ACCELERATION OF A LOCALLY TUNED SINE NON LINEAR VIDEO
ENHANCEMENT ALGORITHM ON GPGPU

Thesis
Submitted to
The School of Engineering of the
UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for
The Degree of
Master of Science in Electrical Engineering

By
Julian Daniel John
Dayton Ohio
December, 2011
ACCELERATION OF A LOCALLY TUNED SINE NON LINEAR VIDEO ENHANCEMENT ALGORITHM ON GPGPU

Name: John, Julian Daniel

APPROVED BY:

Tarek Taha, Ph.D.
Advisory Committee Chairman
Associate Professor
Electrical and Computer Engineering

Eric Balster, Ph.D.
Committee Member
Assistant Professor
Electrical and Computer Engineering

Vijayan Asari, Ph.D.
Committee Member
Ohio Research Scholar and Chair
Electrical and Computer Engineering

John G Weber, Ph.D.
Associate Dean
School of Engineering

Tony E Saliba, Ph.D.
Dean, School of Engineering
& Wilke Distinguished Professor
ABSTRACT

ACCELERATION OF A LOCALLY TUNED SINE NON LINEAR VIDEO ENHANCEMENT ALGORITHM ON GPGPU

Name: John, Julian Daniel
University of Dayton
Advisor: Dr. Tarek Taha

Computer Vision based applications support various domains such as medical, manufacturing, military intelligence and surveillance systems. These applications can be divided into: image acquisition, pre-processing, feature extraction, detection or segmentation, and high-level processing. However these tasks are time intensive due to the compute bound nature of the algorithm.

In this thesis, an algorithm, based on an image dependent nonlinear function, the Locally Tuned Sine Nonlinearity (LTSN), is accelerated using NVIDIA’s Computer Unified Device Architecture (CUDA) and the CPU. The main core of the algorithm is a nonlinear sine transfer function which is very flexible in enhancing the dark regions and compressing overexposed regions of an image. The video enhancement algorithm gave 21 frames per second compared to 9 frames per second for a 480p video. It is envisaged that the new technique would be useful for improving the visibility of scenes of night time driving and night security situations in real time.
Dedicated to my wife and parents
ACKNOWLEDGEMENTS

This research would not have been possible without the support, advice and encouragement of my thesis committee. I would like to express my gratitude to my advisor Dr. Tarek Taha, and the other members of the committee Dr. Vijayan Asari and Dr. Eric Balster who were the driving force behind this research.

For the pioneering work in Computer Vision and Image processing I am grateful to the members of the Vision Lab who helped me tackle issues related to the processing of images. I wish to thank the High Performance Lab in helping me tackle performance and optimization issues I had in CUDA.

Lastly I would like to thank Marilyn Knisley of the Electrical and Computer Engineering department and Linda Wallace of Graduate School in helping me with the technicalities of the thesis process.
TABLE OF CONTENTS

ABSTRACT ......................................................................................................................... iv

ACKNOWLEDGEMENTS .................................................................................................. vi

TABLE OF CONTENTS .................................................................................................... vii

LIST OF FIGURES ........................................................................................................... ix

LIST OF TABLES ............................................................................................................... xi

1. INTRODUCTION ........................................................................................................ 1
   1.1 Motivation .................................................................................................................. 1
   1.2 Algorithm .................................................................................................................. 3
   1.3 Platform and Acceleration Results ........................................................................ 4
   1.4 Thesis Outline .......................................................................................................... 4

2. RELATED WORK AND CONTRIBUTIONS .................................................................. 6
   2.1 Adaptive and Integrated Neighborhood-Dependent Approach for Nonlinear
       Enhancement (AINDANE) Algorithm .......................................................................... 6
   2.2 An Illuminance-Reflectance Model for Nonlinear Enhancement (IRME) Algorithm .... 9
   2.3 Multiple Windowed Inverse Sigmoid Function (MWIS) ........................................... 13
   2.4 Summary .................................................................................................................. 16

3. LOCALLY TUNED SINE NON LINEAR ENHANCEMENT ALGORITHM .................. 18
   3.1 Adaptive Intensity Enhancement .......................................................................... 18
       3.1.1 Intensity Computation ....................................................................................... 18
       3.1.2 Enhancement of Dark and Compression of Bright Pixels .................................. 19
       3.1.3 Calculation of Mean Image ............................................................................. 22
       3.1.4 High Frequency Boosting ............................................................................... 23
   3.2 Contrast Enhancement ............................................................................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>Acquiring the Intensity Information of Surrounding Pixels</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Intensity Transformation Process</td>
<td>27</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Contrast Enhancement with Multi-level Convolution Results</td>
<td>27</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Auto Levels</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Color Restoration</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>GPGPU PLATFORM</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>GPU Hardware Architecture</td>
<td>32</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Third Generation Streaming Multiprocessor</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Fermi Dual Scheduler</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>GPU Software Model</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Hardware Execution</td>
<td>37</td>
</tr>
<tr>
<td>5.</td>
<td>CUDA IMPLEMENTATION</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Image Processing with CUDA</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>GPU Kernel Processes</td>
<td>41</td>
</tr>
<tr>
<td>6.</td>
<td>EXPERIMENTAL SETUP AND RESULTS</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Profiling</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Sine Non Linear Function Performance</td>
<td>48</td>
</tr>
<tr>
<td>6.3</td>
<td>Gaussian Smoothening and Averaging Kernel Performance</td>
<td>50</td>
</tr>
<tr>
<td>6.4</td>
<td>Color Restoration Kernel Performance</td>
<td>52</td>
</tr>
<tr>
<td>6.5</td>
<td>Overall Algorithm Analysis</td>
<td>55</td>
</tr>
<tr>
<td>7.</td>
<td>CONCLUSION AND FUTURE WORK</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>64</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Flow chart of the Sine Nonlinear Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Various curves of Nonlinear Transfer function</td>
<td>21</td>
</tr>
<tr>
<td>4.1 CPU v/s GPU comparison of floating point operations per second</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Transistor allocations for CPU and GPU</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Fermi architecture</td>
<td>33</td>
</tr>
<tr>
<td>4.4 Fermi Streaming Multiprocessor</td>
<td>34</td>
</tr>
<tr>
<td>4.5 The Dual Warp Scheduler</td>
<td>35</td>
</tr>
<tr>
<td>4.6 CUDA Hierarchy of threads, blocks, and grids</td>
<td>36</td>
</tr>
<tr>
<td>5.1 2D hierarchy of blocks and threads for a 48 x 32 pixel image</td>
<td>40</td>
</tr>
<tr>
<td>5.2 GPU kernels flow diagram</td>
<td>43</td>
</tr>
<tr>
<td>6.1 Complexity profiling of the building blocks of the proposed LTSN algorithm</td>
<td>46</td>
</tr>
<tr>
<td>6.2 Performance analysis of Sine Nonlinear function</td>
<td>48</td>
</tr>
<tr>
<td>6.3 Speedup gained by the Sine Nonlinear function for various image sizes</td>
<td>50</td>
</tr>
<tr>
<td>6.4 Performance analysis of Gaussian smoothening and averaging</td>
<td>51</td>
</tr>
<tr>
<td>6.5 Speedup gained by Gaussian smoothening and averaging</td>
<td>52</td>
</tr>
<tr>
<td>6.6 Performance analysis of Color Restoration</td>
<td>53</td>
</tr>
<tr>
<td>6.7 Speedup gained by Color Restoration</td>
<td>54</td>
</tr>
<tr>
<td>6.8 Performance analysis of overall CPU+GPU enhanced LTSN algorithm</td>
<td>55</td>
</tr>
<tr>
<td>6.9 Speedup achieved by the GPU accelerated LTSN video enhancement algorithm for a frame</td>
<td>57</td>
</tr>
<tr>
<td>6.10 Original image and enhanced image</td>
<td>57</td>
</tr>
<tr>
<td>6.11 Original image and enhanced image</td>
<td>57</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 GPU memory copy time</td>
<td>47</td>
</tr>
<tr>
<td>6.2 Execution times and speed up of Sine Nonlinear function</td>
<td>49</td>
</tr>
<tr>
<td>6.3 Execution times and speed up of Gaussian smoothening algorithm</td>
<td>52</td>
</tr>
<tr>
<td>6.4 Color Restoration GPU kernel timing for 640X480 pixel image</td>
<td>53</td>
</tr>
<tr>
<td>6.5 Execution times and speed up of Color Restoration function</td>
<td>54</td>
</tr>
<tr>
<td>6.6 Execution times and speed up of LTSN algorithm</td>
<td>56</td>
</tr>
<tr>
<td>6.7 Execution times and speedup of LTSN video algorithm</td>
<td>61</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

This paper emphasizes on the acceleration and efficient implementation of a locally tuned sine nonlinear video enhancement algorithm (LTSN) [1] on Graphics Processing Units (GPUs). These algorithms are crucial components of computer vision based enhancement algorithms that deal with Signal to Noise ratio (SNR), edge sharpness and color accuracy. A real world scene would always have extremely high illumination that underexposes the dark regions and makes the low contrast regions less visible to the human eye. Computer vision experts have strived to enhance such real world scenes to extract information from them in applications such as visual surveillance, organizing information, modeling objects or environments, navigation of robotic systems. The vision of image enhancement and computer vision algorithms is to extract images from the real world and interpret them in real time. This thesis deals with realizing that dream to make the computer vision based algorithms suited for real time operations.

