A REAL-TIME IMPLEMENTATION OF GRADIENT DOMAIN HIGH DYNAMIC RANGE COMPRESSION USING A LOCAL POISSON SOLVER

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Lavanya Vytla

May, 2010

A REAL-TIME IMPLEMENTATION OF GRADIENT DOMAIN HIGH DYNAMIC RANGE COMPRESSION USING A LOCAL POISSON SOLVER

Lavanya Vytla

Thesis

Approved:	Accepted:
_____	_____
Advisor Dr. Joan E. Carletta	Dean of the College Dr. George K. Haritos
_____	_____
Committee Member Dr. Robert Veillette	Dean of the Graduate School Dr. George R. Newkome
_____	_____
Committee Member Dr. Tom Hartley	Date
_____	
Department Chair Dr. Alex De Abreu-Garcia	

ABSTRACT

This thesis presents a real-time hardware implementation of a gradient domain high dynamic range compression algorithm; this technique is useful for processing high dynamic range (HDR) images to be able to view them on standard display devices. The hardware implementation is described in VHDL and synthesized for a field programmable gate array (FPGA) device. The maximum operating frequency achieved is fast enough to process high dynamic range videos with one megapixel per frame at a rate of about 100 frames per second. The hardware is tested on standard HDR images from the Debevec library. The output images have good visual quality and are similar to the output images obtained using floating-point arithmetic. An alternate hardware implementation that does not involve any multipliers is also discussed.

The inverse gradient transformation required for reconstructing back the image from the manipulated gradients is implemented by means of a local Poisson solver that utilizes only local information around each pixel along with special boundary conditions. The local Poisson solver requires a small and fixed amount of hardware and memory for any image size. The quality of the images obtained using our local Poisson solver for both fixed-point and floating-point implementations is compared to the quality of images produced by a system that solves the Poisson equation using a full-image fast Fourier transform (FFT).

DEDICATION

Dedicated to my parents, sister, and Chaitu.

ACKNOWLEDGEMENTS

I would like to thank my committee members Dr. Joan E. Carletta, Dr. Robert Veillette and Dr. Tom Hartley for their guidance and support throughout my Master's program. I would like to specially thank Dr. Firas Hassan for helping me throughout this thesis work. I would also like to thank Dr. Joan Carletta for shaping my thoughts and research work into a good manuscript.

I am much obliged for the assistance provided by the Department of Electrical and Computer Engineering for supporting me as a graduate assistant for Circuits I and II Labs. I would like to specially thank Prof. Kult for making my teaching experience enjoyable and memorable. I appreciate Mrs. Gay Boden for her help right from the day I have received admission into the University of Akron.

I owe my heartfelt regards to my dad, mom, sister, Chaitu, friends and God who have been my constant source of inspiration, motivation and encouragement.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

I. INTRODUCTION 1

II. BACKGROUND AND RELATED WORK 7

2.1 The Reinhard Operator 8

2.2 Fattal's Operator for Local Tone Mapping 10

2.2.1 Gradient Transformation 12

2.2.2 Attenuation Factor Computation 12

2.2.3 Gradient Manipulation 14

2.2.4 Inverse Gradient Transformation 14

2.2.5 Inverse Logarithm Transformation 15

III. SOLVING THE POISSON EQUATION LOCALLY 16

3.1 Derivation of the Poisson Equation 17

3.2 Poisson Solvers 19

3.3 The FFT-Based Poisson Solver 21

3.4 Our Proposed Local Poisson Solver 24

IV. PRELIMINARIES TO DEVELOPMENT OF A REAL-TIME IMPLEMENTATION OF FATTAL'S OPERATOR 37

4.1 Attenuation Parameters 37

4.2 Obtaining Gaussian Pyramids 43

V. HARDWARE IMPLEMENTATION 47
5.1 Conversion to Floating-Point 51
5.2 Attenuation Factor Computation 52
5.3 Gradient Transformation and Gradient Manipulation 59
5.4 Local Poisson Solver 61
5.5 Inverse Logarithm Transformation 62
VI. SYNTHESIS AND SIMULATION RESULTS 63
6.1 Hardware Cost and Synthesis Results 63
6.2 Simulation Results 66
6.3 Quality Measurement 72
VII. CONCLUSIONS AND FUTURE WORK 74
BIBLIOGRAPHY 78

LIST OF TABLES

Table Page

3.1 Complexity of Poisson solver algorithms. 19

6.1 Summary of hardware synthesis report. 65

6.2 PSNR values for floating-point and fixed-point implementations using our local Poisson solver. 72

LIST OF FIGURES

Figure Page

2.1 Block diagram of Fattal's operator. 11

3.1 The original Nave image. 34

3.2 Output images obtained using zero Dirichlet and semi-implicit boundary conditions on the Nave image.
(a) Zero Dirichlet boundary condition 35
(b) Semi-implicit boundary condition 35

4.1 Experiment showing the effect of the parameter *a* on the contrast of the Nave image.
(a) *a*=0.2 (b) *a*=1.5 (c) *a*=2.5 40

4.2 Experiment showing the effect of the parameter *b* on the tone mapping of the dynamic range of the Nave image.
(a) *b*=0.6 (b) *b*=0.8 (c) *b*=0.85 41
(d) *b*=0.9 (e) *b*=0.95 (f) *b*=1.0 41

4.3 Histogram for Nave with *a*=1.5 and *b*=0.9 42

4.4 The original gray scale Nave image in logarithmic domain. 44

4.5 Comparison of four-level Gaussian pyramid images, calculated using a full exponential and powers of-two approximation 46

5.1 Block diagram of Fattal's operator on a 3×3 window. 48

5.2 High level view of our hardware. 49

5.3 Conversion to floating-point block. 51

5.4 Hardware block to calculate the central difference for a single scale in a floating-point (mantissa and exponent) format. 54

5.5 Hardware to compute attenuation factor from the central differences of the five scales in floating-point (mantissa and exponent) format. 56

5.6 Central differences for a 3×3 window. 58

5.7 Hardware to obtain gradients and manipulated gradients. 60

6.1 The Nave image obtained using the proposed fixed-point and floating-point implementations of the Fattal's operator.
(a) as processed using the proposed fixed-point implementation of the Fattal's operator. 68
(b) as processed using a floating-point implementation of the Fattal's operator. 68

6.2 High dynamic range images from the Debevec library [9], processed using the proposed hardware implementation of Fattal's operator, and converted to color using MATLAB.
(a) Nave (b) GroveC (c) VineSunset 69
(d) GroveD (e) Rosette (f) Memorial 69

6.3 The Nave image after a four-level gradient domain tone mapping, using three different approaches to solving the Poisson equation.
(a) Full-image FFT as Poisson solver 70
(b) 8×8 block-based FFT as Poisson solver 70
(c) Local Poisson Solver 70

7.1 Comparison of the Nave image obtained using hardware with and without multipliers.
(a) using hardware without multipliers 75
(b) using hardware with multipliers 75

CHAPTER I

INTRODUCTION

Ordinary digital images use eight bits per pixel for each of the red, green and blue channels. Each channel can therefore take only 256 different values; this range, which is approximately two orders of magnitude, is not sufficient to represent the real world. The dynamic range of illumination in a real world scene is high, on the order of about 10^8:1, and therefore requiring about 32 bits per pixel to represent fully. The human eye itself can adapt to changes in illumination of about 10 orders of magnitude.

Modern digital cameras are equipped with advanced computer graphics for producing high-resolution images that meet the increasing demand for more dynamic range, color depth, and accuracy. Increasing use of such cameras has led to the emerging field of high dynamic range (HDR) imaging. HDR images can represent a large part of the vast dynamic range of illumination in real world scenes. They can be either captured by using HDR cameras that fuse images from multiple sensors, or compiled by combining multiple exposures of a scene into a single image. HDR images are encoded in special formats for storage, as they represent a wider range than the standard 24-bit RGB format [1]. HDR images find wide application in remote sensing, medical imaging, and night vision; all these applications share the need for highly detailed images.

The standard 24-bit RGB format is a good match to standard display devices. Unfortunately, standard display devices, with their small dynamic range of the order of 256:1, are not capable of displaying raw HDR images. In order to display an HDR image on a standard display device, its dynamic range must first be compressed, ideally in a way that preserves image details. This compression of an HDR image to produce a low dynamic range image is called *tone mapping*. There are several techniques for tone mapping. They are broadly classified into: *global tone mapping*, which applies a single fixed mapping function to each pixel in an image, and *local tone mapping*, which adapts the mapping function used for a pixel depending on the neighborhood of the pixel. Local tone mapping operators (TMOs) are generally accepted as giving better quality output images than global TMOs [1].

Implementing TMOs in real time is a challenging task, but is also required if the TMO is to be embedded within a camera as part of an electronic viewfinder, or within a display device for real-time display of HDR video. TMOs are generally implemented in software. A local TMO proposed by Fattal [2] works by manipulating the gradients in the HDR image, based on the fact that human eye is more sensitive to relative changes in local intensities than to absolute intensities. Fattal's TMO is implemented on a gray scale (luminance) image in the logarithmic domain, as gradients obtained from the logarithms of the input pixels are ratios, or relative changes, in the linear domain. Gradients of high magnitude, which correspond to parts of the image to which the eye is highly sensitive, are attenuated; this compresses the dynamic range of the output image without much effect on output image quality. Similarly, gradients of small magnitude are magnified so that fine details that would not otherwise be visible can be seen. The attenuation factor is

computed using central differences around each pixel; thus, tone mapping is local in nature since the mapping function varies from pixel to pixel. Once the gradients have been manipulated, an inverse gradient transform is done to find an image that most closely corresponds to the manipulated gradients. This reconstructed output image can be displayed on standard display devices, while preserving details in both dark and bright regions of the image. Fattal's operator helps in preserving local contrast and results in few or no visual artifacts in the output image.

The inverse gradient transform is equivalent to solving a Poisson equation, and solving the Poisson equation is the most computationally intensive part of any gradient domain method. One straightforward method involves applying a two-dimensional fast Fourier transform (FFT) to the entire image. Any Poisson solving technique, including the FFT, that requires the entire image to find a single pixel of the output image, requires that an entire image frame be buffered before any of the output image can be produced. This requires an amount of memory dependent on the image size. This makes the hardware size dependent on image size, introduces one image frame of lag in the computation, and makes implementation in real time difficult.

While Fattal's operator is generally accepted as a high quality local tone mapper, it is difficult to implement in real time because solving the Poisson equation for the entire manipulated gradient image is costly and slow. The need for enough memory to buffer the entire image is also a limitation. The goal of this research is to implement hardware for a real-time gradient domain local TMO:

- that produces a faithful reproduction of the input image, with good detail.
- that does not introduce visual artifacts in the output image.

- that uses a simplified Poisson solver that can be implemented in real time.
- that uses the same hardware for any size image.

The HDR local TMO proposed in the thesis relies on a novel technique for solving the Poisson equation. Our local Poisson solver [3] gives hardware that is independent of image size, and amenable to real-time implementation. Rather than work with the entire image at once to find the complete output image, our local Poisson solver finds one pixel of the output image at a time, considering it to be the center pixel in a 3×3 neighborhood from the input image, and solving a Poisson equation for an image consisting only of that neighborhood. Zero Dirichlet boundary conditions are assumed beyond the neighborhood. To find the complete output image, the local Poisson solver is slid over the entire image. The implementation uses a small and fixed hardware with small memory buffers. The hardware complexity of the method is independent of the image size, and although the method produces a solution that is different from the traditional one, the output images produced are of high quality.

The Poisson equation plays an important role in gradient domain image processing techniques, including tone mapping. It also finds wide application throughout photography; different photography artifacts, such as lighting, shadow, and reflection [4]-[6], can be removed by processing the gradients of the image and solving the Poisson equation to reconstruct a processed image. In addition, images can be fused together by merging their gradients and solving the Poisson equation [7]. The Poisson equation is also used in an image editing program allowing artists to paint in the gradient domain with real-time feedback [8]. While the proposed local Poisson solver is used in this thesis in the context of tone mapping, it could be used in these other applications as well.

Our hardware implementation of Fattal's operator is designed assuming the use of fixed-point arithmetic; a careful fixed-point quantization for all the arithmetic computations is done so as to obtain an output that closely matches the output that is obtained when the method is implemented using floating-point arithmetic. The hardware for Fattal's operator using the proposed local Poisson solver has been successfully described in VHDL and synthesized using Altera Quartus tools for a Stratix II field programmable gate array (FPGA) device. An operating frequency of 114.18MHz is achieved. The maximum operating frequency achieved is fast enough to process high dynamic range videos with one megapixel per frame at a rate of about 100 frames per second. The hardware has been tested on HDR images from the Debevec library [9], and produces output images having good visual quality. A small and fixed amount of hardware and memory is needed for implementation of the Poisson solver, regardless of image size.

The remainder of this document is organized as follows. Related work is described in Chapter 2. Chapter 3 introduces the Poisson equation and the proposed local Poisson solver, considering the effect of various boundary conditions carefully. There are several parameters that have to be chosen to get a good quality output from Fattal's operator. Chapter 4 describes how a series of MATLAB simulations were done to choose the best values for these parameters. Chapter 5 details the hardware implementation for Fattal's operator using the local Poisson solver. It also describes a possible multiplierless implementation of the technique. Chapter 6 shows the hardware costs and synthesis details when Fattal's operator using our local Poisson solver is implemented on an FPGA device. It also shows simulation results that indicate how the hardware implementation of

Fattal's operator performs on HDR images from the Debevec library [9]. Finally, conclusions are drawn and possible future work is described in Chapter 7.

CHAPTER II

BACKGROUND AND RELATED WORK

Tone mapping operators can be classified according to whether their operators are global or local. Because global tone mapping operators do the same transformation on every pixel, they lend themselves well to parallel implementations based on lookup tables. However, it is not possible for a single transformation based on global parameters to properly eliminate the effects of illumination in images for which illumination varies locally. In contrast, local tone mapping operators use transforms that vary locally with the neighborhood of the pixel. Overall, local tone mapping operators produce higher quality enhanced images but also have some drawbacks. Visually, their main disadvantage is the appearance of "halo" artifacts; these artifacts are shadow effects around bright regions of an image. Also, local tone mapping operators require complex computations, and implementing them in real-time is a challenging task.

Two well-known local tone mapping operators that are of special interest to us are the Reinhard local TMO [10] and Fattal's local TMO [2]. Neither operator suffers from significant halo artifacts, and both involve relatively moderate amounts of computation. Other local tone mapping operators include iterative methods [11], nonlinear filters [12]-[14] and image appearance models [15], [16]; however, there is no published evidence of real-time implementation of these more complex methods.

2.1 The Reinhard Operator

The Reinhard operator normalizes each pixel of an image according to a weighted sum of the logarithmic global average of the image and the local luminance surrounding the pixel. The local luminance is estimated by averaging the pixels in a scale image selected from among a set of scale images that form a Gaussian pyramid around the pixel. The Gaussian pyramid is an image pyramid that consists of a set of copies of the original image with both sample and resolution decreased in regular steps [28]. Computing the scale images of the Gaussian pyramid is a matter of applying a series of two-dimensional Gaussian-weighted low pass filters to the original image using a different standard deviation at each scale. Each scale image provides an estimate of the local luminance in a differently-sized neighborhood around the pixel; the Reinhard operator uses the largest neighborhood that does not contain drastic changes in illumination.

Although there is some evidence in the literature [17]-[20] that the Reinhard operator can be implemented in real-time, including work by our own team in [20], the algorithm suffers from two major limitations. One limitation lies in the fact that an entire image is needed to properly compute its logarithmic global average; in a video application, implementation of this global average would result in an entire frame of latency, since an entire frame has to be received to begin processing. Implementation would also require a buffer large enough to store the entire frame. If the video frame rate is fast enough, this limitation can be overcome by using the logarithmic global average of the previous frame to normalize the current frame.

The second major limitation of the Reinhard operator is that it uses a Gaussian pyramid with nine scales, with the largest surround lying in a 64×64 pixel window. The

Gaussian kernels are computed on the pixels in the linear domain, which requires a large number of bits for each pixel; as a result, the hardware requires a large amount of embedded memory. Further, the computations, which involve exponents, are of high complexity. The Gaussian pyramid in general is constructed using a Gaussian function centered at zero:

$$G(i, j, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(i^2+j^2)}{2\sigma^2}} \tag{2.1}$$

where i and j are indices indicating distance from the center and σ is the scale of the Gaussian surround, or standard deviation. The standard deviation controls the rate of decay of the Gaussian surround around the pixel. The use of approximate Gaussian surrounds around each pixel implemented with an accumulator-based approach can reduce the complexity of the operator without significantly affecting the quality of the output [20]. However, the memory usage remains high, as described in [20], and this is reflected in the size of hardware needed to implement the operator.

