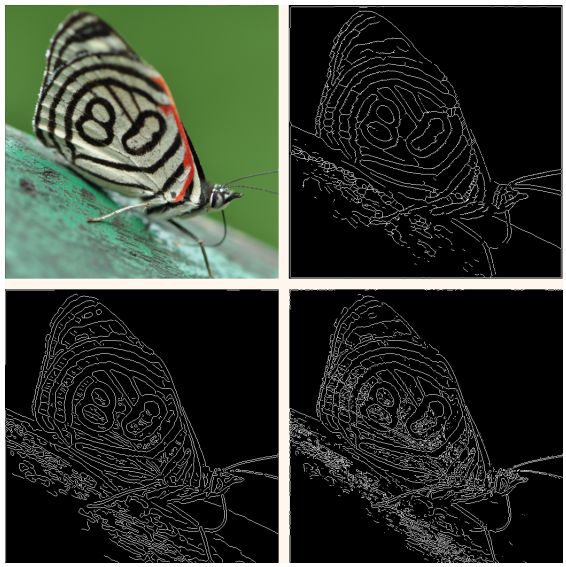

Figure 8. Edge maps for Butterfly.bmp (top-left); ours (top-right), Canny's (bottom-left) and Prewitt's (bottom-right)

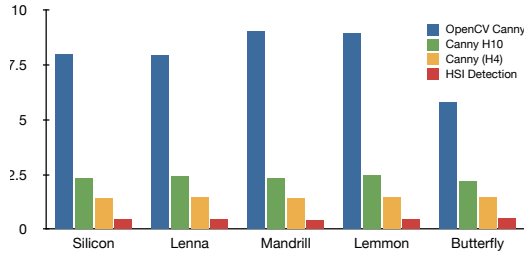

Figure 9. A comparison of the average time taken by each edge detection implementation using a variety of color input images on the GPU and CPU

Input	OpenCV	GPU Canny H10/H4	HSI Detection
Silicon	8.01	2.33 / 1.42	0.48
Lenna	7.95	2.41 / 1.46	0.46
Mandrill	9.05	2.34 / 1.43	0.44
Lemmon	8.95	2.48 / 1.49	0.48
Butterfly	5.8	2.22 / 1.45	0.52

Table 1. The average time taken in milliseconds by each edge detection implementation on the GPU and CPU

the results from the Canny CUDA implementation were obtained used older Compute 1.0 hardware, we tested and re-calculated the results using our updated setup to provide a more accurate and fair comparison. The same applies for the OpenCV implementation. Running the hysteresis step four times as proposed by Luo and Duraiswami for their CUDA Canny implementation was rarely sufficient and we have also included timings from executing the step 10 times, producing a more accurate Canny output edge map. In Fig. 9, Table 1, and Table 2, H4 (resp. H10) means executing the hysteresis step 4 (resp.10) times. Fig. 9 and Table 1 show the times taken by the three implementations for the five mentioned images. Table 2 shows the speedup attained by our algorithm.

6 Conclusion

We proposed a new edge detection method based on QCI in HSI color space, which establishes a nonlinear relationship between a pixel's color and its volume. By using the QCI,

Input	OpenCV Speedup	GPU Canny H10/H4 Speedup
Silicon	16.69x	4.85x / 2.96x
Lenna	17.28x	5.24x / 3.17x
Mandrill	20.57x	5.32x / 3.25x
Lemmon	18.65x	5.17x / 3.10x
Butterfly	11.15x	4.27x / 2.79x

Table 2. The speedup of our algorithm obtained over the OpenCV Canny implementation and GPU Canny Implementation

we are able to find edges that are not detectable when only intensity information is used. Experimental results show that our proposed algorithm performs better than Prewitt's algorithm. Moreover, in most of the cases, our algorithm has a higher signal-to-noise ratio and is more robust than Canny's algorithm. The limitation of our proposed algorithm is that the constant value for thresholding has to be obtained manually, and which limits our ability in testing large number of images for the proposed algorithm. We implemented our algorithm in CUDA so as to utilize the computing ability of GPUs. By optimizing the memory usage, for an image of size 512×512, the algorithm's total execution time is about $0.5ms$ (see Table 1), which means we can detect edges for about $2,000$ images of size 512×512 within one second. As shown in Table 2, our GPU implementation is up to 20 times faster than the IPP Canny algorithm, and is about 5 times faster than the existing CUDA Canny algorithm. Due to the speed of our algorithm we would be able to process a video feed or array of images in real time.