1.1 Motivation

CPU processor advances such as the Intel’s MMX/SSE instructions were introduced into their designs to improve the performance significantly throughout multimedia applications. But CPU is usually heavily loaded, due to which it cannot
meet the processing power requirements. Video enhancement algorithms such as the locally tuned sine nonlinear algorithms (LTSN) cannot be processed for real time applications even with high computing power in the present generation CPUs. With the development of silicon technologies, inexpensive yet powerful graphical processing units (GPUs) are used today for higher processing speeds. The computationally intensive part of the algorithm is offloaded to the GPUs.

The GPU is a massively parallel device which is capable of processing large amounts of data to perform various compute and memory bound operations with great speed and efficiency without affecting the accuracy of the data. The NVIDIA Fermi architecture chipset contains 3.0 billion transistors which is more than the Intel Pentium processor. The internal pipelined processing fashion makes it more suitable for streams processing. The GPU speed grows faster than the famous Moore’s law for CPUs, which is 2.4 times a year compared to 2 times in 18 months.

Unlike previous generations of GPUs that partitioned computing resources into vertex and pixel shaders, the Computer Unified Device Architecture (CUDA) included a uniform shader pipeline, allowing every arithmetic logic unit (ALU) on the chip to assemble by a program intending to perform general-purpose computations. NVIDIA intended the CUDA architecture to be used for general purpose computing, these ALUs were built to comply with the IEEE requirements for single precision floating point arithmetic and were tailored to use an instruction set tailored for general purpose computation rather than specifically for graphics. The CUDA architecture was added in order to create a GPU that would excel at computation in addition to performing well at traditional graphics tasks.
This feature of NVIDIA GPUs triggered a lot of interest in the area of High Performance Computing (HPC) due to its powerfulness, SIMD operation and programmability. The success achieved for non-graphics oriented applications such as numerical computations like basic linear algebra [11] and image processing [5-10] has been a driving force for the research on the acceleration of LTSN video enhancement algorithms. In [5] the authors presented a technique to multiply large matrices quickly using graphics hardware. Strzodka and Rumpf have solved parabolic differential equations fully in graphics hardware using complicated numerical schemes [6]. High computation bound operation like FFT on GPU were reported by Doggett et al in [9].

In this thesis the issue of the rendering time taken by the enhancement algorithm for each frame was analyzed and studied. The processing time on a CPU does not suit the algorithm for real time operations. Achieving high processing speeds are essential for various real time applications in computer vision.

1.2 Algorithm

A Locally Tuned Sine Nonlinearity (LTSN) is proposed for enhancing extremely high contrast images. The algorithm is a new nonlinear image enhancement algorithm, based on an image dependent nonlinear function. The control parameters are based on image statistics and they are determined adaptively. The proposed algorithm is capable of compressing bright regions and at the same time enhancing dark regions by preserving the main structure of the illuminance - reflectance characteristics. The main core of the algorithm is a new nonlinear sine transfer
function that is very flexible in enhancing the dark regions and compressing overexposed regions in an image. A neighborhood dependent approach is employed for contrast enhancement. The Laplacian filtered image (reflectance) preserves the finer details of the enhanced image. The quality of the enhanced image is further improved by applying a contrast stretch process. A basic linear color restoration process based on the chromatic information of the original image is employed to convert the enhanced intensity image back to a color image.

1.3 Platform and Acceleration Results

The rendering is done for each pixel in a frame; the algorithm is a potential candidate for GPU acceleration as it comprises of SIMD instructions. Although SIMD instructions run well on the GPU, there are certain functions that are not too computationally bound that processes faster on the CPU. Thus a benchmarking and profiling operation was done to divide the algorithm into time slices to determine the instructions that take the majority of the processing time. The instructions that run slower on the CPU are more likely to get a high speed up when processed on the GPU. Therefore a CPU+GPU execution was the best approach to accelerate the LTSN algorithm in this research.

1.4 Thesis Outline

The rest of the thesis is organized in the following manner. Chapter 2 provides the reader with the background information necessary to understand the research area to understand the implementation explained in the later chapters. The related work and the contributions in the image enhancement field are mentioned. Chapter 3
describes the supporting work for the algorithms that enabled to come up with the LTSN algorithm. Additionally the differences between the previous and the current algorithm have been mentioned. Chapter 5 gives the CUDA implementation of the LTSN algorithm. The details of the GPU processing are mentioned along with their mapping into CUDA. Chapter 6 explains the experimental setup, the approaches taken to get the final optimized code for higher performance and the results obtained for each function. The speedup attained for the instructions processed on the GPU is found by comparing it to the processing time taken for those instructions on the CPU. An overall algorithm performance study was done for various frame sizes. Chapter 7 details possible extensions for future work and the conclusion.
CHAPTER 2

RELATED WORK AND CONTRIBUTIONS

In this section we review the inspirations behind the development of the LTSN algorithm. The algorithms that helped develop the proposed algorithm are AINDANE, IRME and MWIS methods.

2.1 Adaptive and Integrated Neighborhood-Dependent Approach for Nonlinear Enhancement (AINDANE) Algorithm

The AINDANE[11] algorithm is an adaptive version of the INDANE algorithm. Similar to INDANE, AINDANE algorithm consists of two main parts: Adaptive luminance enhancement and adaptive contrast enhancement. Adaptive luminance enhancement is a nonlinear intensity transformation which is self-tuned by the histogram statistics of the input image. During intensity transformation, the luminance of the dark pixels is increased and the dynamic range of the image is compressed at the same time. Adaptive contrast enhancement, which is adaptively controlled by the global statistics of the image, tunes the intensity of each pixel based on its relative magnitude with respect to the neighboring pixels.
In AINDANE, the color images are converted to intensity (gray-scale) images, using the specification in the NTSC (National Television System Committee) standard method.

\[ I(x,y) = \frac{76.245R + 149.6851G + 29.07B}{255} \]  

(2-1)

where R, G, and B are the values of the red, green and blue color bands of the tricolor images respectively. The image intensity is normalized as

\[ I_n(x,y) = \frac{i(x,y)}{255} \]  

(2-2)

After the image intensity is normalized between 0 and 1, the intensity images are treated by a nonlinear transfer function that enhances the dark region of the image and compresses the dynamic range. This transfer function is defined as

\[ I_n'(x,y) = \frac{I_n(x,y)^{0.75z+0.25} + (1 - I_n(x,y))0.4(1-z) + I_n(x,y)^{(2-z)}}{2} \]  

(2-3)

where z is a parameter that provides the curve of the transfer function. z is related to the image histogram and can be defined as
where $L$ is the intensity level corresponding to where cumulative distribution function (CDF) = 1. Here, $L$ is used as an indication to determine how dark the 10% of pixels in an image are. If these pixels are very dark, ($L < 50$), they will enhance the most. If they are sufficiently bright, ($L > 150$), no pixel will be enhanced. If they are dark, ($50 > L > 150$), they will be less enhanced. In Figure 2.10, the transfer functions with different $z$ values are illustrated.