An alternative local TMO, Fattal's operator, does a gradient transformation of the original logarithmic luminance input image to accomplish tone mapping [2]. It attenuates the large gradients that are responsible for the image's high dynamic range, while at the same time magnifying small gradients that correspond to detail that may otherwise not be visible. This requires that an attenuation factor be calculated for each pixel in the gradient domain image. The attenuation factors are then used to manipulate the image in the gradient domain. An inverse gradient transformation is done to bring the image back into the logarithmic luminance domain. Finally, an inverse logarithm transformation brings

the now low dynamic range image to a linear luminance domain, appropriate for use on standard display devices.

Like the Reinhard operator, Fattal's operator calculates the attenuation factor around each pixel in a multi-scale fashion using a Gaussian pyramid. However, four scales, with the largest surround lying in a 32×32 pixel window, are enough. Further, the Gaussian kernels are computed on the pixel values in the logarithmic domain, so that a smaller number of bits are needed for each pixel. Thus, the required embedded memory is smaller than is needed for Reinhard's method. Moreover, the operator preserves local contrast, and introduces no visual artifacts. For this reason, we chose to focus on Fattal's operator in this thesis. In the remainder of this chapter, the various steps in Fattal's operator are described in detail with the help of a block diagram.

2.2 Fattal's Operator for Local Tone Mapping

Fattal [2] has proposed an effective method for compressing the dynamic range of HDR images. The block diagram for Fattal's operator is shown in Figure 2.1. The inputs to the system are the logarithmic luminance of a HDR image, and its four scale images. A HDR image in the logarithmic luminance domain can be produced using a HDR camera. The scale images of the input image, obtained from a Gaussian pyramid, can be produced in real time using the hardware described in [20]. The *gradient transformation* block transforms the input image into the gradient domain, computing gradients of the logarithmic luminance of the HDR image using forward differences. The *attenuation factor computation* block computes the attenuation factors used to modify the gradient domain image; the attenuation factors depend on the logarithmic luminance of the HDR

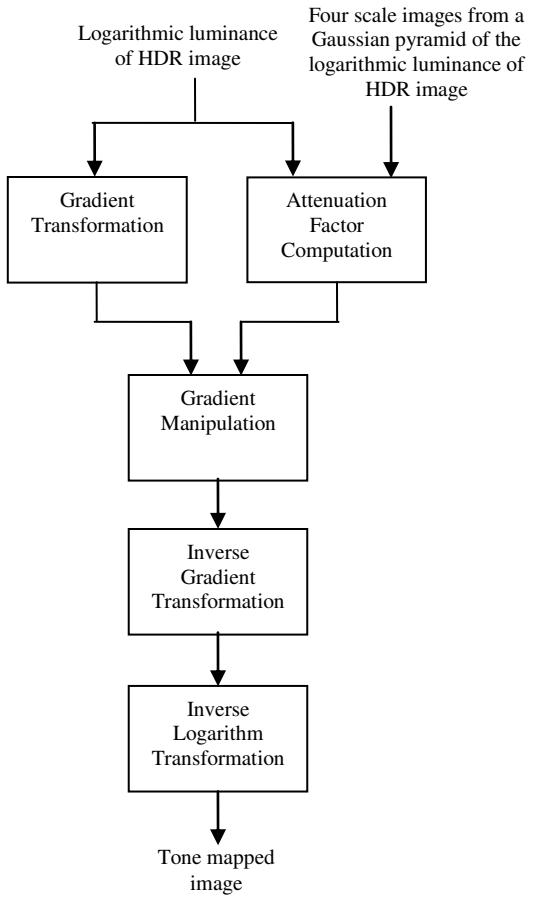


Figure 2.1 Block diagram of Fattal's operator.

image, and its four scale images. The gradients are multiplied with the attenuation factors to obtain manipulated gradients in the *gradient manipulation* block. Then, an *inverse gradient transformation* is done on the manipulated gradient domain image to return to the logarithmic luminance domain. An *inverse logarithm transformation* for the inverse gradients is done to obtain the tone mapped image in linear luminance domain, appropriate for display on a standard display device.

2.2.1 Gradient Transformation

Fattal's operator requires that the input image, which is in the logarithmic luminance domain, be transformed to the gradient domain. This is accomplished by use of a forward difference. In the gradient domain, each pixel is represented by two components, the negative gradients in the horizontal and vertical directions, respectively:

$$G^V_{i,j} = I^0_{i,j} - I^0_{i+1,j} \text{ and } G^H_{i,j} = I^0_{i,j} - I^0_{i,j+1}, \tag{2.2}$$

where I^0 is the logarithmic luminance of the input HDR image, G^V is the gradient in the vertical direction, G^H is the gradient in the horizontal direction, and i and j are indices of the pixels.

2.2.2 Attenuation Factor Computation

The attenuation factor for the gradient of a given pixel is calculated using information from a multi-scale Gaussian pyramid centered on the pixel. The work in [2] does not specify the number of scales; in this thesis, four scales are used. The original logarithmic luminance image is denoted by I^0, and the four scale images, from the finest scale with a standard deviation, or scale index, of 1 to the coarsest scale with a standard deviation, or scale index, of 4, are denoted by I^1, I^2, I^3, and I^4. For our implementation, each scale image is full resolution, with the same number of pixels as the original image; each pixel in a scale image is the Gaussian weighted average of a neighborhood of the pixel in the original image. The largest and coarsest scale image, at scale index 4, uses a 32×32 pixel Gaussian. While we do not report on hardware to compute the multi-scale

Gaussian pyramid in this work, previous work has shown that it can be effectively approximated in real time [20], and our results use this approximation.

The attenuation factor makes use of horizontal and vertical gradients at each scale index k that are computed using central differences:

$$H_{i,j}^{k} = \frac{I_{i,j-1}^{k} - I_{i,j+1}^{k}}{2} \quad \text{and} \quad V_{i,j}^{k} = \frac{I_{i-1,j}^{k} - I_{i+1,j}^{k}}{2}, \tag{2.3}$$

where k=0, 1, 2, 3, 4 and H^k, V^k are the central differences in the horizontal and vertical directions, respectively, at scale index k. At each scale index, the gradients are used to compute a scale attenuation factor based on the magnitude of the gradient at that scale index:

$$\varphi_{i,j}^{k} = \left(\frac{\left|H_{i,j}^{k}\right| + \left|V_{i,j}^{k}\right|}{a} \right)^{b-1}, \tag{2.4}$$

where a and b are parameters used to control the attenuation. The scale attenuation factor is designed to attenuate gradients of magnitude greater than a, and to magnify gradients of magnitude smaller than a. The parameter b is assumed to be less than 1. As shown in the equation, one-norms of the horizontal and vertical central differences are used for calculating the scale attenuation factors. A two-norm or an infinity-norm can also be used in place of the one-norm; our experiments show that they give similar results. The one-norm is used for ease of implementation.

The final attenuation factor $\Phi_{i,j}$ for a given pixel is obtained using all four scale indices to ensure that intensity transitions at every scale are taken into account:

$$\Phi_{i,j} = \prod_{k=0}^{4} \varphi_{i,j}^{k}. \tag{2.5}$$

This factor is applied to both the horizontal and vertical components of the gradient domain pixel.

2.2.3 Gradient Manipulation

Fattal's operator accomplishes tone mapping by manipulating the gradient domain version of the image. The manipulated gradient domain pixel is found by multiplying both the vertical and horizontal components of the original gradient domain pixel by the pixel's attenuation factor:

$$\hat{G}^V_{i,j} = \Phi_{i,j} \cdot G^V_{i,j} \quad \text{and} \quad \hat{G}^H_{i,j} = \Phi_{i,j} \cdot G^H_{i,j}, \tag{2.6}$$

where ˆ is used to denote the manipulated gradients. To avoid spatial distortions, only the magnitudes of the gradients are changed; the directions are unaltered.

2.2.4 Inverse Gradient Transformation

The next step in Fattal's operator is an inverse gradient transformation to bring the image, now of low dynamic range, back into the logarithmic luminance domain. Because the image has been manipulated in the gradient domain, it may be that no exact inverse image I^{out} exists. As a result, the objective of the inverse gradient transformation block is to compute the image I^{out} whose gradients most closely match the manipulated gradients in a least mean square sense. This transformation is a key to Fattal's operator, and is the subject of Chapter 3.

2.2.5 Inverse Logarithm Transformation

The logarithmic luminance output image produced by the inverse gradient transformation is converted to linear luminance by taking the inverse logarithm of each pixel. Note that at this point, we have a linear luminance (gray-scale) image.

The most computationally difficult part of Fattal's operator is the inverse gradient transformation which is used to obtain the tone-mapped image. Chapter 3 discusses the solution to the inverse gradient transformation and various approaches to obtain it. The preliminaries required to develop Fattal's operator in real-time are described in Chapter 4. The hardware for the real-time implementation of the full operator is then discussed, in Chapter 5.

CHAPTER III

SOLVING THE POISSON EQUATION LOCALLY

All algorithms that do processing in the gradient domain face a common problem of obtaining an inverse gradient transform once the original gradients have been manipulated. In a one-dimensional continuous case, an inverse gradient can be obtained by integrating the manipulated gradients along with an additional constant. However, in a two-dimensional case, the manipulated gradients may not be integrable [2]; there may not exist a solution I^{out} such that $\hat{G} = \nabla I^{\text{out}}$, where $\hat{G}$ denotes the manipulated gradient consisting of both the horizontal and vertical components and ∇ is the gradient operator. For a high dynamic range (HDR) compression of images, this means that there may not exist an image whose gradient exactly matches the manipulated gradient of the original image; that is, it may be that no exact solution to the inverse gradient transformation problem exists. This problem is resolved by employing a least mean squares technique to find the image whose gradient is closest to the manipulated gradients. In what follows, the solution to the inverse gradient transformation is derived in the continuous domain, and then discretized using finite differences for application to images.

3.1 Derivation of the Poisson Equation

Employing a least mean square approach, the output solution I^{out} must minimize the integral $\iint F(\nabla I^{\text{out}}, \hat{G})dxdy$, where

$$F(\nabla I^{\text{out}}, \hat{G}) = \left\| \nabla I^{\text{out}} - \hat{G} \right\|_2^2 = \left(\frac{\partial I^{\text{out}}}{\partial x} - \hat{G}^H \right)^2 + \left(\frac{\partial I^{\text{out}}}{\partial y} - \hat{G}^V \right)^2 . \tag{3.1}$$

Note that $\hat{G}$ has both the horizontal ($\hat{G}^H$) and vertical ($\hat{G}^V$) components in x and y directions respectively. According to the Variational Principle [2], the I^{out} that minimizes the integral of $F(\nabla I^{\text{out}}, \hat{G})$ must satisfy the Euler-Lagrange equation:

$$\frac{\partial F}{\partial I^{\text{out}}} - \frac{d}{dx}\frac{\partial F}{\partial I_x^{\text{out}}} - \frac{d}{dy}\frac{\partial F}{\partial I_y^{\text{out}}} = 0. \tag{3.2}$$

Solving equations (3.1), (3.2) and rearranging, we obtain the Poisson equation

$$\nabla^2 I^{\text{out}} = \text{div}\hat{G}, \tag{3.3}$$

where ∇^2 is the Laplacian operator and $\text{div}\hat{G}$ is the divergence of $\hat{G}$. The two sides of equation (3.3) are

$$\nabla^2 I^{\text{out}} = \frac{\partial^2 I^{\text{out}}}{\partial x^2} + \frac{\partial^2 I^{\text{out}}}{\partial y^2} \ , \tag{3.4}$$

$$\text{div}\hat{G} = \frac{\partial \hat{G}^H}{\partial x} + \frac{\partial \hat{G}^V}{\partial y}. \tag{3.5}$$

The Poisson equation (equation (3.3)) is generally used in gradient domain applications to implement the inverse gradient domain transform. The right hand side $\text{div}\hat{G}$ is known, and the equation is used to solve for I^{out}.

The Poisson equation in the case of images should be discretized using finite differences since images are represented in discrete form. The discrete Poisson equation

can be obtained by approximating the left hand side (equation (3.4)) using finite differences:

$$\nabla^2 I_{i,j}^{\text{out}} \approx I_{i-1,j}^{\text{out}} + I_{i+1,j}^{\text{out}} + I_{i,j-1}^{\text{out}} + I_{i,j+1}^{\text{out}} - 4I_{i,j}^{\text{out}} \tag{3.6}$$

and by approximating the right hand side of the Poisson equation (equation (3.5)) using backward differences:

$$\text{div}\hat{G}_{i,j} \approx \hat{G}_{i,j}^{H} - \hat{G}_{i-1,j}^{H} + \hat{G}_{i,j}^{V} - \hat{G}_{i,j-1}^{V}. \tag{3.7}$$

Note that the left hand side of the discrete Poisson equation can be obtained by passing a Laplacian window over the output image. The discrete Poisson equation can be rewritten in a matrix form that relates the pixels of I^{out} to the manipulated divergences:

$$L\overrightarrow{I^{\text{out}}} = \overrightarrow{\text{div}\hat{G}}, \tag{3.8}$$

where L is a Laplacian matrix whose coefficients are determined by the type of boundary conditions assumed when using the Laplacian operator. $\overrightarrow{I^{\text{out}}}$ is the image vector to be reconstructed and $\overrightarrow{\text{div}\hat{G}}$ is the vector consisting of the manipulated divergences. From here on in this thesis, the discrete Poisson equation is referred simply as the Poisson equation.

For Fattal's operator, the output image I^{out} is found by solving the Poisson equation (3.3). The Poisson equation is a traditional differential equation with boundary conditions. It is computationally intensive to solve because the only known boundaries are the borders of the image. This implies that solution of the Poisson equation for even a single pixel requires computations involving the entire image; in other words, although the gradient domain operator is local in nature, solving for the image with a particular

Table 3.1 Complexity of Poisson solver algorithms.

Algorithm	Time complexity for image size (N)	Type of Solver
Gaussian Elimination	N^3	Direct
Inverse(∇^2)$\times$div$\hat{G}$	N^2	Direct
Jacobi	N^2	Indirect
Conjugate Gradient (CG)	$N^{3/2}$	Indirect
Successive Over Relaxation (SOR)	$N^{3/2}$	Indirect
Multigrid	N	Indirect
FFT-based	$N\log N$	Direct
Proposed local Poisson solver	N	Direct

gradient requires global information. Because global information is needed, a traditional Poisson solver requires that a complete image frame be buffered. The rest of the chapter discusses a variety of Poisson solvers; the FFT-based solver is described in detail, as it is considered the gold standard in this thesis. Finally, the chapter introduces our proposed local Poisson solver.

3.2 Poisson Solvers

There exist several numerical methods to solve the Poisson equation. The Poisson solvers vary in their time complexity, memory usage and convergence or the amount of error in the output. The methods are categorized and described next.

The various methods for solving the Poisson equation can be categorized into direct solvers and indirect solvers. Direct solvers give an exact solution after a finite

number of steps. Indirect solvers are iterative and give an approximate solution; every iteration decreases the error of the solution by a constant factor.

Table 3.1 lists published Poisson solvers [29, 30], showing their time complexity for an image with a total of *N* pixels. Each solver is also categorized as either a direct or indirect solver. The Gaussian Elimination and the Inverse $(\nabla^2) \times \mathrm{div}\hat{G}$ methods are direct methods, but are very slow and require large memories. Jacobi, Conjugate Gradient (CG) and Successive Over Relaxation (SOR) methods are indirect methods having relatively large error and may also converge very slowly.

The fast Fourier transform (FFT) based method is also a well known technique for solving the Poisson equation; it requires order *N*log*N* operations, where *N* is the number of pixels in the image. The FFT-based solver is a direct approach giving an exact solution when such a solution exists. Implementing this method in real-time is not feasible as it is hardware-expensive and memory-intensive. Wang [23] has modified the FFT-based solver to make it implementable in real-time by solving on 8×8 tiled blocks of the image independently and reconstructing the full-size output image by tiling the processed 8×8 blocks; this approach fixes the hardware size, since regardless of the image size the blocks processed are 8×8. The shortcoming to this block-based approach is that the output images have blocking effects around the 8×8 blocks; i.e., there are visible tile lines in the output image.