Our future research will focus on obtaining the constant value for thresholding automatically for our proposed algorithm and applying it in video tracking on the GPU.

References

[1] G. Papari and N. Petkov, "Edge and line oriented contour detection: State of the art," *Image and Vision Computing*, vol. 29, no. 2-3, pp. 79–103, 2011.

[2] D. Ziou and S. Tabbone, "Edge detection techniques - an overview," *International Journal of Pattern Recognition and Image Analysis*, vol. 8, pp. 537–559, 1998.

[3] C. Novak and S. A. Shafer, "Color edge detection," in *Proceedings of DARPA Image Understanding Workshop*, 1987, pp. 35–37.

[4] B. Bouda, L. Masmoudi, and D. Aboutajdine, "Cvvefm: Cubical voxels and virtual electric field model for edge detection in color images," *Signal Processing*, vol. 88, no. 4, pp. 905–915, 2008.

[5] C. Lopez-Molina, H. Bustince, J. Fernandez, P. Couto, and B. D. Baets, "A gravitational approach to edge detection based on triangular norms," *Pattern Recognition*, vol. 43, no. 11, pp. 3730 – 3741, 2010.

[6] P. Melin, O. Mendoza, and O. Castillo, "An improved method for edge detection based on interval type-2 fuzzy logic," *Expert Systems with Applications*, vol. 37, no. 12, pp. 8527 – 8535, 2010.

[7] L. G. Shapiro and G. C. Stockman, *Computer Vision*. Prentice Hall, 2001.

[8] A. Koschan and M. Abidi, "Detection and classification of edges in color images," *Signal Processing Magazine, IEEE*, vol. 22, no. 1, pp. 64 – 73, jan. 2005.

[9] O. M.A. and H. H., "Literature review on edge detection," http://www.essex.ac.uk /csee/research/publications/technicalreports/2010/CES-506.pdf, 2010, [Online; accessed 16-Sep.-2011].

[10] NVIDIA, "Nvidia cuda home," http://www.nvidia.com/object/cuda_home_new.html/, 2011, [Online; accessed 16-Sep.-2011].

[11] R. Duraiswami and R. Duraiswami, "Canny edge detection on nvidia cuda," *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–8, 2008.

[12] A. Evans, *Advances in Nonlinear Signal and Image Processing*, ser. EURASIP Book Series on Signal Processing and Communications. Hindawi Publishing Corporation, 2006, ch. 12, pp. 329–356.

[13] S. Zhu, K. N. Plataniotis, and A. N. Venetsanopoulos, "Comprehensive analysis of edge detection in color image processing," *Optical Engineering*, vol. 38, pp. 612–625, Apr. 1999.

[14] D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of the Royal Society of London Series B*, vol. 207, pp. 187–217, 1980.

[15] M. Celebi, H. Kingravi, and F. Celiker, "Fast colour space transformations using minimax approximations," *Image Processing, IET*, vol. 4, no. 2, pp. 70 – 80, April 2010.

[16] F. Samopa and A. Asano, "Hybrid image thresholding method using edge detection," *International Journal of Computer Science and Network Security*, vol. 9, no. 4, pp. 292 – 299, 2009.

[17] L. Lam, S. Lee, and C. Suen, "Thinning methodologies-a comprehensive survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, pp. 869–885, 1992.

[18] K. Ogawa, Y. Ito, and K. Nakano, "Efficient canny edge detection using a gpu," in *2010 First International Conference on Networking and Computing*. IEEE, 2010, pp. 279–280.

[19] F. Perez and C. Koch, "Toward color image segmentation in analog vlsi: algorithm and hardware," *International Journal of Computer Vision*, vol. 12, pp. 17–42, February 1994.

[20] Z. Guo and R. W. Hall, "Parallel thinning with two-subiteration algorithms," *Commun. ACM*, vol. 32, pp. 359–373, March 1989. [Online]. Available: http://doi.acm.org/10.1145/62065.62074

[21] T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Commun. ACM*, vol. 27, pp. 236–239, March 1984. [Online]. Available: http://doi.acm.org/10.1145/357994.358023

[22] J. Stam, "Convolution soup," http://www.nvidia.com/content/GTC/documents /1412_GTC09.pdf, 2009, [Online; accessed 16-Sep.-2011].

[23] S. Eddins, "Performance optimization for applylut," http://blogs.mathworks.com/steve/2008/06/13/performance-optimization-for-applylut/, 2008, [Online; accessed 16-Sep.-2011].