After adaptive luminance enhancement, the contrast of the luminance-enhanced images is degraded, so contrast enhancement is applied. The AINDANE algorithm uses almost the same contrast enhancement method as INDANE. But, unlike the INDANE algorithm, the AINDANE algorithm uses adaptive contrast enhancement by using a parameter $P$, related to the global standard deviation, of the input intensity image $I(x, y)$. $P$ is used as a power function of the ratio of the surrounding intensity information over input image and it can be expressed as

$$E(x,y) = \left[ \frac{I_{conv}(x,y)}{I(x,y)} \right]$$

(2-5)
In Equation (2-5) \( P \) is an image dependent parameter, which is used to tune the contrast enhancement process. \( P \) is an adaptive parameter related to the global standard deviation of the input intensity image \( I(x, y) \) and can be determined as

\[
P = \begin{cases} 
3, & \sigma \leq 3 \\
\frac{27 - 2\sigma}{7}, & 3 < \sigma \leq 10 \\
1, & \sigma > 10
\end{cases}
\]  

(2-6)

where \( \sigma \) is the indication of the contrast level of the original intensity image. If the standard deviation is less than 3, it means the image has poor contrast and \( P \) will be 3, and it will further increase contrast enhancement. If the standard deviation is much more than 10, it means the image has sufficient contrast and \( P \) will be 1, and it does not increase contrast enhancement. Otherwise, there is a linear relationship between \( P \) and \( \sigma \).

2.2 An Illuminance-Reflectance Model for Nonlinear Enhancement (IRME) Algorithm

IRME[12] algorithm is an algorithm that is based on a physical description of the creation of a radiance map of the real world scene. It divides the object radiance into two parts: illumination and reflectance. We can describe illumination as the light intensity incident on an object’s surface and reflectance as the light reflection properties of the object’s surface. This separation provides a method to process images for the purpose of obtaining an improved visual perception of those scenes.

The algorithm consists of four parts: (a) illumination estimation and reflectance extraction; (b) adaptive dynamic range compression of illuminance; (c) adaptive mid-tone frequency components enhancement; (d) image restoration.
The first step of the first part of the algorithm is to obtain the intensity image, this step can be expressed as

\[ I(x,y) = \max[R(x,y), G(x,y), B(x,y)] \]

where R, G, and B are the values of the red, green and blue color bands of pixels (for 8-bit images) respectively. This step is the definition of the value (V) component in HSV color space.

The image intensity can be simply formulated as

\[ I(x,y) = L(x,y)R(x,y) \]

where \( L(x, y) \) is the illumination of the image and \( R(x, y) \) is the reflectance of the image.

In this algorithm, it is assumed that the illumination \( L(x, y) \) is contained in the low frequency components of the image and the reflectance \( R(x, y) \) is contained in the high frequency components of the image. In the real world, the dynamic range of the illumination variation can be several orders larger than the dynamic range of the reflectance. Therefore, in this algorithm, while the dynamic range of the illumination part of the image compresses, the dynamic range of the reflectance part of the image does not compress.
The last step of the first part of the algorithm is to normalize the illumination by dividing with 255 for 8-bit images. After the illumination $L$ is obtained using Equation (2-9), the reflectance $R$ is computed using Equation (2-8) to use further steps of the algorithm.

$$I_{\text{conv}}(x,y) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} I(m,n)F_i(m+x,n+y)$$

(2-9)

The second part of the algorithm is adaptive dynamic range compression of illuminance using the windowed-inverse sigmoid (WIS) function. The sigmoid function is defined as

$$f(v) = \frac{1}{1 + e^{-av}}$$

(2-10)

The inverse sigmoid function can be used to pull down the intensity of the over-lighted pixels and at the same time pull up the low intensity of dark pixels. This function is used as the intensity transfer function for dynamic range compression by performing the following steps described in Equations (2-10) – (2-13).

$$L'_n = L_n[f(v_{\text{max}}) - f(v_{\text{min}})] + f(v_{\text{min}})$$

(2-11)

$$L''_n = \frac{1}{a} \ln\left(\frac{L'_n}{L_n} - 1\right)$$

(2-12)

$$L_{n,\text{enh}} = \frac{L''_n - v_{\text{min}}}{v_{\text{max}} - v_{\text{min}}}$$

(2-13)
where Equation (2-11) linearly maps the input range [0 1] of the normalized illuminance $L_n$ to the input range \([f(v_{min}) f(v_{max})]\) for windowed-inverse sigmoid. Equation (2-13) is the inverse sigmoid function. Equation (2-13) is applied to normalize the output illuminance to range [0 1]. Parameters $v_{max}$ and $v_{min}$ are used to tune the curve shape of the transfer function.

The third part of the IRME is adaptive mid-tone frequency component enhancement. This process is very similar to the contrast enhancement part of AINDANE. The equations for this part are

\[
L'_{enh} = L_{enh}(x, y)^{E(x, y)}
\]

(2-14)

where

\[
E(x, y) = R_s(x, y)^p = \left(\frac{I_{cond}(x, y)}{I(x, y)}\right)^p
\]

(2-15)

where

\[
p = \begin{cases} 
2, & \sigma \leq 30 \\
-0.03 + 2.9, & 30 < \sigma \leq 80 \\
1/2, & \sigma \geq 80
\end{cases}
\]

(2-16)

Finally, the enhanced image is obtained by multiplying the illumination $L'_{enh}(x, y)$ and reflectance $R(x, y)$. 

\[
L'_{enh} = L_{enh}(x, y)^{E(x, y)}
\]
To convert the enhanced intensity image to RGB color image, a linear color restoration process is applied. The RGB values \((r', g', b')\) can be expressed as

\[
I'(x, y) = L'_{n,\text{enh}}(x, y) \, R(x, y)
\]

(2-17)

To enhance the dark and bright regions of an image, a windowed inverse sigmoid function is utilized \([13]\).

\[
r' = \frac{I'}{I} \, r \\
g' = \frac{I'}{I} \, g \\
b' = \frac{I'}{I} \, b
\]

(2-18)

2.3 Multiple Windowed Inverse Sigmoid Function (MWIS)

MWIS\([13]\) is composed of three main parts: adaptive intensity enhancement, contrast enhancement and color restoration. The first step of adaptive intensity enhancement is to obtain the intensity image, \(I(x, y)\), using NTSC color space. The first assumption to estimate the illumination is that an image may be characterized by two components \([18]\): illumination and reflectance components denoted by \(L(x, y)\) and \(R(x, y)\), respectively. The two functions combine as a product to form an image as in equation (2-8). In such a combination, it is possible to control the illumination and the reflectance components independently. Thus, it is possible to modify the dynamic range of the illumination without any modification in the details. The second assumption is the illumination included in the low frequency components of the image and the reflectance illustrates the high frequency components of the image. The reflectance generally varies much faster than illumination in most regions of the image except for a sudden change of illumination. In the algorithm, as the estimation
of the illumination, the Gaussian low-pass filtered result of the intensity image is used.

The estimated illumination is based on the fact that the illumination changes quite smoothly in the parts of the image illuminated from the same luminous source, but however, it can also present abrupt variation when the scene is illuminated by different light sources in the case of background lights. To reduce the influence of neighborhood areas in which luminance produces a high contrast, which would lead to artifacts, a weighted averaging method is used for bright pixels. So the illumination estimate values for less than 80% of the highest gray-scale value (i.e. 255 for 8-bit image). For the other gray-scale values, it is a weighted average of illumination and intensity values, which decreases the contribution of the illumination linearly as the value of the gray scale increases. This averaging can be mathematically expressed for 8-bit image as:

$$L'(x, y) = \frac{I(x, y) - 204}{51} I(x, y) + \left(1 - \frac{I(x, y) - 204}{51}\right)L(x, y)$$

(2-19)

After obtaining the new illumination estimation, the reflectance estimation can be obtained by Equation (2-8). Before being treated by the enhancement process, the new illumination values $L'(x, y)$ is normalized to the range [0-10] using Equation (2-20)
The normalized illumination values are treated by an enhancement and compression process to increase the illumination values of low-illumination (dark) pixels, and also to reduce the illumination values of high-illumination (bright) pixels using a specifically designed nonlinear multiple windowed-inverse sigmoid (MWIS) transfer function. This process also normalizes the illumination values to the range [0-1] at the same time. This transfer function can be defined as

\[
L''(x,y) = \frac{L'(x,y)}{25.5}
\]  

(2-20)

for 8 bit images.