The Multigrid algorithm [29], [30] is an indirect solver which requires order *N* operations to solve the Poisson equation approximately. This technique replaces the problem on a fine, pixel-by-pixel grid by an approximation on a coarser grid. This grid is further replaced by a coarser grid approximation recursively until the coarsest grid is

obtained. This coarsest grid is solved approximately and used as an initial solution for solving the next fine grid; the solution for one grid is used as the initial solution for the next grid so as to iteratively solve until the finest pixel-by-pixel grid is solved. A good approximation to the multigrid problem depends on the accuracy of the initial solution on the coarsest grid. This method is also not feasible for real-time implementation as the hardware cost is very high. It also requires more memory than the FFT-based solver [21], [22] as it is iterative in nature and there is information transferred between the grids.

Our proposed local Poisson solver is also a direct solver; it gives an exact solution to the Poisson equation when a solution exists for the inverse gradient transformation. The main advantage over other methods is that it is suitable for a real-time implementation; in particular, the hardware is simple and easy to implement. It also requires significantly less memory than other methods. It works on a 3×3 sliding window and gives a single pixel of I^{out} at a time. There are no blocking effects in the output image; this is because each application of the solver produces a single pixel as its output and these pixels form the full image. The results from the proposed local Poisson solver are compared with the results obtained using the FFT-based solver. The FFT-based solver is assumed to be the gold standard since it gives an exact solution when one exists. The FFT-based solver is described in the next section.

3.3 The FFT-Based Poisson Solver

The Poisson equation is a linear system that can be solved using an FFT-based solver. The left hand side of equation (3.3) is shown in equation (3.4) and can be

discretized using a one-dimensional Laplacian window [-1 2 -1] in each of the x and y directions. This approximation is:

$$\frac{\partial^2 I^{\text{out}}}{\partial x^2} \approx I^{\text{out}}_{j-1,k} - 2I^{\text{out}}_{j,k} + I^{\text{out}}_{j+1,k}, \tag{3.9}$$

$$\frac{\partial^2 I^{\text{out}}}{\partial y^2} \approx I^{\text{out}}_{j,k-1} - 2I^{\text{out}}_{j,k} + I^{\text{out}}_{j,k+1}. \tag{3.10}$$

The right hand side of equation (3.3) is discretized as shown in equation (3.7). The discrete Poisson equation can be written in the form of a matrix equation as:

$$TI^{\text{out}} + I^{\text{out}}T = \text{div}\hat{G}. \tag{3.11}$$

The size of each of the matrices in this equation is $n \times n$. The T matrix is a $n \times n$ symmetric tri-diagonal matrix with 2 along its main diagonal, -1 along the diagonal above and below the main diagonal, and zero elsewhere. This matrix can be factorized using an eigenvalue decomposition as $T = Q\lambda Q^{-1}$ where Q is a non-singular matrix consisting of the eigenvectors of T as its columns, and λ is a diagonal matrix of eigenvalues of T. The eigenvalues of λ are

$$\lambda_j = 2\left(1 - \cos\left(\frac{\pi j}{n+1}\right)\right) \tag{3.12}$$

and the elements of Q matrix consisting of the eigenvectors of T are obtained by

$$Q_{j,k} = \sqrt{\frac{2}{n+1}} \sin\left(\frac{j \cdot k \cdot \pi}{n+1}\right) = Q_{k,j}. \tag{3.13}$$

Substituting $T = Q\lambda Q^{-1}$ in equation (3.11), we get

$$Q\lambda Q^{-1} I^{\text{out}} + I^{\text{out}} Q\lambda Q^{-1} = \text{div}\hat{G}. \tag{3.14}$$

Pre-multiplying equation (3.14) with Q^{-1} and post-multiplying with Q, the equation reduces to

$$\{\lambda(Q^{-1}I^{\text{out}}Q)\}+\{(Q^{-1}I^{\text{out}}Q)\lambda\}=Q^{-1}\text{div}\hat{G}Q, \text{ or } \lambda\hat{I}+\hat{I}\lambda=\hat{H} \tag{3.15}$$

where $\hat{I}=Q^{-1}I^{\text{out}}Q$ and $\hat{H}=Q^{-1}\text{div}\hat{G}Q$. The Q matrix is orthogonal, which implies that $Q^{-1}=Q^{\text{T}}$. Also, $Q=Q^{\text{T}}$; thus, we have that $Q=Q^{-1}$.

The Poisson equation, $TI^{\text{out}}+I^{\text{out}}T=\text{div}\hat{G}$, now reduced to $\lambda\hat{I}+\hat{I}\lambda=\hat{H}$, can be solved as follows.

i) Compute $\hat{H}=Q^{-1}\text{div}\hat{G}Q$.

ii) Find $\hat{I}_{j,k}=\dfrac{\hat{H}_{j,k}}{\lambda_j+\lambda_k}$ for each j=0,1,…..n-1 and k=0,1,…..n-1.

iii) Calculate $I^{\text{out}}=Q\hat{I}Q^{-1}$.

These steps require two two-sided multiplications of a matrix by Q. We can observe from equation (3.13) that the Q matrix consists of sinusoidal terms that match those used in the discrete sine transform (DST). A fast Fourier transform (FFT) algorithm is generally used in solving the DST; the DST of a matrix can be derived from the imaginary part of the FFT of the same matrix. Details for this approach can be found in [30, 31]. Therefore, the computation of $\hat{H}$ can be done by taking the FFT of $\text{div}\hat{G}$ and extracting a part of the imaginary part of the result. The computation of I^{out} is done similarly.

The FFT-based solver works well as a Poisson solver but is not amenable to real-time embedded hardware implementation. In this thesis, it is used as a gold standard with

which to compare our proposed local Poisson solver. The next section introduces our local Poisson solver and the boundary condition assumed to obtain a good solution.

3.4 Our Proposed Local Poisson Solver

We want a technique that does not tile the image, so that there will not be blocking effects, and that works only locally, so that it can use fixed size hardware regardless of the size of the image. Taking into account the limitations of the FFT-based solver on a full image or 8×8 blocks, we propose a Poisson solver [3] that takes a 3×3 window of the image as input and solves for a single pixel at the center of the window. A pixel can be found using the values of the neighboring pixels around that pixel in an image. The proposed Poisson solver works on this principle, taking local information around the pixel to be found, and so we call it a *local Poisson solver*. The pixel at the center of each 3×3 window is obtained while sliding the window over the entire manipulated gradient domain image, thereby solving for all of the pixels in the output image.

This proposed method for solving the Poisson equation is fundamentally different from traditional solvers; it requires a different divergence calculation. In traditional solvers, divergence is calculated assuming the entire image, with boundary conditions only at the image borders; one divergence is calculated for each pixel in the image, and all of the divergences are needed in order to solve the Poisson equation. Here, divergence is calculated assuming a 3×3 window around the pixel, with a certain boundary condition beyond that window. The trade-off is that this specially-defined divergence must be calculated not only for the pixel itself, but also for its eight spatial neighbors, even to find

a single pixel of the output image. Thus, the proposed method does nine times as many divergence calculations as the traditional solver, but the redundancy is such that computations can be done locally and in parallel, with simple hardware and no need to buffer an entire frame of data.

The specially-defined nine gradient divergences needed to solve for output image pixel $I_{i,j}^{\text{out}}$ are represented by $\text{div}\mathcal{G}_{p,q}^{i,j}$, where $p = 1, 2, 3$ and $q = 1, 2, 3$. These specially-defined divergences in a 3×3 neighborhood around a given pixel can be computed using backward differences as:

$$\text{div}\mathcal{G}_{p,q}^{i,j} = \mathcal{G}_{p,q}^{H_{i,j}} - \mathcal{G}_{p-1,q}^{H_{i,j}} + \mathcal{G}_{p,q}^{V_{i,j}} - \mathcal{G}_{p,q-1}^{V_{i,j}}, \tag{3.16}$$

where $\mathcal{G}^{H_{i,j}}$ and $\mathcal{G}^{V_{i,j}}$ are gradients in horizontal and vertical directions that are computed using a 3×3 sliding window from the input image and extending it with boundaries to a 5×5 window. The 3×3 window from the input image along with general boundaries is

$$\begin{bmatrix} b_1 & b_2 & b_3 & b_4 & b_5 \\ b_6 & I_{i-1,j-1}^0 & I_{i-1,j}^0 & I_{i-1,j+1}^0 & b_7 \\ b_8 & I_{i,j-1}^0 & I_{i,j}^0 & I_{i,j+1}^0 & b_9 \\ b_{10} & I_{i+1,j-1}^0 & I_{i+1,j}^0 & I_{i+1,j+1}^0 & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \end{bmatrix},$$

where the b_k values are the (yet to be determined) pixels outside the window that set the boundary conditions. For convenience in notation, this 5×5 window is denoted as

$$\begin{bmatrix} \mathcal{F}_{0,0}^{i,j} & \mathcal{F}_{0,1}^{i,j} & \mathcal{F}_{0,2}^{i,j} & \mathcal{F}_{0,3}^{i,j} & \mathcal{F}_{0,4}^{i,j} \\ \mathcal{F}_{1,0}^{i,j} & \mathcal{F}_{1,1}^{i,j} & \mathcal{F}_{1,2}^{i,j} & \mathcal{F}_{1,3}^{i,j} & \mathcal{F}_{1,4}^{i,j} \\ \mathcal{F}_{2,0}^{i,j} & \mathcal{F}_{2,1}^{i,j} & \mathcal{F}_{2,2}^{i,j} & \mathcal{F}_{2,3}^{i,j} & \mathcal{F}_{2,4}^{i,j} \\ \mathcal{F}_{3,0}^{i,j} & \mathcal{F}_{3,1}^{i,j} & \mathcal{F}_{3,2}^{i,j} & \mathcal{F}_{3,3}^{i,j} & \mathcal{F}_{3,4}^{i,j} \\ \mathcal{F}_{4,0}^{i,j} & \mathcal{F}_{4,1}^{i,j} & \mathcal{F}_{4,2}^{i,j} & \mathcal{F}_{4,3}^{i,j} & \mathcal{F}_{4,4}^{i,j} \end{bmatrix}.$$

The gradients in horizontal and vertical directions are computed using negative forward differences as follows:

$$\mathcal{G}_{u,v}^{V_{i,j}} = \mathcal{F}_{u,v}^{i,j} - \mathcal{F}_{u+1,v}^{i,j} \text{ and } \mathcal{G}_{u,v}^{H_{i,j}} = \mathcal{F}_{u,v}^{i,j} - \mathcal{F}_{u,v+1}^{i,j}. \tag{3.17}$$

where u and v vary from 0 to 3. The divergence of the corresponding 3×3 window is

$$\begin{bmatrix} \text{div}\mathcal{G}_{1,1}^{i,j} & \text{div}\mathcal{G}_{1,2}^{i,j} & \text{div}\mathcal{G}_{1,3}^{i,j} \\ \text{div}\mathcal{G}_{2,1}^{i,j} & \text{div}\mathcal{G}_{2,2}^{i,j} & \text{div}\mathcal{G}_{2,3}^{i,j} \\ \text{div}\mathcal{G}_{3,1}^{i,j} & \text{div}\mathcal{G}_{3,2}^{i,j} & \text{div}\mathcal{G}_{3,3}^{i,j} \end{bmatrix}.$$

Note that the gradients in horizontal and vertical directions are manipulated to obtain tone mapping in our actual implementation. As a result, the divergences used by the Poisson solver would also be manipulated divergences. The manipulated gradients or divergences are denoted with a '^' symbol. In what follows in this chapter, we develop our local Poisson solver using notation that shows unmanipulated gradients and divergences. However, manipulated gradients and divergences may be substituted for the unmanipulated ones.

To solve the Poisson equation, each of the pixels in the 3×3 window is passed through a Laplacian operator which serves to take a moving average in the window. The two-dimensional kernel of the Laplacian operator used for this purpose is

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

The operator is passed over each pixel in the window in turn, centered over the pixel, so that a weighted average is computed for each pixel. The neighbors of each of the pixels along the boundary of the 3×3 window are needed as the Laplacian operator is passed over the window. Therefore, a suitable boundary condition is required for the 3×3 window for solving the local Poisson equation. Note that we use the same boundary condition for calculating the divergences and also for solving the Poisson equation. The 3×3 window obtained by solving the Poisson equation is

$$\begin{bmatrix} \bar{I}_{1,1}^{i,j} & \bar{I}_{1,2}^{i,j} & \bar{I}_{1,3}^{i,j} \\ \bar{I}_{2,1}^{i,j} & \bar{I}_{2,2}^{i,j} & \bar{I}_{2,3}^{i,j} \\ \bar{I}_{3,1}^{i,j} & \bar{I}_{3,2}^{i,j} & \bar{I}_{3,3}^{i,j} \end{bmatrix}.$$

The pixel at the center of the 3×3 window, $\bar{I}_{2,2}^{i,j}$, is the output of the local Poisson solver; that is, it is the pixel $I_{i,j}^{\text{out}}$ in the output image. The remaining pixels in this 3×3 window are thrown away.

The matrix equation for the local Poisson solver on a 3×3 window can also be written in the vector form $L\overrightarrow{\bar{I}^{i,j}} = \overrightarrow{\text{div}\mathcal{G}^{i,j}}$, as follows:

$$\begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} \bar{I}_{1,1}^{i,j} \\ \bar{I}_{1,2}^{i,j} \\ \bar{I}_{1,3}^{i,j} \\ \bar{I}_{2,1}^{i,j} \\ \bar{I}_{2,2}^{i,j} \\ \bar{I}_{2,3}^{i,j} \\ \bar{I}_{3,1}^{i,j} \\ \bar{I}_{3,2}^{i,j} \\ \bar{I}_{3,3}^{i,j} \end{bmatrix} = \begin{bmatrix} \mathrm{div}\mathcal{G}_{1,1}^{i,j} + b_2 + b_6 \\ \mathrm{div}\mathcal{G}_{1,2}^{i,j} + b_3 \\ \mathrm{div}\mathcal{G}_{1,3}^{i,j} + b_4 + b_7 \\ \mathrm{div}\mathcal{G}_{2,1}^{i,j} + b_8 \\ \mathrm{div}\mathcal{G}_{2,2}^{i,j} \\ \mathrm{div}\mathcal{G}_{2,3}^{i,j} + b_9 \\ \mathrm{div}\mathcal{G}_{3,1}^{i,j} + b_{10} + b_{13} \\ \mathrm{div}\mathcal{G}_{3,2}^{i,j} + b_{14} \\ \mathrm{div}\mathcal{G}_{3,3}^{i,j} + b_{11} + b_{15} \end{bmatrix}. \quad (3.18)$$

Boundary conditions play an important role in determining the solution to the Poisson equation. We consider three types of boundary conditions.

i) Zero Dirichlet boundary condition. The pixels outside of the 3×3 window are assumed to be zeros.

ii) Neumann boundary condition. The boundaries have zero gradient values. This means that the pixels outside the window are symmetric extensions of the pixels in the window.

iii) Semi-implicit boundary condition. A combination of boundary conditions is used. The pixel values b_1 to b_8 in equation (3.18) use a zero Dirichlet boundary condition in the full-size output image and b_9 to b_{16} use a Neumann boundary condition in the 3×3 window. Note that if an image is processed in row major order starting in the top left corner, pixels b_1 to b_8 correspond to pixels in the output image that have already been solved when the local Poisson solver is applied to pixel at i, j in the full-size image. Thus, it is possible to fill in each of the pixels b_1 to b_8 with either a zero if the pixel

corresponds to a boundary of the original full-size image, or with an already calculated output pixel if the pixel corresponds to a location within the interior of the original full-size image. Pixels b_9 to b_{16} have not been previously solved, and are filled in using symmetric extension of the pixels in the 3×3 window.

The three choices for setting the boundary conditions for the local Poisson solver are described in more detail next. A 3×3 window, extended to 5×5 window using each of the three considered types of boundary conditions, is presented in each case. The boundary values that are known, either as zeros or as previously calculated pixel values, are represented in red. The unknown boundary values are represented in green; these are values that are symmetrically extended. For the semi-implicit boundary condition, the known boundary values are zeros if the window is at the border of the full-size image, and previously calculated output pixels otherwise. The corresponding Poisson equation in matrix form, rearranged where required, for each boundary condition choice is also presented.