After obtaining illumination enhancement, the enhanced intensity image can be expressed as

\[
L''_{enh} = \frac{1}{1 + e^{-\alpha L''}} + \frac{1}{1 + e^{-\beta (L''-10)}} - 0.5
\]  

(2-21)

where \( \alpha \) is a parameter to adjust the curve for dark pixels and \( \beta \) is a parameter to adjust the curve for bright pixels. This transfer function is the sum of two inverse sigmoid functions and a constant 0.5 is used to shift down the transfer function. The MWIS transfer function is used to pull up the illumination of dark pixels and to pull down the illumination of bright pixels; meanwhile all pixels are normalized to the range [0 1]. Therefore, dynamic range compression of the illumination is realized.
During this process, a few number of bright pixels that are surrounded by dark pixels are out of the range [0 1]. They are simply clipped. A surrounding pixel-dependent contrast enhancement technique might supply sufficient contrast, even higher than that of the original image, while maintaining the dynamic range compression that was set in the previous step. This adaptive process must be based on the intensity information of the processed (center) pixel and its surrounding pixels. The contrast enhancement process, which was used in AINDANE and IRME, is implemented due to its high quality contrast process and control in the dynamic range expansion.

In the MWIS algorithm, among the basic linear and nonlinear approaches for color consistency, a basic linear color restoration process based on the chromatic information of the input image is applied. This process can be expressed as

\[ I_{\text{enh}}(x,y) = L''_{\text{enh}}(x,y)R(x,y) \]

(2-22)

\[ S_j(x,y) = S(x,y) \frac{I(x,y)}{I(x,y)} \]

(2-23)

where \( j \) represents the red, green, blue spectral band.

2.4 Summary

In this chapter, the theory and principles of the advanced image enhancement techniques in the spatial domain were investigated. These techniques inspired the
development of the LTSN algorithm. Advanced techniques such as AINDANE, IRME, and MWIS are good at dynamic range compression, improving local contrast to achieve high visual quality.
A single curve is not enough to enhance an image consisting of poor lighting, sunlight, shadows, and an overexposed lighting region. There is a need for various curves for enhancing different regions in an image. A locally adaptive nonlinear function, which is capable of providing the desired curves, is required to perform the enhancement of such images.

The LTSN algorithm consists of three important stages that lead to good enhancement results, (a) adaptive intensity enhancement, (b) contrast enhancement, and (c) color restoration.

3.1 Adaptive Intensity Enhancement

The algorithm uses a sine nonlinear transfer function to adaptively compress the dynamic range of the intensity image while maintaining important features.

3.1.1 Intensity Computation

The intensity of the RGB image is found using Equation (3-1) where $IR(x, y)$, $IG(x, y)$, and $IB(x, y)$ represents the R, G, and B values respectively for each pixel at location $(x, y)$. 
Before being treated by the enhancement process, the new illumination values are normalized to the range \([0 \ 1]\) using Equation 3.2.

\[
I(x, y) = \frac{I(x, y)}{255}
\]

(3-2)

3.1.2 Enhancement of Dark and Compression of Bright Pixels

An enhancement and compression process is applied to the normalized intensity values. This improves the illumination values of low-illumination (dark) pixels, while reducing the illumination values of high-illumination (bright) pixels by using a specifically designed sine nonlinear transfer function. This process normalizes the illumination values to the range \([0\ -1]\) at the same time. This transfer function is defined as

\[
I_{\text{enh}}(x, y) = \sin^2 \left( I_n(x, y)^3 \times \frac{\pi}{2} \right)
\]

(3-3)
The transfer function has been used for white board scanning at \( q=0.75 \) [40]. From Equation (3-3) we observe that the sine nonlinear transfer function is dependent on the parameter \( q \). An adaptive technique is used to find the value of \( q \) and it’s found by using a tangent function with a normalized mean of the pixel as its input value.

\[
q = \tan \left[ \frac{I_{M_R}(x,y) \times \pi}{c_1} \right] + c_2
\]

(3-4)
In Equation (3-4) $I_{Mn}(x,y)$ is the normalized mean of the pixel intensity at $(x,y)$ and $c_1$ and $c_2$ are determined empirically determined based on several experiments conducted on typical images. As $c_1$ approaches 2, $q$ becomes infinite. This means that $c_1$ closer to 2; the over-exposed (white) areas become black because of the high values of $q$. $c_1$ should be greater than 2. To compute the bounds, $c_1$ is plotted vs. $q$, for $I_{Mn}$ in Figure 3.2, the range of $c_1$ for various $q$ values is 2.1 to 2.4. As $c_1$ increases from 2.1 to 2.4 the amount of pull down intensity will decrease for brighter pixels. For better results, $c_1$ (by experiment) is set to 2.25. On the contrary, for $q$ values, which are closer to 0, the noise in the extreme dark regions will also be enhanced. Hence the $q$ value’s corresponding to the mean value below 0.2 are considered as extreme dark regions, and $q$ for those pixels can be calculated by

$$q = \left[ \log \left( \frac{2I_{Mn}(x,y)}{c_1} \right) + 2 \right]$$

\(3-5\)

![Figure 3.2 Various curves of Nonlinear Transfer function.](image)
The parameter $c_2$ in Equation (3-4) can be calculated equating the $I_{Mn}(x,y) = 0.2$ in Equation (3-4) to Equation (3-5) to maintain continuity. The extreme bright pixels that has a mean closer to 1 will be processed by a curve with a specific $q$ value. The transfer function in Equation (3-3) is a squared sine function. It can be observed from the curves in Figure 3.2 that this transformation greatly boosts the luminance of darker pixels (regions) and simultaneously decreases the luminance of brighter pixels (regions).

### 3.1.3 Calculation of Mean Image

The mean information of neighborhood pixels is found using 2D discrete spatial convolution with a Gaussian kernel. Gaussian kernel is very close to the way in which the human visual system works. For a $M \times N$ intensity image, 2D discrete spatial convolution can be expressed as

$$I_N(x,y) = \sum_{m=-\frac{M}{2}}^{\frac{M}{2}} \sum_{n=-\frac{N}{2}}^{\frac{N}{2}} I(m,n)G(m+x,n+y)$$

(3-6)

where the Gaussian function can be obtained as

$$G(x,y) = K.e^{-\frac{(x^2+y^2)}{c^2}}$$

(3-7)

$K = \iint G(x,y) \, dx \, dy = 1$ and $c$ is the Gaussian surround space constant which determines the extent of the neighborhood. With a single scale Gaussian mean with smaller $c$ values, the details in the image are clearly visible but they suffer halo
artifacts and poor global impression. For larger c values, global impression is maintained but the details are not clear. In order to have a better balance between global impressions, visibility details, and to minimize the halo artifacts, multi-scale Gaussian mean is considered for the calculation of q. It has been observed that for three scales the proposed algorithm produces better results. The filtering of the original intensity image I(x, y) of size M?N is performed by discrete 2D convolution with a multi-scale Gaussian function as

\[ I_{M}(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(m, n)G_i(m + x, n + y) \]  

(3-8)

where \( G_i \) is the weighted sum of i Gaussian functions with different scales (i=3 in this case). The image is normalized to make the range between 0 and 1 and, hence, \( I_{M}(x, y) \) can be obtained. Multi-level convolution results will be discussed in Section 3.3.

### 3.1.4 High Frequency Boosting

In the intensity enhancement process, there is a chance of losing the sharply detailed regions with rapid illumination changes. The edges and fine details of an image are important in many image processing applications. In order to keep those details, a high frequency filtered image is added to the enhanced image. The computation of high frequency components is achieved by a Laplacian operator.
3.1.4.1 Laplacian mask

The Laplacian is a 2D measure of the second spatial derivative of an image. The Laplacian of an image highlights regions of rapid intensity changes. Image sharpening falls into a category of image processing called spatial filtering. One can take advantage of how quickly or abruptly gray-scale values or colors change from one pixel to the next. First order operators (using first derivative measurements) are particularly good at finding edges in images. The Sobel and Roberts edge enhancement operators are examples of these first order filters, sometimes called gradient filters. The Laplacian operator is an example of an isotropic second order or second derivative method of enhancement. It is particularly good at finding the fine details in an image. Any feature with a sharp discontinuity (like noise, unfortunately) will be enhanced by a Laplacian operator. Thus, one application of a Laplacian operator is to restore fine details to an image that has been smoothed to remove noise. (The median operator is often used to remove noise in an image. The Laplacian $L(x, y)$ of an image with pixel intensity values $I(x, y)$ is given by

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$  

(3-9)

The input image is represented as a set of discrete pixels, thus we use a discrete convolution kernel that can approximate the second derivatives in the definition of the Laplacian.
3.2 Contrast Enhancement

In the enhancement stage, the contrast of the image is downgraded, so the appearance of the image is grayed out. In order to improve the overall quality of the images, a contrast enhancement process must be applied to restore or even enhance the original image. The conventional global contrast enhancement methods simply increase the intensity for bright pixels and decrease the intensity of dark pixels [14]. This process is the opposite of the requirement for our image. Increase in the intensity for bright pixels and decrease in the intensity for dark pixels can be obtained with significant expanded dynamic range. From the other point of view, these methods have limited performance for bringing out fine details where adjacent pixels have small intensity differences about the threshold point after the intensity enhancement process. Therefore, a surrounding pixel (neighborhood)-dependent contrast enhancement technique is needed. A surrounding pixel-dependent contrast enhancement technique might supply sufficient contrast, even higher than that of the original image, while maintaining the dynamic range compression that was set in the previous step. This adaptive process must be based on the intensity information of the processed (center) pixel and its surrounding pixels.