Zero Dirichlet boundary condition

The 3×3 window from the input image extended with zeros at the boundaries, used for calculating the divergences, is

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & I^0_{i-1,j-1} & I^0_{i-1,j} & I^0_{i-1,j+1} & 0 \\ 0 & I^0_{i,j-1} & I^0_{i,j} & I^0_{i,j+1} & 0 \\ 0 & I^0_{i+1,j-1} & I^0_{i+1,j} & I^0_{i+1,j+1} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The output 3×3 window used for solving the Poisson equation uses the same boundaries as the input window and is represented as

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \bar{I}_{1,1}^{i,j} & \bar{I}_{1,2}^{i,j} & \bar{I}_{1,3}^{i,j} & 0 \\ 0 & \bar{I}_{2,1}^{i,j} & \bar{I}_{2,2}^{i,j} & \bar{I}_{2,3}^{i,j} & 0 \\ 0 & \bar{I}_{3,1}^{i,j} & \bar{I}_{3,2}^{i,j} & \bar{I}_{3,3}^{i,j} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Note that the boundaries are represented in red. The resulting Poisson equation in matrix form is written as

$$\begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} \bar{I}_{1,1}^{i,j} \\ \bar{I}_{1,2}^{i,j} \\ \bar{I}_{1,3}^{i,j} \\ \bar{I}_{2,1}^{i,j} \\ \bar{I}_{2,2}^{i,j} \\ \bar{I}_{2,3}^{i,j} \\ \bar{I}_{3,1}^{i,j} \\ \bar{I}_{3,2}^{i,j} \\ \bar{I}_{3,3}^{i,j} \end{bmatrix} = \begin{bmatrix} \mathrm{div}\mathcal{G}_{1,1}^{i,j} \\ \mathrm{div}\mathcal{G}_{1,2}^{i,j} \\ \mathrm{div}\mathcal{G}_{1,3}^{i,j} \\ \mathrm{div}\mathcal{G}_{2,1}^{i,j} \\ \mathrm{div}\mathcal{G}_{2,2}^{i,j} \\ \mathrm{div}\mathcal{G}_{2,3}^{i,j} \\ \mathrm{div}\mathcal{G}_{3,1}^{i,j} \\ \mathrm{div}\mathcal{G}_{3,2}^{i,j} \\ \mathrm{div}\mathcal{G}_{3,3}^{i,j} \end{bmatrix}. \tag{3.19}$$

The matrix form for the Poisson equation using the zero Dirichlet boundary condition shows the computations needed when applying the Poisson solver. This matrix equation is used to obtain the center pixel of the 3×3 window. The inverse of the matrix equation yields the relationship:

$$\begin{aligned} I_{i,j}^{\mathrm{out}} = \bar{I}_{2,2}^{i,j} &= \frac{1}{16}(\mathrm{div}\mathcal{G}_{1,1}^{i,j} + \mathrm{div}\mathcal{G}_{3,1}^{i,j} + \mathrm{div}\mathcal{G}_{1,3}^{i,j} + \mathrm{div}\mathcal{G}_{3,3}^{i,j}) \\ &+ \frac{1}{8}(\mathrm{div}\mathcal{G}_{2,1}^{i,j} + \mathrm{div}\mathcal{G}_{2,3}^{i,j} + \mathrm{div}\mathcal{G}_{1,2}^{i,j} + \mathrm{div}\mathcal{G}_{3,2}^{i,j}) + \frac{3}{8}\mathrm{div}\mathcal{G}_{2,2}^{i,j}. \end{aligned} \tag{3.20}$$

Note that $I_{i,j}^{\text{out}}$ is the output pixel and equal to $\overline{I}_{2,2}^{i,j}$; this is the center pixel of the output 3×3 window where i, j is the index for the pixel in the output image. This equation is simple to implement in hardware. Because the coefficients have denominators that are all powers-of-two, the multiplications can be easily implemented with just shifts and additions. Division by a power-of-two is accomplished by shifting a variable bit-wise to the right. The 3 in the numerator is also easily implemented using only shifts and additions; multiplying a number by three is equivalent to adding the number to a version of itself that has been shifted one bit to the left.

Neumann boundary condition

The 3×3 window from the input image symmetrically extended at the boundaries, used for calculating the divergences, is

$$\begin{bmatrix} I_{i-1,j-1}^0 & I_{i-1,j-1}^0 & I_{i-1,j}^0 & I_{i-1,j+1}^0 & I_{i-1,j+1}^0 \\ I_{i-1,j-1}^0 & I_{i-1,j-1}^0 & I_{i-1,j}^0 & I_{i-1,j+1}^0 & I_{i-1,j+1}^0 \\ I_{i,j-1}^0 & I_{i,j-1}^0 & I_{i,j}^0 & I_{i,j+1}^0 & I_{i,j+1}^0 \\ I_{i+1,j-1}^0 & I_{i+1,j-1}^0 & I_{i+1,j}^0 & I_{i+1,j+1}^0 & I_{i+1,j+1}^0 \\ I_{i+1,j-1}^0 & I_{i+1,j-1}^0 & I_{i+1,j}^0 & I_{i+1,j+1}^0 & I_{i+1,j+1}^0 \end{bmatrix}.$$

The output 3×3 window used for solving the Poisson equation also uses symmetric extension and is represented as

$$\begin{bmatrix} \overline{I}_{1,1}^{i,j} & \overline{I}_{1,1}^{i,j} & \overline{I}_{1,2}^{i,j} & \overline{I}_{1,3}^{i,j} & \overline{I}_{1,3}^{i,j} \\ \overline{I}_{1,1}^{i,j} & \overline{I}_{1,1}^{i,j} & \overline{I}_{1,2}^{i,j} & \overline{I}_{1,3}^{i,j} & \overline{I}_{1,3}^{i,j} \\ \overline{I}_{2,1}^{i,j} & \overline{I}_{2,1}^{i,j} & \overline{I}_{2,2}^{i,j} & \overline{I}_{2,3}^{i,j} & \overline{I}_{2,3}^{i,j} \\ \overline{I}_{3,1}^{i,j} & \overline{I}_{3,1}^{i,j} & \overline{I}_{3,2}^{i,j} & \overline{I}_{3,3}^{i,j} & \overline{I}_{3,3}^{i,j} \\ \overline{I}_{3,1}^{i,j} & \overline{I}_{3,1}^{i,j} & \overline{I}_{3,2}^{i,j} & \overline{I}_{3,3}^{i,j} & \overline{I}_{3,3}^{i,j} \end{bmatrix}.$$

Note that the boundaries are represented in green. The resulting Poisson equation in matrix form is written as

$$\begin{bmatrix} 2 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 3 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} \bar{I}_{1,1}^{i,j} \\ \bar{I}_{1,2}^{i,j} \\ \bar{I}_{1,3}^{i,j} \\ \bar{I}_{2,1}^{i,j} \\ \bar{I}_{2,2}^{i,j} \\ \bar{I}_{2,3}^{i,j} \\ \bar{I}_{3,1}^{i,j} \\ \bar{I}_{3,2}^{i,j} \\ \bar{I}_{3,3}^{i,j} \end{bmatrix} = \begin{bmatrix} \mathrm{div}\mathcal{G}_{1,1}^{i,j} \\ \mathrm{div}\mathcal{G}_{1,2}^{i,j} \\ \mathrm{div}\mathcal{G}_{1,3}^{i,j} \\ \mathrm{div}\mathcal{G}_{2,1}^{i,j} \\ \mathrm{div}\mathcal{G}_{2,2}^{i,j} \\ \mathrm{div}\mathcal{G}_{2,3}^{i,j} \\ \mathrm{div}\mathcal{G}_{3,1}^{i,j} \\ \mathrm{div}\mathcal{G}_{3,2}^{i,j} \\ \mathrm{div}\mathcal{G}_{3,3}^{i,j} \end{bmatrix}. \tag{3.21}$$

When the Neumann boundary condition is used, the corresponding L matrix is singular and the equation does not have a unique solution; for this reason, Neumann boundary condition cannot be used for our hardware approach and will not be considered further.

Semi-implicit boundary condition

The 3×3 window from the input image using a semi-implicit boundary condition is represented as

$$\begin{bmatrix} I_{i-2,j-2}^{\text{out}} & I_{i-2,j-1}^{\text{out}} & I_{i-2,j}^{\text{out}} & I_{i-2,j+1}^{\text{out}} & I_{i-2,j+2}^{\text{out}} \\ I_{i-1,j-2}^{\text{out}} & I_{i-1,j-1}^{0} & I_{i-1,j}^{0} & I_{i-1,j+1}^{0} & I_{i-1,j+2}^{\text{out}} \\ I_{i,j-2}^{\text{out}} & I_{i,j-1}^{0} & I_{i,j}^{0} & I_{i,j+1}^{0} & I_{i,j+1}^{0} \\ I_{i+1,j-1}^{0} & I_{i+1,j-1}^{0} & I_{i+1,j}^{0} & I_{i+1,j+1}^{0} & I_{i+1,j+1}^{0} \\ I_{i+1,j-1}^{0} & I_{i+1,j-1}^{0} & I_{i+1,j}^{0} & I_{i+1,j+1}^{0} & I_{i+1,j+1}^{0} \end{bmatrix}.$$

The output 3×3 window used for solving the Poisson equation also uses a semi-implicit boundary condition and is represented as

$$\begin{bmatrix} I^{out}_{i-2,j-2} & I^{out}_{i-2,j-1} & I^{out}_{i-2,j} & I^{out}_{i-2,j+1} & I^{out}_{i-2,j+2} \\ I^{out}_{i-1,j-2} & \bar{I}^{i,j}_{1,1} & \bar{I}^{i,j}_{1,2} & \bar{I}^{i,j}_{1,3} & I^{out}_{i-1,j+2} \\ I^{out}_{i,j-2} & \bar{I}^{i,j}_{2,1} & \bar{I}^{i,j}_{2,2} & \bar{I}^{i,j}_{2,3} & \bar{I}^{i,j}_{2,3} \\ \bar{I}^{i,j}_{3,1} & \bar{I}^{i,j}_{3,1} & \bar{I}^{i,j}_{3,2} & \bar{I}^{i,j}_{3,3} & \bar{I}^{i,j}_{3,3} \\ \bar{I}^{i,j}_{3,1} & \bar{I}^{i,j}_{3,1} & \bar{I}^{i,j}_{3,2} & \bar{I}^{i,j}_{3,3} & \bar{I}^{i,j}_{3,3} \end{bmatrix}.$$

Note that the previously calculated pixels are represented in red and symmetric extension in green. The resulting Poisson equation in matrix form is written as

$$\begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} \bar{I}^{i,j}_{1,1} \\ \bar{I}^{i,j}_{1,2} \\ \bar{I}^{i,j}_{1,3} \\ \bar{I}^{i,j}_{2,1} \\ \bar{I}^{i,j}_{2,2} \\ \bar{I}^{i,j}_{2,3} \\ \bar{I}^{i,j}_{3,1} \\ \bar{I}^{i,j}_{3,2} \\ \bar{I}^{i,j}_{3,3} \end{bmatrix} = \begin{bmatrix} \mathrm{div}\mathcal{G}^{i,j}_{1,1} + I^{out}_{i-2,j-1} + I^{out}_{i-1,j-2} \\ \mathrm{div}\mathcal{G}^{i,j}_{1,2} + I^{out}_{i-2,j} \\ \mathrm{div}\mathcal{G}^{i,j}_{1,3} + I^{out}_{i-2,j+1} + I^{out}_{i-1,j+2} \\ \mathrm{div}\mathcal{G}^{i,j}_{2,1} + I^{out}_{i,j-2} \\ \mathrm{div}\mathcal{G}^{i,j}_{2,2} \\ \mathrm{div}\mathcal{G}^{i,j}_{2,3} \\ \mathrm{div}\mathcal{G}^{i,j}_{3,1} \\ \mathrm{div}\mathcal{G}^{i,j}_{3,2} \\ \mathrm{div}\mathcal{G}^{i,j}_{3,3} \end{bmatrix}. \tag{3.22}$$

The matrix form for the Poisson equation using the semi-implicit boundary condition is used to obtain the center pixel of the 3×3 window. The inverse of the matrix equation yields the relationship:

$$\begin{aligned} I^{out}_{i,j} = \bar{I}^{i,j}_{2,2} = {} & 0.1177(\mathrm{div}\mathcal{G}^{i,j}_{1,1} + I^{out}_{i-2,j-1} + I^{out}_{i-1,j-2}) + 0.2096(\mathrm{div}\mathcal{G}^{i,j}_{1,2} + I^{out}_{i-2,j}) \\ & +0.1468(\mathrm{div}\mathcal{G}^{i,j}_{1,3} + I^{out}_{i-2,j+1} + I^{out}_{i-1,j+2}) + 0.2614(\mathrm{div}\mathcal{G}^{i,j}_{2,1} + I^{out}_{i,j-2}) \\ & +0.5738(\mathrm{div}\mathcal{G}^{i,j}_{2,2}) + 0.3776(\mathrm{div}\mathcal{G}^{i,j}_{2,3}) + 0.3540(\mathrm{div}\mathcal{G}^{i,j}_{3,1}) \\ & +0.4466(\mathrm{div}\mathcal{G}^{i,j}_{3,2}) + 0.4121(\mathrm{div}\mathcal{G}^{i,j}_{3,3}). \end{aligned} \tag{3.23}$$

We can observe from equation (3.23) that the coefficients are not simple to implement in hardware and we need to pass information from previously calculated output pixels.

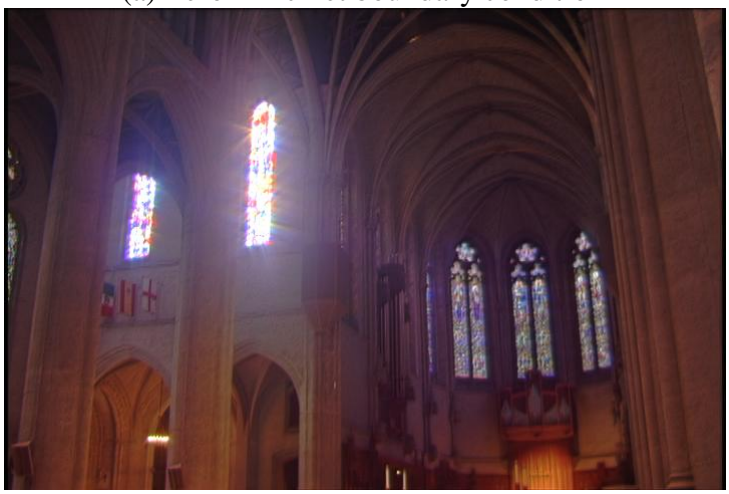
A simulation experiment was conducted using MATLAB to assess which choice of the boundary conditions that can be implemented in hardware gives the best result. The Nave image from the Debevec library [9], shown in Figure 3.1, was used as a test case for this experiment. Figure 3.1 was displayed by linearly mapping the pixels between 0 and 255, i.e., without tone mapping. Fattal's operator using a local Poisson solving approach with a 3×3 window was simulated, based on Poisson equations (3.20) and (3.23) obtained using the zero Dirichlet and semi-implicit boundary condition choices, respectively. The output images produced are in color. The conversion required to produce a linear red output pixel R^{out} of the tone-mapped output image is:

$$R^{\text{out}} = \left(\frac{R^{\text{in}}}{L^{\text{in}}} \right)^{s} L^{\text{out}}, \tag{3.24}$$

where R^{in} and L^{in} are the linear red and luminance components of the input HDR image, L^{out} is the linear luminance component of the tone-mapped image. The exponent s is color saturation factor; a value between 0.4 and 0.6 gives satisfactory results. The linear green

Figure 3.1 The original Nave image.

(a) Zero Dirichlet boundary condition



(b) Semi-implicit boundary condition

Figure 3.2 Output images obtained using zero Dirichlet and semi-implicit boundary conditions on the Nave image.

and blue pixels G^{out} and B^{out} of the tone-mapped color output image are computed similarly.

The output images produced using the zero Dirichlet and semi-implicit boundary condition choices are shown in Figure 3.2. The zero Dirichlet boundary condition works well; detail in the output image is clear. The semi-implicit boundary condition also gives a good quality image but the details on the windows are clear using zero Dirichlet

boundary condition. The zero Dirichlet boundary condition is chosen for our proposed local Poisson solver; this is because the hardware implementation is simpler, with no need to pass information from previously calculated output pixels. Also, we can observe that the coefficients in our local Poisson solver using zero Dirichlet boundary condition are powers-of-two which can be implemented using shift operations.

A Poisson solver produces the image whose gradients are closest to the input manipulated gradient domain image in a least squares sense, thereby doing a kind of inverse gradient transform. Note that if the input were a gradient domain image whose gradients had not been manipulated, the inverse gradient transformation would have an exact solution, and the Poisson equation would give a perfect reconstruction of the image. Both the FFT-based solver and the proposed local Poisson solver using zero Dirichlet boundary condition work successfully in obtaining an inverse gradient transformation in the sense that they give a perfect reconstruction of the image when the input gradient domain image is not manipulated.

Thus a local Poisson solver is proposed and described in detail that has significant advantages over traditional solvers for real-time applications. It has a fixed hardware size irrespective of the size of the image, requires less memory than traditional solvers, and is easier to implement in real-time hardware. The next chapter discusses preliminaries that are required to develop a real-time implementation of Fattal's operator.

CHAPTER IV

PRELIMINARIES TO DEVELOPMENT OF A REAL-TIME IMPLEMENTATION OF FATTAL'S OPERATOR

Our aim of the thesis is to implement Fattal's operator in real time using our local Poisson solver. Both Fattal's operator and our local Poisson solver were discussed in the previous chapters. Before a real-time implementation of Fattal's operator can be designed, it is essential to decide on values for the various parameters that form a part of the operator. Fattal's operator involves an attenuation factor calculation that requires two attenuation parameters, *a* and *b*. Here, simulation experiments are done to determine appropriate values for the attenuation parameters. Fattal's operator calculates the attenuation factor using a multi-resolution edge detection scheme to identify all the significant intensity transitions. Multi-level Gaussian pyramids are used for this purpose. In this chapter, we discuss the implementation of the Gaussian pyramids.

4.1 Attenuation Parameters

As described in Chapter 2, an attenuation factor is required to tone map the high dynamic range (HDR) image. Tone mapping is done by multiplying each pixel by an attenuation factor calculated from the one-norm of the central differences around that pixel in both the vertical and horizontal directions, $\mathscr{H}_{u,v}^{k_{i,j}}$ and $\mathscr{V}_{u,v}^{k_{i,j}}$, respectively. The full

calculation, which is given in equation (2.4), is rewritten for our 3×3 window approach as:

$$\zeta_{u,v}^{k_{i,j}} = \left(\frac{\left| \mathscr{H}_{u,v}^{k_{i,j}} \right| + \left| \mathscr{V}_{u,v}^{k_{i,j}} \right|}{a} \right)^{b-1}, \tag{4.1}$$

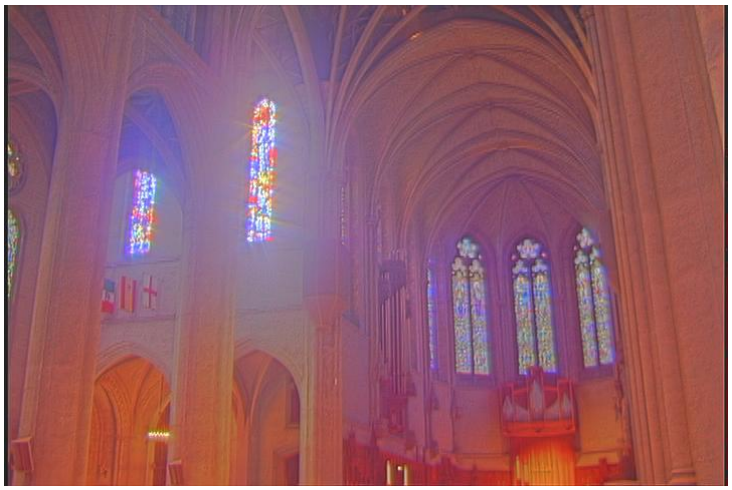
and involves two attenuation parameters a and b, that determine the amount by which the HDR pixels should be compressed. Note that the indices u and v vary from 0 to 3 within the 3×3 window along with its boundaries and the index i, j indicates the pixel being solved in the output image. Details of our approach are discussed in detail in Chapter 5. These attenuation factors are calculated for each of the pixels in each of the 3×3 windows, extended with zero Dirichlet boundary condition, of the input HDR image and its four scale images of the Gaussian pyramid. Gradients of larger magnitude than a are attenuated and those smaller than a are magnified. The value of parameter b is assumed to be less than 1.

To get a satisfactory low dynamic range compressed image, the parameters a and b should be chosen wisely. One way to decide these parameters is to look at a histogram of pixels of tone-mapped output image. A histogram is a graphical representation of the distribution of pixel intensities of an image. A good quality image has a histogram with pixel values spread throughout the intensity range in a Gaussian-like distribution that spreads widely from the center towards the edges of the intensity range. An experiment was carried out using MATLAB to study the effect of the attenuation parameters on the quality of a tone-mapped test image. One attenuation parameter is varied while keeping the other parameter fixed, and vice-versa. The effect of the varying parameter on the

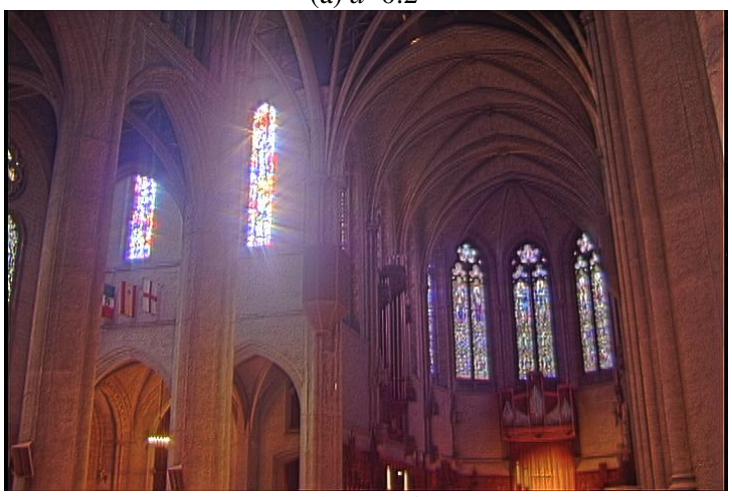
output image is studied qualitatively. Later the histogram of the pixels of the output image produced using the chosen values of *a* and *b* is shown.

An experiment was setup to implement Fattal's operator using our local Poisson solver and simulated using floating-point mathematics in MATLAB to produce a tone-mapped output image. The Nave image from the Debevec library [9], previously shown in Figure 3.1, is chosen for this experiment. The effect of the attenuation parameter *a* on the quality of the output image was investigated by fixing *b* to 0.9 and setting *a* to different values. The output images are obtained in color as discussed in Chapter 3.

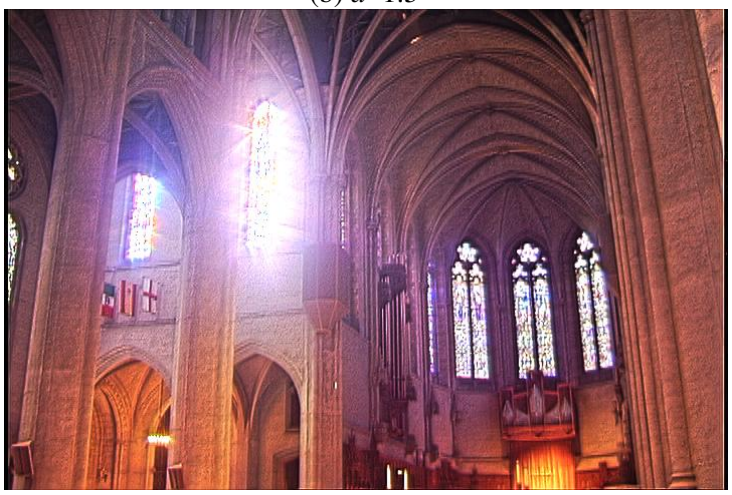
When the output images for different values of *a* were observed, it is evident that higher values of *a* gave higher contrast and lower values of *a* gave lower contrast. To get a good contrast in the output image, *a* should be chosen appropriately. Higher values of *a* could lead to saturation of pixels, with loss of detail in the bright parts of the image. Lower values of *a* deteriorate the contrast of the image. The output images with the value of *a* set to 0.2, 1.5, and 2.5, respectively, are shown in Figure 4.1. For a=0.2, the output image has a low contrast; as a result, the image does not look realistic. For a=2.5, the windows on the left side of the image are saturated and the details cannot be seen. The windows on the right side also start to wash out. The Nave image with a=1.5 has a good contrast and tone maps the dynamic range well without losing detail in the windows on either side of the image. As a result, a=1.5 is chosen for our implementation. Note that a wide range of values for *a* are tested; only a few sample images are included here. In addition, *a* was varied while fixing *b* to values other than 0.9, and the other images in the Debevec library were also tested. Overall, an *a* value of 1.5 gave good quality tone-mapped output images.



(a) a=0.2

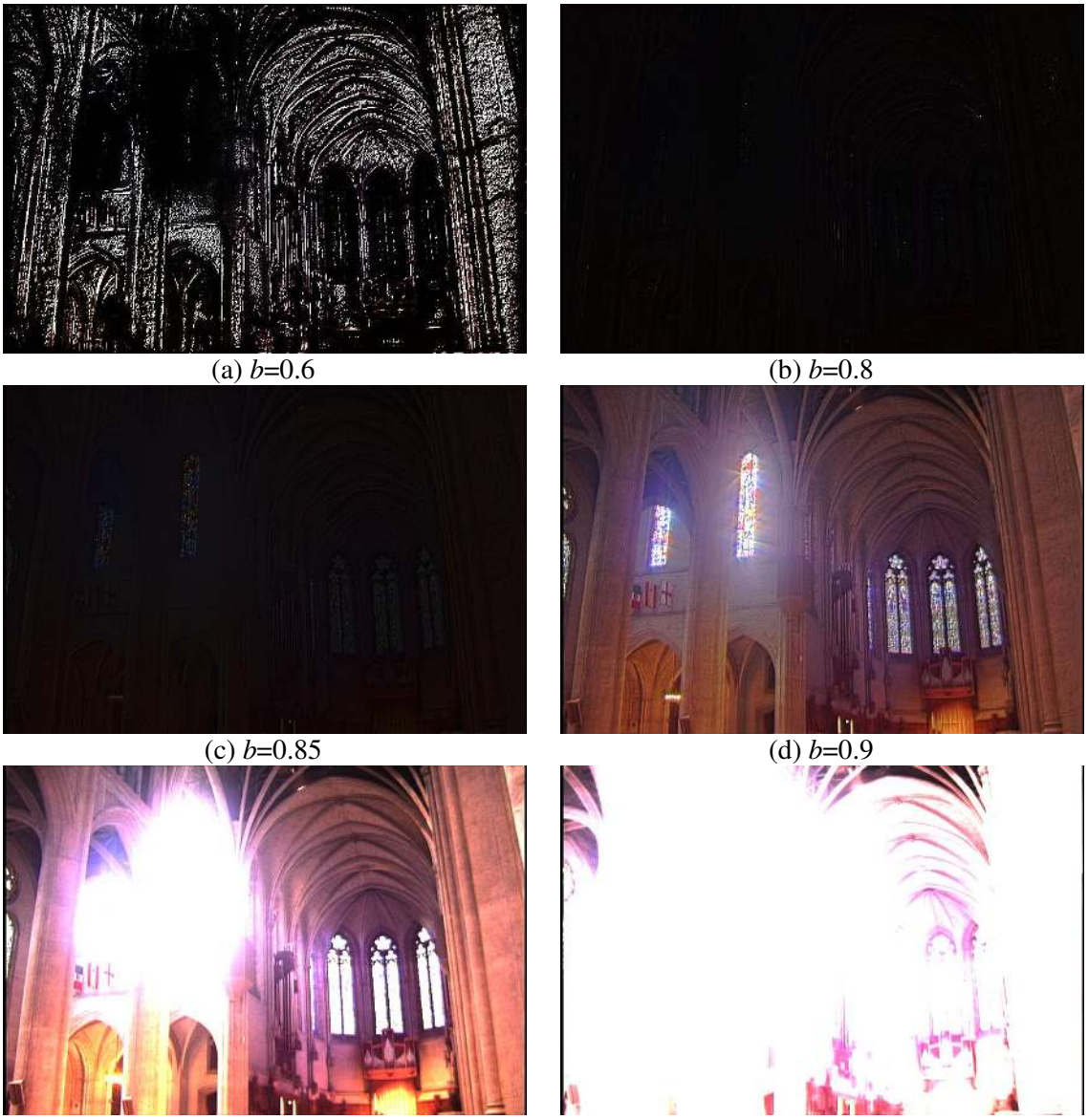


(b) a=1.5



(c) a=2.5

Figure 4.1 Experiment showing the effect of the parameter a on the contrast of the Nave image.



(a) b=0.6 (b) b=0.8

(c) b=0.85 (d) b=0.9

(e) b=0.95 (f) b=1.0

Figure 4.2 Experiment showing the effect of the parameter b on the tone mapping of the dynamic range of the Nave image.

Now that we have chosen a value for a, we fix a to 1.5 and vary the value of b to study its effect on the quality of the Nave test image. Fattal's operator using our local Poisson solver is again simulated using floating-point mathematics in MATLAB. Fattal suggests a b value between 0.8 and 0.9 in his original paper [2]. However, we considered a wider range of b values.

As the value of *b* is decreased, the tone mapping algorithm produces poor quality and strange output images. Higher values of *b* do not tone map the image properly; detail in dark and bright regions are lost. The output images produced using values of *b* set to 0.6, 0.8, 0.85, 0.9, 0.95, and 1.0 are shown in Figure 4.2. For *b*=0.6, we can observe that the visual quality of the output image is poor. As the value for *b* is increased, the quality of the tone mapping improves, up to a certain point. For *b*=0.9, good tone-mapping is obtained. However, when *b* is increased to 0.95, the windows on the left side are washed out; the bright areas of the image are not properly tone mapped. For *b* equal to 1, there is no tone mapping of the high dynamic range image at all; as can be seen in equation (2.4), the attenuation factor in this case is equal to 1. A *b* value of 0.9 was chosen for our

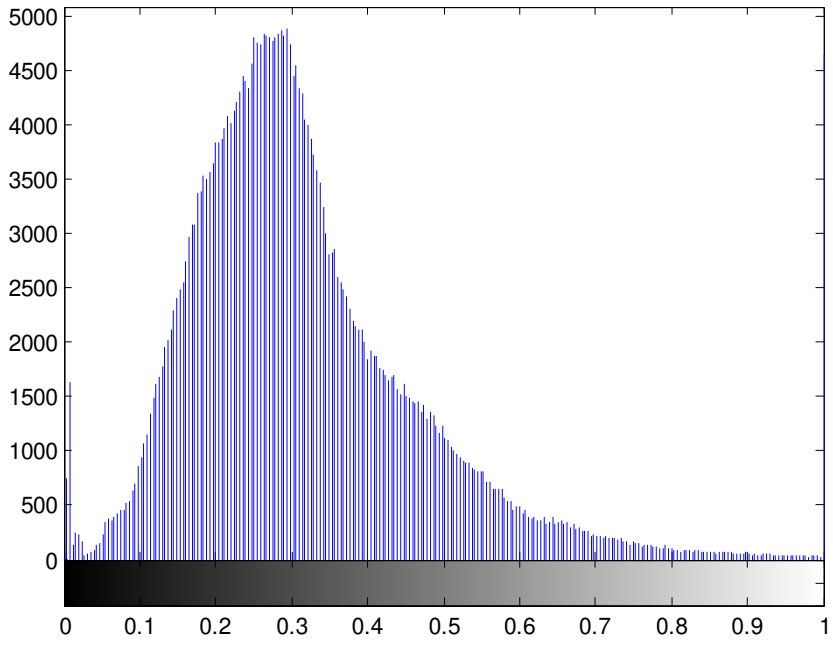


Figure 4.3 Histogram for Nave with *a*=1.5 and *b*=0.9

implementation. Note that the remaining images from the Debevec library were also tested for different b values and good quality tone-mapped images were obtained for b=0.9.

Figure 4.3 shows the histogram for Nave image obtained after tone mapping implemented in MATLAB using a=1.5, b=0.9. The x-axis shows the intensity range using 256 bins normalized between 0 and 1, and the y-axis shows the number of pixels in each bin. The histogram has a Gaussian-like spread, suggesting that the parameters a=1.5, b=0.9 are suitable choices for implementing the tone mapping operator in hardware. It is also evident from Figure 4.2 that a=1.5 and b=0.9 is a good choice of parameters to obtain a high quality output image.