3.2.1 Acquiring the Intensity Information of Surrounding Pixels

The intensity information of surrounding pixels of a M×N grayscale image can be obtained by performing a 2D discrete spatial convolution with a Gaussian kernel which is expressed as
where $c$ is the scale or Gaussian surround space constant.

The estimated illumination is based on the fact that the illumination changes quite smoothly in the parts of the image illuminated from the same luminous source, but however, it can also present abrupt variation when the scene is illuminated by different light sources in the case of background lights. To reduce the influence of neighborhood areas in which luminance produces a high contrast, which would lead to artifacts, a weighted averaging method is used for bright pixels. So the illumination estimate values for less than 80% of the highest gray-scale value (i.e. 255 for 8-bit image) are the illumination that is obtained in Equation (3-10). For the other gray-scale values, it is a weighted average of illumination and intensity values, which decreases the contribution of the illumination linearly as the value of the gray scale increases. This averaging can be mathematically expressed for 8-bit image as:

$$L'(x, y) = \frac{I(x, y) - 204}{51} I(x, y) + \left(1 - \frac{I(x, y) - 204}{51}\right)L(x, y)$$

(3-12)
3.2.2 Intensity Transformation Process

After obtaining surrounding intensity information, it is compared with the intensity value of the center pixel. The result is used to identify the value of the corresponding enhanced intensity pixel $I_{enh}$. These two processes can be defined as

$$E(x, y) = \left[ \frac{I_{conv}(x,y)}{I(x, y)} \right]$$

(3-13)

If $E(x, y)$ is less than 1, $I_{enh}(x, y)^{E(x,y)}$ will be greater than $I_{enh}(x, y)$ (i.e. the center pixel is brighter than the surrounding pixels).

If $E(x, y)$ is greater than 1, $I_{enh}(x, y)^{E(x,y)}$ will be less than $I_{enh}(x, y)$ (i.e. the center pixel is darker than the surrounding pixels).

3.2.3 Contrast Enhancement with Multi-level Convolution Results

As discussed in Section 3.1.3, for better image quality, at the expense of losing some details, multiple convolutions are necessary to achieve a graceful balance between dynamic range compression and tonal rendition. Then, for contrast enhancement, it is be preferable to use a linear combination of multiple convolutions with different scales. Generally, enhancement with a small scale (i.e. a few neighboring pixels can provide intensity information of about the nearest neighborhood pixel) convolution tends to enhance local contrast or fine details and enhancement with a large scale (i.e. large number of neighboring pixels can provide intensity information of the entire image) convolution can provide a global tonality, with a smooth and natural looking image. A medium scale can provide a mixture of
both details and image rendition. Obviously, convolution with multiple scales can yield more complete information on the image’s intensity distribution, and, hence, lead to more balanced image enhancement. The contrast enhancement with multiscale convolutions can be described by the following equations

\[
F_i(x, y) = K \exp\left(-\frac{(x + y)^2}{c_i^2}\right)
\]

(3-14)

\[
I_{conv,i}(x, y) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} I(m, n)F_i(m + x, n + y)
\]

(3-15)

\[
S_i(x, y) = 255I_{enh}^{E(x,y)}
\]

(3-16)

\[
S_i(x, y) = \sum w_i S_i(x, y)
\]

(3-17)

3.2.4 Auto Levels

Auto levels is a commonly used image enhancement function, which provides good processed results for input images that have narrow dynamic range. After contrast enhancement process, some pixels, with values close to the threshold point, have very small intensity differences with their adjacent pixels. This is similar to a contrast stretch process.

For the enhancement of an overexposed image, there might be many colors at the high end. Similarly in the process of enhancement of under exposed images the
low end has many colors. The Auto levels effect detects and fixes this kind of imbalance. It scans through the levels of intensity within the image and chooses a level that should be regarded as black (low intensity) and another that should be regarded as white (high intensity). It then stretches the levels in the image so that all the intensities present lie between the black and the white points. This results in an image with a good span of color intensities.

To soothe the impact of outliers - small numbers of pixels at extreme values of intensity - a clipping percentage is used. By default, the value is 0.5%, which means that the bottom and top 0.5% of pixels will be ignored when determining the black and white points. This process makes the images span an entire range of color intensities.

3.3 Color Restoration

The human visual system has a complex nonlinear mechanism that determines the perceived color by spatial comparisons of color signals across the scene. Therefore, it is a complex problem for digital vision. To deal with this problem, various color restoration methods exist. They involve strong assumptions, such as constant illumination, that are in general, unsatisfied in complex environments.

In the LTSN algorithm, among the basic linear and nonlinear approaches for color consistency, a basic linear color restoration process based on the chromatic information of the input image is applied. This process can be expressed as
\[ S_j(x,y) = S(x,y) \frac{I_j(x,y)}{I(x,y)} \]  

(3-18)

where \( j \) represents the red, green and blue spectral band.

### 3.4 Summary

In this chapter, a new nonlinear image enhancement technique, based on LTSN, has been proposed. The goals of this algorithm are:

a. To enhance the dark pixels and bright pixels in an image at the same time using a special nonlinear transfer function.

b. To adaptively compute the control parameter in the nonlinear function.

c. To apply a contrast enhancement process to preserve important visual details.

d. To restore the color for the RGB images based on the chromatic information of the original image.
CHAPTER 4

GPGPU PLATFORM

Graphics processing units (GPGPUs) have been used in the past as dedicated rendering devices for computers and gaming consoles. In recent years however with the advent of CUDA and OpenCL, commodity GPUs have become increasingly programmable and are now capable of performing much more than graphics specific computations. The specialized rendering hardware provides an advantage for the GPU over the CPU when performing compute-intensive, highly parallel computations.

Figure 4.1 CPU v/s GPU comparison of floating point operations per second
The primary reason for the increasing bandwidth and performance of NVIDIA GPU’s over Intel CPUs is the transistor allocation for the GPU vs. the CPU. The majority of the transistors on the GPU are devoted to data processing rather than flow control and data caching.

![Figure 4.2 Transistor allocations for CPU and GPU](image)

4.1 GPU Hardware Architecture

The C2070 NVIDIA range of GPUs is based on the revolutionary Fermi architecture. The Fermi processor focused on increasing the raw computer horsepower and through architectural innovations offers dramatically increased programmability and compute efficiency. The first Fermi based GPU (Figure 4.3), implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 Streaming Multiprocessors (SMs) of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.
The Fermi architecture greatly improves the performance with its 32 CUDA cores per SM. The Dual Warp Scheduler simultaneously schedules and dispatches instructions from two independent warps. Concurrent kernel execution and Out of Order thread block execution helps in stream processing of the GPU kernels which was used in color restoration in this algorithm.

![Fermi architecture](image)

*Figure 4.3 Fermi architecture*

### 4.1.1 Third Generation Streaming Multiprocessor

The architectural innovations used in the Fermi architecture not only make it the most powerful SM, but also the most programmable and efficient. There are 512 High Performance CUDA cores. Each SM features 32 CUDA processors—a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).
The Fermi Streaming Multiprocessor has 16 load/store units, allowing source and destination addresses to be calculated for sixteen threads per clock. Supporting units loads and store the data at each address to cache or DRAM. There are four Special Function Units execute transcendental instructions such as sin, cosine, reciprocal, and square root. Each SFU executes one instruction per thread, per clock; a warp executes over eight clocks. The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied. The new SFU structure enabled faster computation of the sine transfer function in the LTSN algorithm. The Dual Warp Scheduler stands out from the previous NVIDIA GPU designs. The SM schedules threads in groups of 32 parallel
threads called warps. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently.

4.1.2 Fermi Dual Scheduler

Fermi’s dual warp (Fig 4.5) scheduler selects two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen load/store units, or four SFUs. Warps execute independently and Fermi’s scheduler does not need to check for dependencies from within the instruction stream. Fermi achieves peak performance with the introduction of the Dual Warp Scheduler and the two instruction and dispatch units.

![The Dual Warp Scheduler](image)

*Figure 4.5: The Dual Warp Scheduler*

Most of the instructions can be issued concurrently thus improving the performance of compute bound algorithms. The instructions can be dual issued; two
integer operations, two floating instructions or a mix of integer, floating point, load or store and SFU instructions.

4.2 GPU Software Model

CUDA is the GPU programmable language that has created a lot of attention in the academic and scientific area. A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.

*Figure 4.6 CUDA Hierarchy of threads, blocks, and grids*

CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces. A grid is an array of
thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-Block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization.

4.2.1 Hardware Execution

CUDA’s hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.
CHAPTER 5

CUDA IMPLEMENTATION

In Chapter 3 the hardware model of GPUs were explored and it’s potential to give high computational speeds. CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages. A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results. The massive parallelism it could offer is due to the large number of cores and light weight threads. Computer vision algorithms require fast response times. The video enhancement algorithms processes each frame which are computationally intensive, making them impossible for real time applications if executed sequentially. Each frame is processed for each pixel, making it a very strong candidate for GPU acceleration. They are SIMD instructions and can be offloaded to the GPU and free the CPU for other tasks such as device IO, user interaction or running various background tasks.
The LTSN algorithm is computationally bound but holds great potential to be accelerated because of the parallel nature of the algorithm. The process of enhancing low illumination regions and compressing high illumination or contrast regions in a nonlinear fashion is done in parallel. This chapter deals with the approach taken to accelerate such computationally intense functions on the GPU using CUDA.

5.1 Image Processing with CUDA

Image Processing is done per pixel with the number of light weight threads the CUDA device is able to launch for a kernel. A 2D grid is be laid over the image and is split up into several rectangular sections called blocks as illustrated in Fig 5.1. The image is divided into several blocks of equal size for simplicity. If the image has DIM x DIM pixels, DIM/32 x DIM/32 blocks are launched so and to get one thread per pixel.

Figure 5.1 illustrates a 2D hierarchy of blocks and threads to process a 48 x 32 pixel image using one thread per pixel. The number of threads per block is (16, 16) i.e. 256 threads per block in this illustration.
Since the image is ridiculously small 48 x 32 pixels, blocks with width (48/16, 32/16) i.e. (3, 2) having 16 x16 threads each are launched. The 2D structure makes it easier for the programmer to assign each thread to a pixel for computation for the enhancement of a frame or image.

A condition arises when the dimensions of the image are not a factor of the block dimension. The block dimension in 2D coordinates will be (32, 32) for a 1024 thread per block Fermi architecture GPU. If above equation (DIM/32, DIM/32) for an image which is not a factor of 32, during run time we could get a segmentation error or a clipped image. This is because we haven’t assigned enough blocks for the processing of the image. The following steps are used to find the number of blocks.
The function `iDivUp` returns the number of blocks needed for the successful processing of the image. Below is a code snippet of how the correct number of blocks can be declared for processing the image, where `BLOCK_DIM` is the number of threads in the block.

```c
int iDivUp(int a, int b){
    return (a % b != 0) ? (a / b + 1) : (a / b);
}
```

This is an example code for processing an image in general

```c
__global__ void kernel(unsigned char *ptr , int height, int width){
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = y + x*width;
    ptr[offset] = 255 – ptr[offset];
}
```

The first three lines are the most important lines in the kernel, each thread takes its index within its block as well as the index of its block within the grid and it translates into a unique (x,y) index within the image.

### 5.2 GPU Kernel Processes

The SIMD functions to be accelerated on the GPU are run as kernels as shown in Figure 5.2. Since the approach taken to accelerate the algorithm comprises of the CPU and the GPU, there are data transfers between the device and host during run time. Since the new generation Fermi architecture GPU have high bandwidth of the order of 150GB/s, the data transfers between host and device do not limit the
performance of the algorithm. In Figure 4.2 the orange blocks are the CPU stages and the blue blocks represent the GPU kernels. The luminance intensity parameters are transferred to the GPU pipeline for computing the Gaussian average. The GPU pipeline also consists of the sine nonlinear transfer function calculation. The results are copied back to the CPU and a contrast stretch functionality is carried out. The results obtained from the contrast stretch function are copied to the GPU for rendering. The CUDA device restores the color of the image, so that the fine details and the color of the enhanced image is intact.

The OpenCV Gaussian filter is replaced by Deriche's recursive method [3] implemented in CUDA. This filter uses the previous outputs of the filter as well as the previous inputs. This is known as an Infinite Response filter (IIR), since its response to an input response can last forever. The advantage of using this method over the conventional Gaussian filter method is because the execution time is independent of the filter width. The GPU processes columns of the image in parallel using CUDA programming language. To avoid non-coalesced reads for the row pass we transpose the image and then transpose it back again afterwards. The reason why this approach
is taken rather than reading the rows and columns together is because the highest latency instructions while reading from global memory is 400-600 clock cycles. There is likely to be a performance bottleneck, but optimizations can greatly increase the performance. Coalescing greatly improves the throughput and is critical to small or
memory bound kernels. The Gaussian filter is applied to the luminance for 3 values of sigma and their resultant values are averaged thus utilizing the bandwidth and parallelism of the device to the fullest.

Figure 5.2 illustrates that the Gaussian Averaging output is transferred from the CUDA device to the CPU. The most computationally intensive part of the algorithm is the sine nonlinear transfer function, which is computed in parallel by the GPU in CUDA for each pixel. The parallel hardware and the large number of threads makes it possible to compute the sine nonlinear transfer function for each pixel. This enhances the performance and ensures a very high speed up compared to the CPU that goes through each pixel in one cycle. The output of the transfer function is transferred from the GPU to the CPU. The CPU does the contrast stretch which is efficient on the CPU, and the output of the contrast stretch function is transferred to the GPU color restoration kernel. The GPU color restoration kernel enhances the RGB components. Three kernels are actively dedicated to the color enhancement of the image. Streams were used for asynchronous execution of the kernels, because the three kernels could be executed independent of each other. This approach has an upper edge compared to the execution of the three color restoration kernels in sync. Thus an efficient CPU+GPU system was developed to accelerate the LTSN video enhancement algorithm.
CHAPTER 6

EXPERIMENTAL SETUP AND RESULTS

This chapter deals with the approaches taken to detect the SIMD functions that could be accelerated using CUDA. The experimentation starts with profiling the sequential algorithm. An estimation of the total speedup is estimated using Amdahl’s law to determine whether accelerating the function would give any significant speedup.

6.1 Profiling

To get a comprehensive idea of the system workload distribution we have measured the computational complexity of each individual component of the LTSN enhancement algorithm. Figure 6.1 shows the complexity profiling of the building blocks of the algorithm. The benchmarking was done on an Intel Xeon 5650 processor which has a clock speed of 2.66 GHz, memory size 288 GB and maximum memory bandwidth of 32GB/s.
Benchmarking of the algorithm is important, as running the whole algorithm on the device could lead to an out of memory error. Only 6GB of memory can be allocated on the GPU. If the entire algorithm is run on the GPU hardware, there will not be enough memory for all the data structures used for the computation. For faster execution and streams processing page locked memory was allocated and thus virtual memory translation will not be possible.

The most time consuming part of the algorithm from Fig 6.1 is the Sine Nonlinear Function which constitutes 47% of the overall processing time. Even the most efficient Sine calculation based on the Maclaurin series take 47% of CPU time for processing an image. The proposed design of the GPU kernels has led to massive parallelism and significant acceleration. The analysis of the acceleration and speedup is mentioned in later sub sections in this chapter. This time consuming function have been efficiently mapped into the GPU with optimized CUDA programming.