4.2 Obtaining Gaussian Pyramids

Real world scenes have edges at multiple scales, and so do the HDR images capturing those scenes. The objects in the images may be of different sizes and at different distances. Some information that can be detected at different scales may be missed if an image is analyzed at only a single scale. To detect all the important intensity transitions, Fattal's operator employs a multi-scale edge detection scheme using Gaussian pyramids that are constructed using the HDR image in the logarithmic luminance domain. In Fattal's original work, the levels of the pyramid, each of which is a lower resolution version of the one before, are produced recursively; from a given scale, a scale of reduced resolution is found by lowpass filtering and downsampling by a factor of two. As previously defined in equation (2.1), the filter used for a Gaussian pyramid is

constructed using a Gaussian density function centered at zero with a specified standard deviation, and extending a specified distance from the center.

Implementing true Gaussian pyramids, with their exponential filter weights, is computationally difficult and requires extensive hardware. Approximating Gaussian pyramids using powers-of-two was proposed by Hassan [20]; this method is easy to implement in hardware. Four square windows of size 4×4, 8×8, 16×16 and 32×32 pixels are used to obtain the four scales of an approximate Gaussian pyramid with σ=1, 2, 4, and 8, respectively. These four scales are used in our implementation of Fattal's operator. The approximated weights obtained using powers of two are as follows:

4×4 square window approximating σ=1 with weights $w_{i,j} = 16^{i-2} \times 16^{j-2}$.

8×8 square window approximating σ=2 with weights $w_{i,j} = 4^{i-4} \times 4^{j-4}$.

16×16 square window approximating for σ=4 with weights $w_{i,j} = 2^{i-8} \times 2^{j-8}$.

32×32 square window approximating for σ=8 with weights $w_{i,j} = 2^{\left\lceil \frac{i}{2} \right\rceil - 8} \times 2^{\left\lceil \frac{j}{2} \right\rceil - 8}$.

where $\lceil \ \rceil$ denotes a ceiling function. Since the windows are two-dimensional, they can

Figure 4.4 The original gray scale Nave image in logarithmic domain.

be implemented using two one-dimensional filters. In [20], the vertical filters are implemented using an accumulator-based structure and horizontal filters are implemented using finite impulse response and accumulator-based structures. A detailed description for the implementation of these filters can be found in [20].

Figure 4.5 compares the Nave image processed using true Gaussian and powers-of-two approximation approaches. It shows an example image and its four scales, in order from finest to coarsest, as computed by the two approaches. The Nave image is taken for this purpose in its logarithmic domain which is shown in Figure 4.4. Looking at the images in Figure 4.5, the coarser the image gets at each scale, the blurrier the image becomes using both the approaches. The powers-of-two approximation does a good job at each scale in low pass filtering the image and obtaining a blurry image similar to that obtained using true Gaussian filters. The blurriness can be observed particularly near the windows in the Nave image. Since the powers-of-two approximation can be implemented using simple shifts in hardware, use of the powers-of-two approximation decreases the computation cost. Thus, the powers-of-two based Gaussian filtering is a reasonable approximation to true Gaussian filtering.

Now that the preliminaries to developing a real-time implementation of Fattal's operator have been decided, the next step is to design the hardware for the implementation. The hardware is implemented using VHDL. The next chapter describes the hardware implementation for Fattal's operator using our local Poisson solver. Based on the experimental simulation results given in this chapter, the attenuation parameters are fixed at a=1.5 and b=0.9.

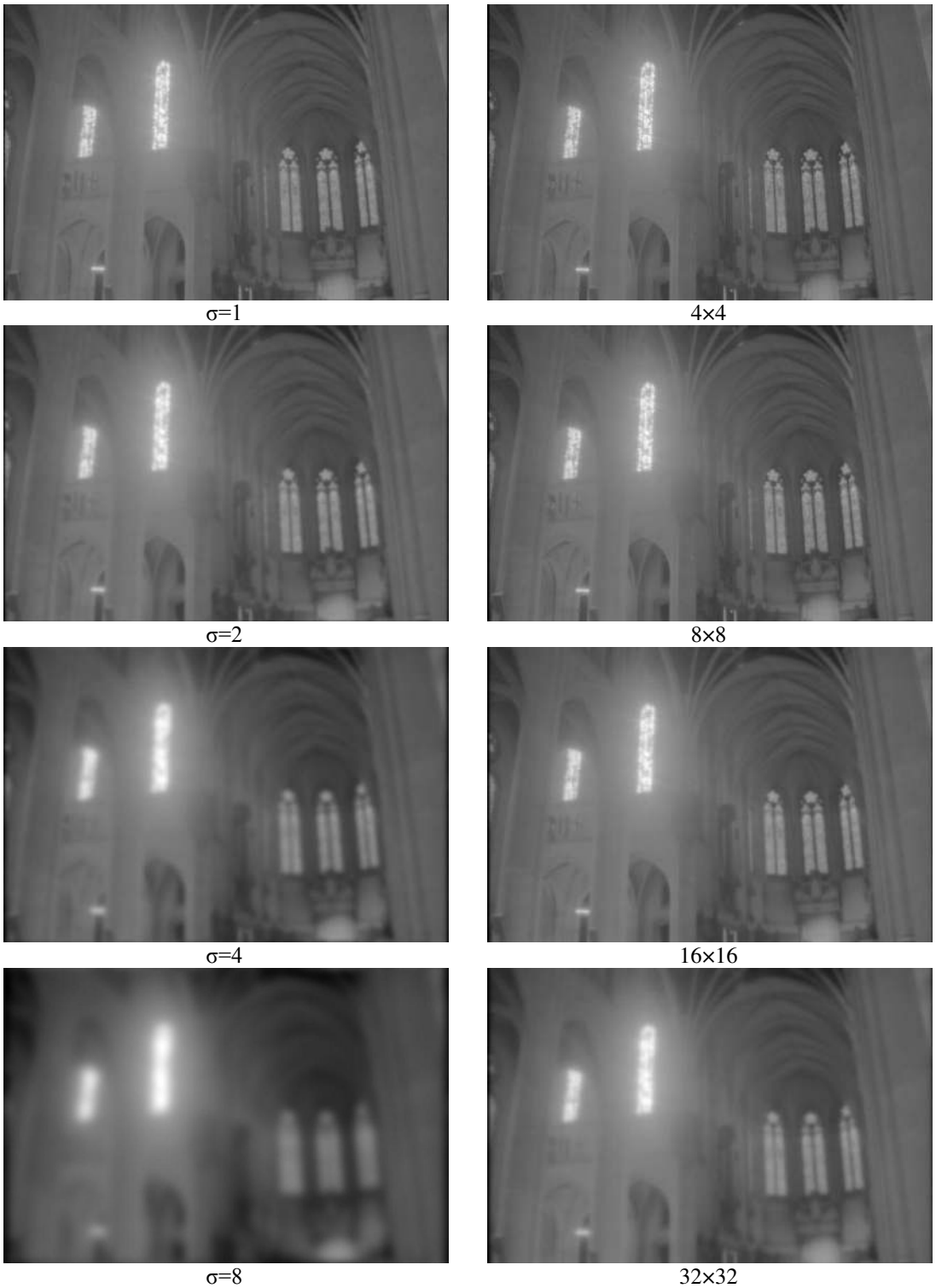


Figure 4.5 Comparison of four-level Gaussian pyramid images, calculated using a full exponential and powers of-two approximation.

CHAPTER V

HARDWARE IMPLEMENTATION

An image processing algorithm would require implementation in real-time to be usefully embedded within a camera or a display device. Most of the published tone mapping operators have been developed to be applied on stored images without considering their hardware implementation. Successful real-time implementation of these algorithms requires making hardware-friendly approximations that preserve the basic sense of the computations and do not adversely impact the quality of the processed images. It is also necessary to consider the effects of the use of fixed-point mathematics; generally, floating-point mathematics is too slow and expensive for real-time embedded systems. The preliminaries required to implement Fattal's operator have been discussed in Chapter 4. This chapter describes the hardware implementation of Fattal's operator taking fixed-point arithmetic and approximations for high complexity computations into account.

The block diagram of the hardware implementation of Fattal's operator using our local Poisson solver is shown in Figure 5.1. The input is a 3×3 window of the HDR image in the logarithmic luminance domain and the four scales of the Gaussian pyramid obtained from the full resolution HDR image. The four scale images of the Gaussian pyramid are obtained using a hardware-friendly powers-of-two approximation previously proposed in [20] for standard deviations of 1, 2, 4 and 8, as already discussed in Chapter

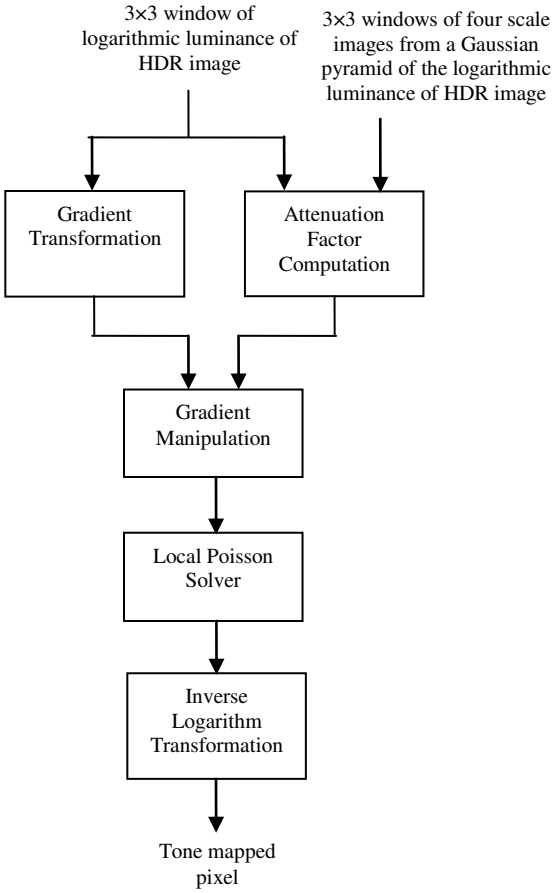


Figure 5.1 Block diagram of Fattal's operator on a 3×3 window.

4. We take a 3×3 window from the image, and calculate horizontal and vertical gradients around each pixel in the 3×3 window. At the same time, the 3×3 window and corresponding 3×3 windows from the Gaussian pyramid are used to calculate an attenuation factor for each pixel in the window. The attenuation factors are multiplied pixel-by-pixel with the vertical and horizontal gradients to obtain the manipulated gradients. Using our local Poisson solver, we calculate the output pixel in the middle of the window from its surrounding manipulated gradients. An inverse logarithm is applied to the output pixel to transform to its linear luminance (i.e., gray-scale) form. This process is repeated for each pixel in the output image, as the 3×3 window slides over the

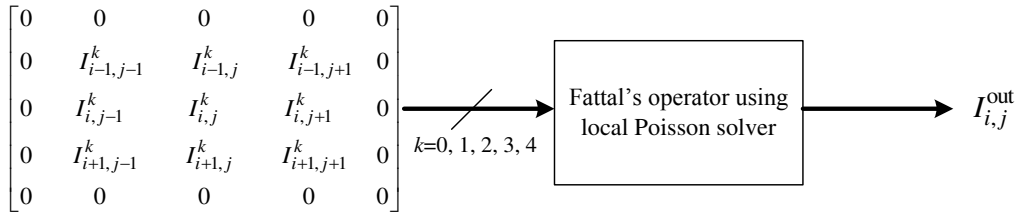


Figure 5.2 High level view of our hardware.

full-size input image. This process is shown as a high level diagram in Figure 5.2. Note that in the entire description of our hardware that follows, each of the 3×3 windows along with zero Dirichlet boundary condition along the borders, slid throughout the input image, is fed into our hardware and solved independently. The output of our hardware is a single pixel which is the center pixel of the 3×3 window. Any computations done on the 3×3 window, along with its borders, are used to solve for that center pixel only. Each pixel of the tone-mapped output image is produced completely independently of the other output pixels, using only the 3×3 window surrounding that pixel.

Many published algorithms for processing HDR images were developed without taking into account the effects of precision when the algorithms are implemented in fixed-point arithmetic. Implementation of Fattal's operator in hardware requires development of two main parts: calculation of the attenuation factor for a given pixel, and calculation of the tone-mapped output pixel. It is important to determine the number of integer bits and fraction bits required to implement the method in hardware without adversely affecting the quality of the output image; the goal is for the fixed-point embedded implementation to be a close approximation to a floating-point (non-embedded) implementation.

To assist with the precision analysis, a MATLAB simulation is developed that is true to the hardware at the bit level. The simulation allows us to try different fixed-point precisions and see the quality of the results, so that we can decide on the fraction and integer bits required at every stage. First, the attenuation factor is calculated using the fixed-point simulation, while the rest of the method is simulated in floating-point to decide on fixed-point formats within the attenuation factor computation block; then, fixed-point formats for the rest of the hardware are decided by simulating the attenuation factor calculation in fixed-point using the decided-upon formats, and trying a variety of fixed-point formats for the rest of the hardware. Different numbers of fraction bits are tried during the process; by looking at the visual quality of the tone-mapped output images, an appropriate number of fraction bits is chosen. The integer bits are decided by taking into account the maximum range of values possible in every stage of Fattal's operator. Bits are allocated to each variable such that the entire required range of that variable can be represented, with no possibility of overflow. The method used to obtain the fraction bits for the individual variables is discussed as the hardware is described.

For hardware implementation, an appropriate data format must be chosen for each signal value. We use the notation $(n, -f)$ to denote a fixed-point format where f is the number of fraction bits and $n-f$ is the number of integer bits. Throughout our design, the precision of a signal is decided by analyzing the effect of fixed-point quantization on the quality of the output image. We assume that the input pixels, which are logarithmic, have a (15,-10) unsigned fixed-point format; this format covers a large enough dynamic range to represent high dynamic range images, including those from the Debevec library [9].

5.1 Conversion to Floating-Point

As will be described later, part of the attenuation factor computation is done using a floating-point representation. The hardware used to convert to floating-point is described next. The conversion to floating-point block is implemented following the work in [27]. A block diagram is shown in Figure 5.3. The input to this block is a 16-bit fixed-point integer; the output consists of four exponent bits and six mantissa bits. The block consists of registers and multiplexers. The main aim of this block is to look for the

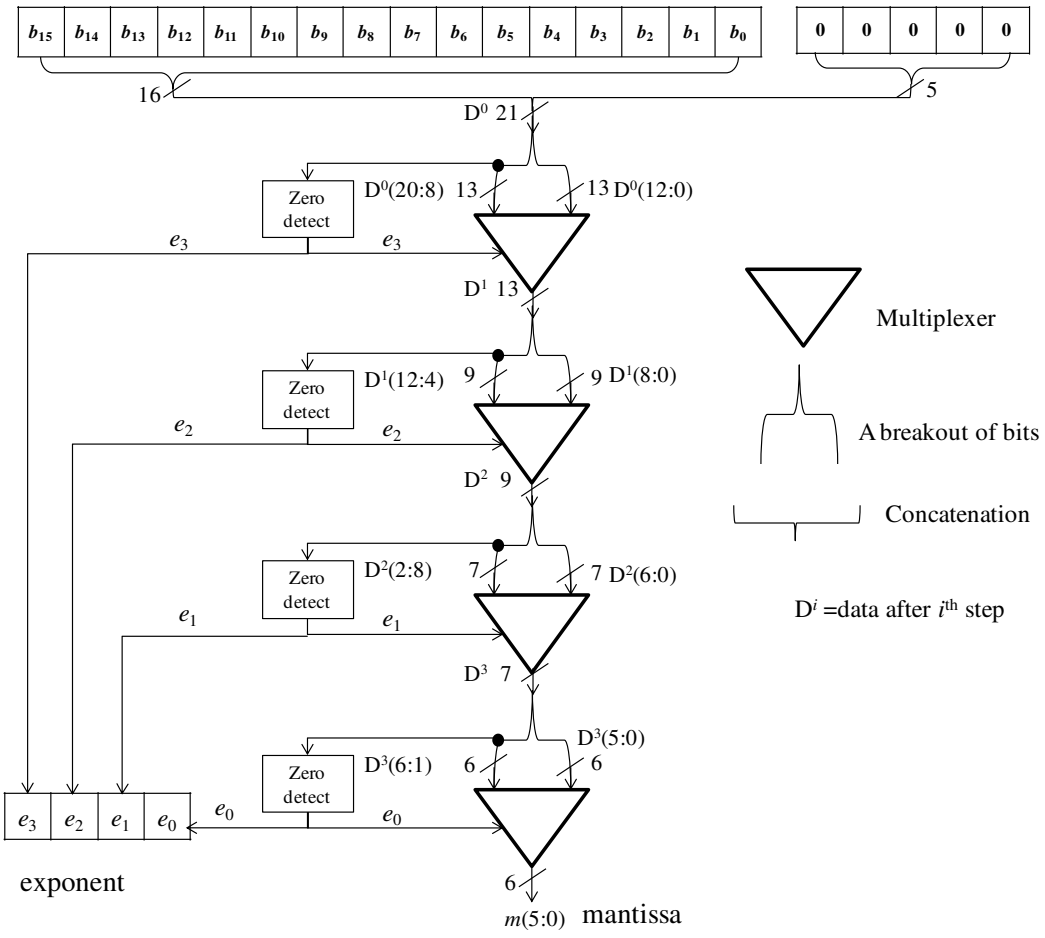


Figure 5.3 Conversion to floating-point block.

most significant bit (MSB) of significant information in the input using a binary search strategy; each step in the binary search determines one additional bit in the value of the exponent. When the exponent is fully determined, the most significant six bits of the input are transferred to the mantissa.

The binary search is accomplished with a series of multiplexers. To start, the input to the conversion block is padded with five zeros on the right, for a total of 21 bits. The first stage checks whether there is significant information in the top 13 bits; if so, the MSB of the exponent is set to '1' and the most significant 13 bits of the input are passed through to the next stage; otherwise, the MSB of the exponent is set to '0' and the least significant 13 bits of the input are passed through to the next stage. The second stage decides on the next most significant bit of the exponent by checking for significant information in the top nine bits of its input, and passes through either the top or bottom nine bits of its input, as appropriate, to the third stage. This process is continued; after four stages, the exponent is fully determined and the remaining six bits of data form the mantissa. The process is such that the top bit of the mantissa is always a '1', unless the input to the block is all zeros. The mantissa is in an unsigned (6, -5) format; in later hardware stages, we consider the mantissa to be in an unsigned (6, -6) format by simply adding an offset of one to the exponent.

5.2 Attenuation Factor Computation

The attenuation factor computation is an important step in Fattal's operator, as it decides the extent to which the gradients of the HDR image are attenuated or magnified.

The hardware implementation of the attenuation factor calculation for a single pixel centered in the 3×3 window is described below.

The attenuation factor for each pixel in the 3×3 window involves the HDR image $\mathcal{I}^0$ and its four scale images from a Gaussian pyramid $\mathcal{I}^k$ (k=1, 2, 3 and 4). The central difference of the pixel in vertical and horizontal directions is calculated for the 3×3 window from the image and 3×3 windows of its four scale images. The one-norm of the central differences, denoted by $\mathcal{C}_{u,v}^k$, are obtained for the pixel and then the attenuation factor $\Psi_{u,v}$ is calculated as:

$$\Psi_{u,v} = \prod_{k=0}^{4} \left(\frac{\mathcal{C}_{u,v}^k}{a} \right)^{(b-1)} \tag{5.1}$$

where a=1.5 and b=0.9, as chosen in Chapter 4. The indices u and v vary from 0 to 3.

Calculating the attenuation factor as given in equation (5.1) is computationally intensive, with multiple multiplications of values with large number of bits. The computation can be simplified by first converting the five central differences to a floating-point representation; floating-point allows for values of large dynamic range to be represented with fewer bits. Each floating-point number consists of a mantissa and an exponent scaled by a base; in our case, the base is two. In a simulation experiment designed to assess the number of fraction bits required for the attenuation factors, the five mantissas obtained from the five central differences from the original HDR image and its four scales of the Gaussian pyramid are fixed to use a certain number of fraction bits and the rest of the computations in Fattal's operator are done in floating-point mathematics.

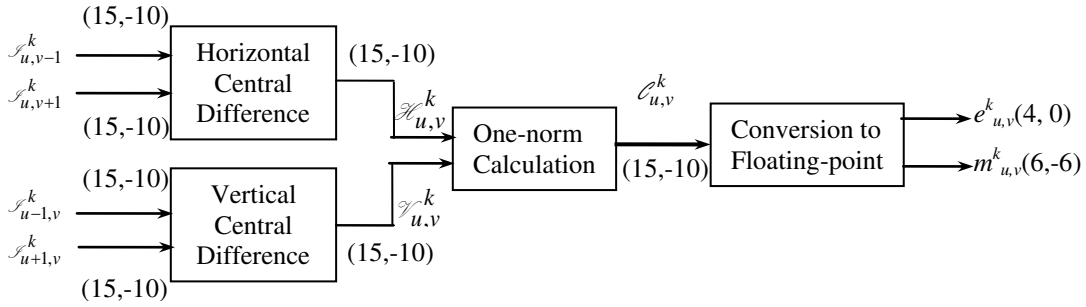


Figure 5.4 Hardware block to calculate the central difference for a single scale in a floating-point (mantissa and exponent) format.

Taking six fraction bits for the five mantissas gave a good visual quality output image, without any halo artifacts.

The floating-point representation used in our hardware then represents a central difference of scale *k* with a mantissa and an exponent as:

$$\mathscr{C}_{u,v}^{k} = m_{u,v}^{k} \cdot 2^{e_{u,v}^{k}} \tag{5.2}$$

where the mantissa $m_{u,v}^{k}$ is a six-bit number in an unsigned (6, -6) format and the exponent $e_{u,v}^{k}$ is a four-bit unsigned integer.

Figure 5.4 shows the hardware to calculate the central difference for a single scale in a floating-point (mantissa and exponent) format. It consists of a horizontal central difference block and vertical difference block to calculate the horizontal and central differences of input pixels at scale *k* respectively. This calculation involves simple subtractions and a right shift operator to achieve the division by two. It is calculated as:

$$\mathscr{H}_{u,v}^{k} = \frac{\mathscr{I}_{u,v-1}^{k} - \mathscr{I}_{u,v+1}^{k}}{2} \text{ and } \mathscr{V}_{u,v}^{k} = \frac{\mathscr{I}_{u-1,v}^{k} - \mathscr{I}_{u+1,v}^{k}}{2} \text{ for } k\text{=0 to 4.} \tag{5.3}$$

These vertical and horizontal central differences are fed to the one-norm block which computes the one-norm as:

$$\mathscr{C}_{u,v}^{k} = \left|\mathscr{H}_{u,v}^{k}\right| + \left|\mathscr{V}_{u,v}^{k}\right|. \tag{5.4}$$

The absolute value of a negative central difference is approximated using the ones-complement of the value. This makes the hardware simpler, because taking the ones-complement involves simply inverting all of the bits. The error is one least significant bit, which is negligibly small. After obtaining the one-norm of the central difference, it is converted to floating-point: mantissa and exponent using the conversion to floating-point block. Note that because the conversion to floating-point block considers the input to be an integer, but the actual input has ten fraction bits, the exponent produced at the output is off by ten.

The attenuation factor, using the floating-point representation with a mantissa and exponent, can be rewritten as:

$$\Psi_{u,v} = \left(\prod_{k=0}^{4} m_{u,v}^{k}\right)^{b-1} \left(\frac{2^{\sum_{k=0}^{4} e_{u,v}^{k}}}{a^{5}}\right)^{b-1}. \tag{5.5}$$

Figure 5.5 shows how the central differences at the four scales, in floating-point format, are used to calculate the attenuation factor. Each signal is annotated with its fixed-point format. The product of the central differences must be found; this is done by multiplying all of the mantissas, and adding all of the exponents. The advantage of computing the attenuation factor in floating-point is that multiplications, now involving only the mantissas, are only six bits by six bits. After each multiplication, the floating-point

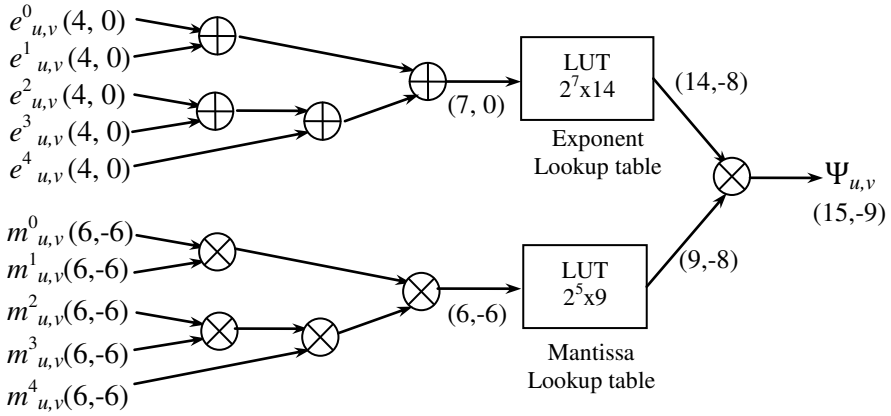


Figure 5.5 Hardware to compute attenuation factor from the central differences of the five scales in floating-point (mantissa and exponent) format.

intermediate result is normalized so that the mantissa of the product is between 0.5 and 1. Note that the normalization is not shown in Figure 5.5. Also not shown in the figure is the offset compensation needed because the mantissas at the output of the conversion to floating-point block were interpreted to be in a (6, -6) format, as mentioned earlier; because there are five such mantissas, the compensation requires adding five to the final exponent of the product.

The values of the two power terms in equation (5.5) are obtained using lookup tables. The first lookup table is the mantissa lookup table and is indexed by the mantissa of the final product. The most significant bit of the mantissa is always an implied '1', so it does not need to be used as part of the address. The input to the mantissa lookup table is only five bits of the mantissa without the MSB; as a result, there are 2^5 entries in the lookup table. This lookup table returns the (b-1) power of the mantissa of the final product; here, the b value is 0.9, and the mantissa is normalized to be between 0.5 and 1, so the output power term is between 1.000 and 1.072. This means that the output of the

mantissa lookup table requires one integer bit; the number of fraction bits is yet to be decided.

The second lookup table is the exponent lookup table and is indexed by the sum of the exponents. Each of the exponents has four bits; when all the exponents are added together, the final sum of the exponents has seven bits to avoid the possibility of any overflow. As a result, the exponent lookup table has 2^7 entries. Note that because each exponent has an offset of -10, the sum has an offset of -50; this means that a sum of zero corresponds to an exponent of -50, and the sums range from -50 to +77. The exponent lookup table produces the second power term in equation (5.5) as its output. Because the term can range from 0.0059 to 39.19, the output requires six integer bits; again, the number of fraction bits is yet to be determined. The values read from the mantissa and exponent lookup tables are multiplied together to get the final attenuation factor for each pixel. This value also requires six integer bits.

An important consideration in the design is how many fraction bits should be used for the values read from the lookup tables. A precision analysis is done by simulating Fattal's operator in MATLAB and observing the quality of the output images. Given that the number of fraction bits in each of the five mantissas has already been set to six, the number of fraction bits for the output of the mantissa lookup table is varied. The rest of the method uses floating-point mathematics. The image with eight fraction bits for the output of the mantissa lookup table does not have any halo artifacts.

Fixing the mantissas to six fraction bits and the output of the mantissa lookup table to eight fraction bits, the fraction bits for the output of the exponent lookup table are varied. Eight fraction bits for the exponent lookup table gave good quality images.

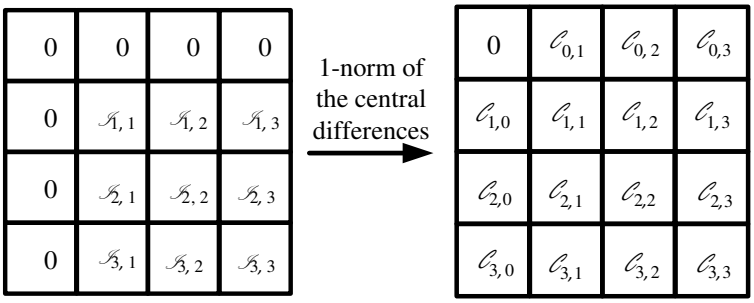


Figure 5.6 Central differences for a 3×3 window.

Similarly, the fraction bits required for the product of the outputs of the two lookup tables are obtained. This product, which is the attenuation factor, requires nine fraction bits.

Now that the fraction bits are decided for each of the lookup tables and the attenuation factor, the sizes of the lookup tables can be determined. The size of the mantissa lookup table is 2^5×9 bits; each entry in the table has one integer bit and eight fraction bits. The size of the exponent lookup table is 2^7×14 bits; each entry in the table has four integer bits and eight fraction bits.

The attenuation factors for all the pixels in the 3×3 window, including the top and left boundaries of the window, are calculated in a similar way. A zero Dirichlet boundary condition is assumed for the boundaries, so that a total of sixteen attenuation factors are calculated for a single 3×3 window. As shown in Figure 5.6, the top corner boundary pixel has a zero central difference in the image and its four scale images, so its attenuation factor is zero. Also, because the top and left boundaries are all zeros, it happens that

$$\mathcal{C}_{0,1}^{k} = \mathcal{C}_{1,0}^{k} \text{ for } k=0, 1, 2, 3, 4 \tag{5.6}$$

so that the corresponding attenuation factors $\Psi_{u\text{-}2,v\text{-}1}$ and $\Psi_{u\text{-}1,v\text{-}2}$ are equal. This results in fourteen unique attenuation factors to be calculated for each pixel in the output image. Each attenuation factor is in a (15, -9) unsigned format.

5.3 Gradient Transformation and Gradient Manipulation

The hardware for gradient transformation and gradient manipulation on a 3×3 block is shown in Figure 5.7. The horizontal and vertical gradients are computed on the 3×3 block of the HDR image using forward differences as:

$$\mathcal{G}^{H}_{u,v} = \mathcal{I}^{0}_{u,v} - \mathcal{I}^{0}_{u,v+1} \text{ and } \mathcal{G}^{V}_{u,v} = \mathcal{I}^{0}_{u,v} - \mathcal{I}^{0}_{u+1,v}. \tag{5.7}$$

The manipulated gradients are obtained by multiplying the gradients in both horizontal and vertical directions with their corresponding attenuation factors:

$$\hat{\mathcal{G}}^{H}_{u,v} = \Psi_{u,v} \cdot \mathcal{G}^{H}_{u,v} \text{ and } \hat{\mathcal{G}}^{V}_{u,v} = \Psi_{u,v} \cdot \mathcal{G}^{V}_{u,v}. \tag{5.8}$$

To obtain the fraction bits required for the gradients and manipulated gradients, Fattal's operator is again simulated in MATLAB as done previously for the precision analysis. The attenuation factor computation is done using fixed-point mathematics and the fraction bits are varied for the gradients and the manipulated gradients. They are assigned the same number of fraction bits for the simulation. Any precision above nine fraction bits gave good quality images without any halo artifacts. Ten fraction bits are used in our implementation.

For our hardware, gradients are in a signed (16, -10) format and the manipulated gradients are in a signed (15, -10) format; the manipulated gradients require one fewer integer bit than the gradients because they are attenuated.