From the profiling chart in Fig 6.1 it is also evident that the Gaussian smoothening and averaging function accounts to 27% of the total processing time. The OpenCV Gaussian smoothening function has been replaced by a more efficient
recursive Gaussian smoothening filter. The recursive Gaussian smoothening and averaging function have been successfully mapped into the GPU using CUDA.

The color restoration functionality done on the CPU took 16\% of the total processing time and thus it was taken into consideration for acceleration. Since the same kernel were used for the enhancing the red, blue and green components, stream processing was possible. A study was done to compare the results of stream processing for speed and accuracy. For larger images, the addition of a color restoration GPU kernel yielded better results.

The approach taken for optimizing and accelerating the video enhancement algorithm was using CPU and a GPU, thus it was important to benchmark the algorithm. Some functions run fast on the CPU, so trying to export those functions to the GPU could result in little or no speedup. In some cases it could even lead to a decrease in processing speed due to the memory bottleneck, therefore it’s important to choose the SIMD instructions that need to be offloaded to the GPU. Table 6.1 describes the time taken for the GPU to do memory transfers.

<table>
<thead>
<tr>
<th>Image Sizes</th>
<th>Size(bytes)</th>
<th>Data Transfer Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>640X480</td>
<td>1228800</td>
<td>0.292</td>
</tr>
<tr>
<td>800X600</td>
<td>1920000</td>
<td>0.394</td>
</tr>
<tr>
<td>1024X1024</td>
<td>4194304</td>
<td>0.8315</td>
</tr>
<tr>
<td>1600X1200</td>
<td>7680000</td>
<td>1.5455</td>
</tr>
<tr>
<td>2272X1704</td>
<td>15485952</td>
<td>4.3231</td>
</tr>
<tr>
<td>2816X2816</td>
<td>31719424</td>
<td>5.2675</td>
</tr>
<tr>
<td>4480X3360</td>
<td>60211200</td>
<td>9.5865</td>
</tr>
</tbody>
</table>

*Table 6.1 GPU memory copy time*
From the above table its evident that memory transfer times to and from the GPU are bottlenecks to the overall performance of the algorithm. Thus compute bound functions are chosen such that the memory copy time can be leveraged with the computational speed of the GPU.

6.2 Sine Non Linear Function Performance

The Sine Non Linear function calculation, a critical component of the LTSN video enhancement algorithm was computed on the GPU using CUDA. OpenCV 32 bit floating point structures were copied to the device and the computation was done for each pixel. The speedup achieved by GPU computation of the functionality was above expected. In Figure 6.2 we could see that as the resolution of the frame increases, the difference between the execution times of the CPU and GPU increases. The computation done by the Sine Nonlinear function kernel on the GPU gives tremendous performance improvement compared to the CPU for high resolution frames.

![Figure 6.2 Performance analysis of Sine Nonlinear function](image)
<table>
<thead>
<tr>
<th>Image Sizes</th>
<th>CPU Time(s)</th>
<th>GPU Time(s)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>400X300</td>
<td>0.015776</td>
<td>0.001246</td>
<td>12.66621705</td>
</tr>
<tr>
<td>640X480</td>
<td>0.043752</td>
<td>0.002056</td>
<td>21.280107</td>
</tr>
<tr>
<td>800X600</td>
<td>0.067337</td>
<td>0.003178</td>
<td>21.18819265</td>
</tr>
<tr>
<td>1024X768</td>
<td>0.107272</td>
<td>0.004832</td>
<td>22.20102031</td>
</tr>
<tr>
<td>1600X1200</td>
<td>0.247454</td>
<td>0.010744</td>
<td>23.03118863</td>
</tr>
<tr>
<td>2272X1704</td>
<td>0.509652</td>
<td>0.015476</td>
<td>32.93112695</td>
</tr>
<tr>
<td>2816X2112</td>
<td>1.02599</td>
<td>0.038282</td>
<td>36.80070633</td>
</tr>
</tbody>
</table>

Table 6.2 Execution times and speed up of Sine Nonlinear function

The –fastmath function, when mentioned in the Makefile during compilation and execution, increases the performance is improved to a great extent. The fastmath function uses the “special function unit” in each multiprocessor taking one instruction, whereas the normal implementation can take many instructions. If we look at the disassembly listing of the code we could find that the number of instructions is greatly reduced when fastmath functions are used. In situations where accuracy is needed more than speed, we tend to avoid these functions, but in image processing applications and computer vision applications, processing speeds gains the upper edge over accuracy. Using these functions, the output results did not look any different from what was achieved by the ordinary functions or the CPU implementation.

By achieving a good speed on the sine nonlinear function, which was close to 36x times for high resolution images, the overall performance of the algorithm was improved to around 1.8 times. This experiment was a good way to prove the power of the GPU on compute bound operations. The large number of cores in the GPU with a functional unit makes it ideal for computing arithmetic and logic operations for large
data[5][7]. Figure 6.3 shows the speedup gained for various frame resolutions/image sizes and we can notice the large increase in speedups for higher resolution images.

![Graph showing speedup gained by the Sine Nonlinear function for various image sizes](image)

**Figure 6.3 Speedup gained by the Sine Nonlinear function for various image sizes**

### 6.3 Gaussian Smoothening and Averaging Kernel Performance

The Gaussian smoothening was done for 3 values of sigma and their average was computed on the GPU using CUDA programming. Initially when the recursive Gaussian blur algorithm [2] was implemented in CUDA for 3 different sigma levels, we had to copy 3 data structures to the OpenCV structures for the CPU to process these as inputs to the averaging function. In order to minimize the number of data structures being copied from and to the device, we decided to do the averaging function within the CUDA kernel so that the number of data structures copied from the device is minimized to one. And only one structure, which is the mean of the smoothened luminance, is copied to the OpenCV structure. This greatly increased the efficiency as the copy to the OpenCV structure took significant amount of CPU time. Therefore we reduced the number of memory copies from device to host and from host to OpenCV from three to one.
After doing the above optimizations we compared the performances of the OpenCV Gaussian blur function and averaging run on the CPU to the recursive Gaussian blur and averaging function programmed in CUDA and executed on the GPU as shown in Figure 6.4. From the graph we could visualize that the performance increases with increase in resolution of the input image due to the massive parallelism available in GPUs. For larger resolution images, a speedup of 2.5 was achieved. Thus we have reduced 23% of the total execution time of the algorithm to around half by implementing a recursive Gaussian smoothening algorithm in CUDA.

Table 6.2 illustrates the execution times for various image sizes; we could see that the speedup increases with the increase in the image size. Figure 6.5 gives the graph that plots the image size versus speedup and it touches 2.5 for large images. As the input data increases below 8MP we would see even higher speedups, thus enabling high resolution processing of the image or video.
<table>
<thead>
<tr>
<th>Image Size</th>
<th>CPU Time(s)</th>
<th>GPU Time(s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>400X300</td>
<td>0.008821</td>
<td>0.006168</td>
<td>1.43020435</td>
</tr>
<tr>
<td>640X480</td>
<td>0.024397</td>
<td>0.012934</td>
<td>1.88635028</td>
</tr>
<tr>
<td>800X600</td>
<td>0.036452</td>
<td>0.019137</td>
<td>1.90481117</td>
</tr>
<tr>
<td>1024X768</td>
<td>0.069598</td>
<td>0.028894</td>
<td>2.40871037</td>
</tr>
<tr>
<td>1600X1200</td>
<td>0.163036</td>
<td>0.063584</td>
<td>2.56412434</td>
</tr>
<tr>
<td>2272X1704</td>
<td>0.316641</td>
<td>0.123661</td>
<td>2.56055668</td>
</tr>
<tr>
<td>2816X2112</td>
<td>0.687516</td>
<td>0.251589</td>
<td>2.73269499</td>
</tr>
</tbody>
</table>

*Table 6.3 Execution times and speed up of Gaussian smoothing algorithm*

![Gaussian Smoothening and Averaging Speedup on GPU](image)

*Figure 6.5 Speedup gained by Gaussian smoothing and averaging*

### 6.4 Color Restoration Kernel Performance

The color restoration component constitutes 16% of the total runtime. There were 5 data structures to the copied to the device memory, and 3 data structures to be copied from device memory to the host. Since the enhancement was done on the Tesla C2070 range of NVIDIA GPUs, there was enough bandwidth to make the 7 memory transfers without limiting the performance of the color restoration function. The Tesla C2070 has the Fermi architecture with memory bandwidth of 144GB/s and a memory interface of 384-bit which ensures that a memory bottleneck will not occur.
The Fermi architecture supports streams, which was used for this function because there were scope for parallel execution of kernels and asynchronous memory transfers. The application of streams increased the performance by around 26% compared to the non-stream version of the color restoration function. Including the Color Restoration kernel gave a 16% speed up to the overall runtime of the algorithm, even with the eight memory transfers happening. Fig 5.5 shows a comparison of the color restore function run on the CPU and GPU. We could find that for larger images, the GPU excels in performance due to its massive parallel hardware.

![Color Restoration](image)

Fig 6.6 Performance analysis of Color Restoration

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Host to Device Memory Transfers</td>
<td>0.25</td>
</tr>
<tr>
<td>3 Device To Host Memory Transfer</td>
<td>0.15</td>
</tr>
<tr>
<td>Color Restoration Computation</td>
<td>0.011</td>
</tr>
<tr>
<td>Total Time</td>
<td>0.411</td>
</tr>
</tbody>
</table>

Table 6.4 Color Restoration GPU kernel timing for 640X480 pixel image
Table 6.5 Execution times and speed up of Color Restoration function

<table>
<thead>
<tr>
<th>Image Size</th>
<th>CPU Time(s)</th>
<th>GPU Time(s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>640X480</td>
<td>0.0119739</td>
<td>0.00438251</td>
<td>2.73220141</td>
</tr>
<tr>
<td>800X600</td>
<td>0.023701</td>
<td>0.00771409</td>
<td>3.072429801</td>
</tr>
<tr>
<td>1024X768</td>
<td>0.0411295</td>
<td>0.0113441</td>
<td>3.625629182</td>
</tr>
<tr>
<td>1600X1200</td>
<td>0.0741003</td>
<td>0.0184525</td>
<td>4.015732286</td>
</tr>
<tr>
<td>2272X1704</td>
<td>0.147664</td>
<td>0.0335725</td>
<td>4.398361754</td>
</tr>
<tr>
<td>2816X2112</td>
<td>0.311041</td>
<td>0.0634061</td>
<td>4.90553748</td>
</tr>
<tr>
<td>4480X3360</td>
<td>0.583698</td>
<td>0.116205</td>
<td>5.023002453</td>
</tr>
</tbody>
</table>

Figure 6.7 Speedup gained by Color Restoration

For large images for around 15Megapixel we could find that the color restore function gets around 5 times speed up. Table 6.3 illustrates the execution times taken for the CPU and GPU to run the color restore function. Table 6.4 illustrates that even though the number of data transfers from the CPU to the GPU device are 5 and the number of transfers from GPU to the CPU are 3, significant speedup is achieved over the CPU execution time. The high bandwidth of the GPU architecture ensures these memory transfers do not affect the performance of the algorithm. Figure 6.7 shows
the speedup achieved by the GPU for the various image sizes. We again notice the rate of increase of speedup as the size of the data increases.

In the next section we will explore the total speedup of the video enhancement algorithm. We have accelerated the major 3 components of the LTSN algorithm and we investigate on how it has impacted the overall performance.

6.5 Overall Algorithm Analysis

The approach taken was to enhance the algorithm using the CPU and the GPU. Certain SIMD functions execute really fast on the CPU, so only the functions that took a lot of CPU time was transferred to the GPU like explained in the chapter. The accelerated GPU kernel functions replaced the CPU functions and the resulting output was accurate.

Figure 6.8 illustrates the performance analysis done for various image sizes A CPU+GPU versus CPU runtime is plotted against image sizes varying from low resolution to very high resolution images. As noticed, we see an increase in
performance for bigger images, compared to the CPU. Table 6.3 gives the finer details of the execution times and the speedup achieved.

<table>
<thead>
<tr>
<th>Image Sizes</th>
<th>CPU Time(s)</th>
<th>CPU+GPU Time(s)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>640X480</td>
<td>0.106786</td>
<td>0.049</td>
<td>2.17930612</td>
</tr>
<tr>
<td>800X600</td>
<td>0.163678</td>
<td>0.0748869</td>
<td>2.18566932</td>
</tr>
<tr>
<td>1024X768</td>
<td>0.28046</td>
<td>0.119197</td>
<td>2.35291157</td>
</tr>
<tr>
<td>1600X1200</td>
<td>0.616025</td>
<td>0.259966</td>
<td>2.3696368</td>
</tr>
<tr>
<td>2272X1704</td>
<td>1.22408</td>
<td>0.495642</td>
<td>2.46968578</td>
</tr>
<tr>
<td>2816X2112</td>
<td>2.54791</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>4480X3360</td>
<td>4.84</td>
<td>1.49</td>
<td>3.24832215</td>
</tr>
</tbody>
</table>

Table 6.6 Execution times and speed up of LTSN algorithm

A minimum speedup of 2 was achieved for low image resolutions and for a very high resolution image, we got an overall speedup of 3.24. Figure 6.9 gives the plot of the image sizes versus the speedup attained on the CPU+GPU configuration. The speedup curve increases as the image size increases in the accelerated LTSN video enhancement algorithm. By accelerating the algorithm for a single image or a frame we have come closer to the objective to produce a faster video enhancement algorithm to achieve real time results.
The accelerated LTSN algorithm was first tested for various resolution images and the following outputs were obtained.
Figure 6.12 Original image and enhanced image

Figure 6.13 Original image and enhanced image
Figure 6.14 Original image and enhanced image

Figure 6.15 Original image and enhanced image
Figure 6.10 – 6.17 shows images that consist of both dark and bright regions. The enhancement brought up the intensities of pixels in the dark region and brought down
the intensities of pixels in the bright region. The quality of the enhancement was not affected by the acceleration done on the GPGPU.

![Video Performance Analysis](image)

**Fig 6.18 FPS of the LTSN video enhancement for video resolutions**

<table>
<thead>
<tr>
<th>Video Resolution</th>
<th>Standard</th>
<th>CPU (fps)</th>
<th>GPU (fps)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 240</td>
<td>HVGA (portrait)</td>
<td>39.8338133</td>
<td>54.0137627</td>
<td>1.355977703</td>
</tr>
<tr>
<td>640 x 480</td>
<td>VGA, MCGA</td>
<td>9.36452343</td>
<td>21.2765957</td>
<td>2.272042553</td>
</tr>
<tr>
<td>1280 x 720</td>
<td>720p (WXGA-H, min.)</td>
<td>3.26112123</td>
<td>9.24052154</td>
<td>2.833541245</td>
</tr>
<tr>
<td>1920 x 1080</td>
<td>HDV 1080i</td>
<td>1.46031236</td>
<td>4.40794665</td>
<td>3.018495744</td>
</tr>
</tbody>
</table>

**Table 6.7 Execution times and speedup of LTSN video algorithm**

![GPU SpeedUp for Video frames](image)

**Fig 6.19 GPU speed up for the LTSN video enhancement algorithm for video resolutions**
The LTSN algorithm was tested for performance on videos of various frame sizes and the performance increased for higher resolutions. 640 x 480 resolution obtained only close to 10 frames per second (fps), which cannot be used for real time video processing. The GPU results were close to 21 fps which was close to the real time rendering time essential for computer vision algorithms.
CHAPTER 7

CONCLUSION AND FUTURE WORK

This thesis has explored the GPU and CPU implementation of the LTSN algorithm. The algorithm was processed per frame on the CPU and GPU to attain speedups close to real time. A video of 640×480 resolution was able to get 21 frames per second compared to 9 frames per second when it was run only on the CPU using OpenCV functions. The compute bound operations like the calculation of a sine nonlinear transfer function got speedups close to 36× which comprised 47% of the total runtime of the algorithm. The Gaussian function which comprised of 23% of the total run time got on an average a 2× speedup. Color Restoration was processed on the GPU as streams and a speedup on an average of 2× was achieved. Thus an overall speedup to up to 3× was got for high resolution frames.

For this work a CPU core and multiple GPU cores were used to process a single frame. However it should be possible to exploit multiple CPU cores for the processing of multiple frames in parallel. This would further increase the throughput as there are dedicated CPU cores processing each frame in parallel. The SIMD instructions that run faster on the GPU could process multiple frames.
REFERENCES