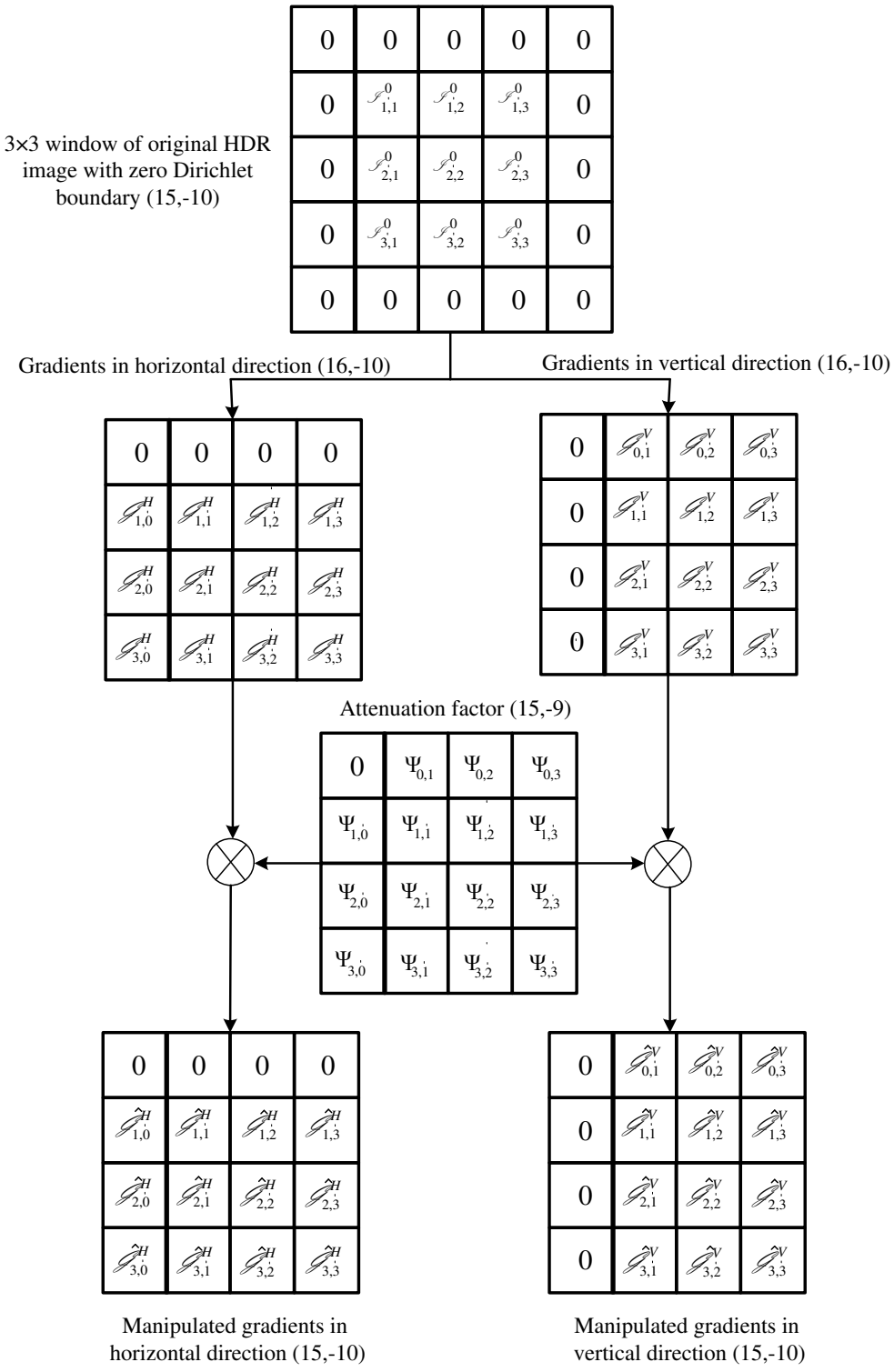


Figure 5.7 Hardware to obtain gradients and manipulated gradients.

5.4 Local Poisson Solver

Once the manipulated gradients are obtained, the divergence of the manipulated gradients needs to be computed to obtain nine different divergences corresponding to the 3×3 window around one pixel in the HDR image. These divergences can be computed using backward differences as:

$$\mathrm{div}\hat{\mathcal{G}}_{p,q} = \hat{\mathcal{G}}^{V}_{p,q} - \hat{\mathcal{G}}^{V}_{p-1,q} + \hat{\mathcal{G}}^{H}_{p,q} - \hat{\mathcal{G}}^{H}_{p,q-1}. \tag{5.9}$$

The indices p and q vary from 1 to 3.

The divergence calculation in equation (5.9) can be combined with the Poisson solving equation in (3.20) to yield a computation that involves only additions, subtractions, and simple shifts:

$$\begin{aligned} I^{\mathrm{out}}_{i,j} = {} & \frac{1}{16}(\hat{\mathcal{G}}^{V}_{2,1} + \hat{\mathcal{G}}^{V}_{3,1} - \hat{\mathcal{G}}^{V}_{0,1} - \hat{\mathcal{G}}^{V}_{1,1}) \\ & + \frac{1}{16}(\hat{\mathcal{G}}^{V}_{2,3} + \hat{\mathcal{G}}^{V}_{3,3} - \hat{\mathcal{G}}^{V}_{0,3} - \hat{\mathcal{G}}^{V}_{1,3}) \\ & + \frac{1}{8}(\hat{\mathcal{G}}^{V}_{3,2} - \hat{\mathcal{G}}^{V}_{0,2}) + \frac{1}{4}(\hat{\mathcal{G}}^{V}_{2,2} - \hat{\mathcal{G}}^{V}_{1,2}) \\ & + \frac{1}{16}(\hat{\mathcal{G}}^{H}_{1,2} + \hat{\mathcal{G}}^{H}_{1,3} - \hat{\mathcal{G}}^{H}_{1,0} - \hat{\mathcal{G}}^{H}_{1,1}) \\ & + \frac{1}{16}(\hat{\mathcal{G}}^{H}_{3,2} + \hat{\mathcal{G}}^{H}_{3,3} - \hat{\mathcal{G}}^{H}_{3,0} - \hat{\mathcal{G}}^{H}_{3,1}) \\ & + \frac{1}{8}(\hat{\mathcal{G}}^{H}_{2,3} - \hat{\mathcal{G}}^{H}_{2,0}) + \frac{1}{4}(\hat{\mathcal{G}}^{H}_{2,2} - \hat{\mathcal{G}}^{H}_{2,1}). \end{aligned} \tag{5.10}$$

A similar precision analysis is carried out, as was done for previous stages of the hardware, for determining the fraction bits required for the output of the local Poisson solver. Ten fraction bits are chosen. The output pixel is in a signed (15, -10) format.

5.5 Inverse Logarithm Transformation

The final step of Fattal's operator is to compute the inverse base-2 logarithm of the output pixel. We separate the output pixel into a signed integer part *X* with five bits and a fraction part *Y* with ten bits. Hence,

$$2^{I_{i,j}^{\text{out}}} = 2^{X_{i,j}^{\text{out}}} \times 2^{Y_{i,j}^{\text{out}}}, \tag{5.11}$$

where $2^{Y_{i,j}^{\text{out}}}$ is a number between 1 and 2. To determine the value of the fraction part of $2^{Y_{i,j}^{\text{out}}}$, a lookup table is used; the integer part is always one. The lookup table used has a ten-bit address and returns an eight-bit fraction. The multiplication by $2^{X_{i,j}^{\text{out}}}$ is accomplished by feeding $2^{Y_{i,j}^{\text{out}}}$ to a barrel shifter controlled by the integer part $x_{i,j}^{\text{out}}$. If the number of positions to be shifted is negative, the barrel shifter outputs a zero. If the number of positions to be shifted is greater than or equal to eight, the barrel shifter saturates its output to 255. The fraction bits of the output of the barrel shifter are truncated to get the final inverse base-2 logarithm of the output image pixel, in an unsigned (8, 0) format.

Hardware for a real-time Fattal's operator is successfully implemented by simplifying complex computations, while taking into account the effects of fixed-point quantization. The hardware is described using VHDL. The hardware developed is independent of the input image size and uses very little memory. It is synthesized on a field programmable gate array (FPGA) and tested on standard HDR images from the Debevec library. The synthesis and simulation results are discussed in Chapter 6.

CHAPTER VI

SYNTHESIS AND SIMULATION RESULTS

This chapter details the synthesis and simulation results for the hardware developed in Chapter 5. The hardware has been described in VHDL and is synthesized using Quartus tools, and details of the synthesis are given. Finally, simulation results are discussed verifying the functionality of the system, and assessing its quality.

6.1 Hardware Cost and Synthesis Results

Our approach is to design a piece of hardware that can do all the computations required to produce a single tone-mapped output pixel, so that the hardware produces one output pixel per clock. The input to the hardware consists of 3×3 windows of the logarithmic luminance HDR image and its four scale images. The window is updated as the pixels of the input video stream enter the system, row by row. Because the window is 3×3, it consists of three rows; this requires that two rows each of the image and its four scale images must be buffered in advance. This is a total of ten rows of pixels that need to be buffered from the logarithmic luminance HDR image and its four scale images; if a frame has 1024 pixels per row, and each input pixel has 15 bits, a total buffer size of 150K bits is needed.

Our implementation of Fattal's operator calculates the gradients on the 3×3 window of the logarithmic luminance HDR image and also computes attenuation factors

that are multiplied by the gradients to obtain manipulated gradients. These manipulated gradients are used in solving the Poisson equation to obtain a single output pixel. The hardware for the Fattal's operator is described in Chapter 5.

As we saw in Section 5.1, we need to calculate a total of 14 attenuation factors associated with a single 3×3 window in one clock period. In order to compute these 14 attenuation factors simultaneously, each computation has its own dedicated hardware. The computations required for computing a single attenuation factor are at most 24 additions, four 6×6 multiplications, one 14×9 multiplication, and two lookup tables of $2^5 \times 9$ and $2^7 \times 14$ bits, respectively. Hence, to calculate the 14 attenuation factors, we need a total of 336 additions, 56 6×6 multiplications, 14 14×9 multiplications, and 14 lookup tables of $2^5 \times 9$ and $2^7 \times 14$ bits, respectively. This is evident from Section 5.1.

Section 5.2 details the computations required to obtain the gradients and manipulated gradients. They require 17 additions and 23 16×15 multiplications which can be found in Figure 5.7. The Poisson equation is solved using equation (5.10) which is described in Section 5.3. It involves additions and shifts, and requires 23 additions. The inverse logarithm transformation involves a lookup table of $2^{10} \times 8$ bits.

An FPGA device is used for the hardware implementation as FPGAs are rich in wired multipliers and embedded memory, permitting low cost implementation. The Fattal's operator is described in VHDL and synthesized using Altera's Quartus II 8.1 Web Edition toolset. Table 6.1 summarizes the results of the synthesis from Quartus. Altera's Stratix II FPGA is used for the synthesis. It is a low-cost device and the smallest in the Stratix II family. The synthesis results show that the area on the FPGA device is effectively used. The DSP block 9-bit elements are used to implement multiplications.

The total block memory bits show the total memory used for the implemented hardware including the lookup tables and buffers; the total memory used is about 38 KB. Other computations apart from the multiplications are implemented by the logic elements.

An operating frequency of 114.18 MHz is achieved. This operating frequency is capable of processing about 100 frames per second for a one megapixel frame. This is calculated by dividing the operating frequency with the frame size which is one megapixel. Note that this is a large improvement over [21], which implemented the same operator using a multigrid solver for the Poisson equation on a notebook PC with a 2.2GHz Core 2 Duo processor and 4GB of RAM; that paper reported a throughput of around three frames per second for a one megapixel frame.

Now that we have the synthesis results, the next step would be to verify the hardware on standard HDR images. Simulation of VHDL code using Altera Quartus and code in MATLAB are compared for this purpose. The simulation results obtained are discussed in the next section.

Table 6.1 Summary of hardware synthesis report.

Device	Stratix II EP2S15F484C3	
Max operating freq.	114.18MHz	
Total registers	6586	
	Used	Available
Total block memory bits	307,200	419,328
DSP block 9-bit elements	88	96
Logic elements	9019	12480

6.2 Simulation Results

The hardware for Fattal's operator has been developed and synthesized on a FPGA device. It is important to verify the correctness of the hardware implementation. This is achieved by testing the hardware on the Nave image, one of the more demanding images from the Debevec library [9].

The images in the library are color, with each color component represented in an unsigned (32, 0) format. Fattal's operator works on gray-scale images, and so the color images must be transformed to gray-scale before processing; this is done by calculating a luminance value for each input pixel as:

$$L^{in}=0.299R^{in}+0.587G^{in}+0.114B^{in}. \tag{6.1}$$

The inputs to the hardware are the logarithm luminance of the gray scale image and its four scale images. The four scale images are produced using a Gaussian pyramid based on powers-of-two approximation in MATLAB as discussed in Chapter 4. All five images are converted to fixed-point (15, -10) format. The values of the attenuation parameters are fixed at a=1.5, and b=0.9. These values, which control the tone mapping, are determined experimentally as described in Chapter 4.

The output image obtained from a simulation of the behavioral VHDL using ModelSim is compared with the output image produced by a fixed-point MATLAB simulation. Both the output images match exactly, bit for bit, verifying that the hardware implemented for the Fattal's operator is correct.

After verifying the correctness of the hardware implementation, it is essential to verify that the visual quality of the output image has not been affected by the fixed-point representations and approximations used in the hardware. This is achieved by comparing

the output image produced by the hardware, which is in gray-scale, with the output image, also in gray-scale, obtained from a floating-point simulation in MATLAB. The Nave image is again used for this purpose.

The floating-point simulation in MATLAB uses an exact Gaussian pyramid to obtain the four scale images of the logarithmic luminance input image. Figure 6.1 shows the Nave image obtained from both the fixed-point implementation and the floating-point simulation. Comparing the images, the fixed-point representations and hardware simplifications are judged not to have a significant effect on the quality of the output image produced by the hardware.

The developed hardware for Fattal's operator produces gray-scale images. Conversion to color can also be implemented directly in hardware. The conversion required to produce a color tone-mapped output image is discussed in Chapter 3. This conversion process is simple, and involves inputting the red, green and blue input HDR pixels corresponding to the output pixel. This conversion to color should have negligible effect on the frame rate; because the conversion to color can be added as additional pipelined stage, the clock rate and the throughput should not be affected. However, adding the conversion to color in the hardware will introduce some additional lag from input to output. The output color images will have values in (8, 0) format to display the values between 0 and 255.

In our experiment, we have converted the gray-scale images produced by the hardware into color using MATLAB. Figure 6.2 shows the compressed HDR images from the Debevec library obtained using our hardware and converted to color using MATLAB. The system compresses the high dynamic range of the original images

effectively, without introducing halo artifacts. Fine details are reproduced well, and the dark regions can be seen clearly. In addition, this sliding window method has the benefit that it does not introduce any blocking effects.

(a) as processed using the proposed fixed-point implementation of the Fattal's operator.

(b) as processed using a floating-point implementation of the Fattal's operator.

Figure 6.1 The Nave image obtained using the proposed fixed-point and floating-point implementations of the Fattal's operator.

(a) Nave

(b) GroveC

(c) VineSunset

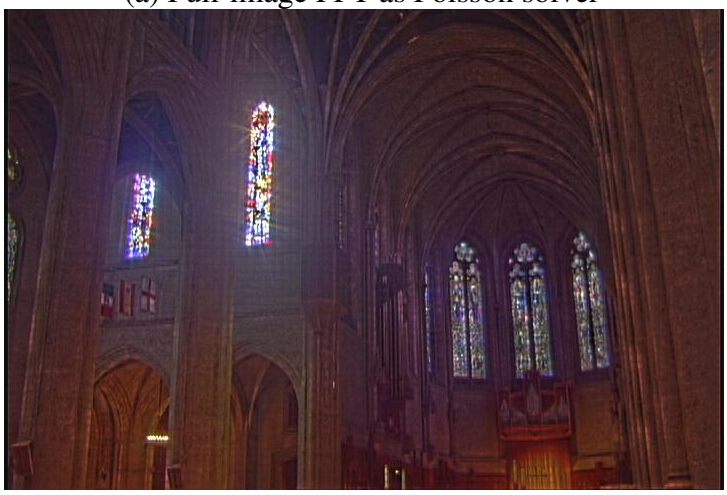
(d) GroveD

(e) Rosette

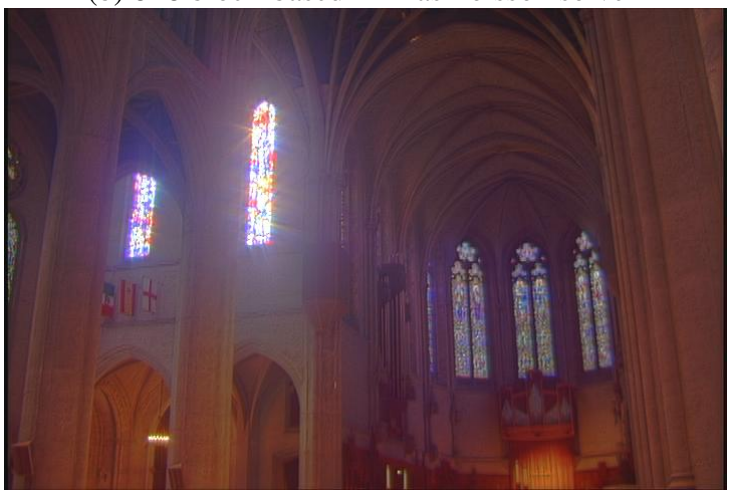
(f) Memorial

Figure 6.2 High dynamic range images from the Debevec library [9], processed using the proposed hardware implementation of Fattal's operator, and converted to color using MATLAB.

(a) Full-image FFT as Poisson solver



(b) 8×8 block-based FFT as Poisson solver



(c) Local Poisson Solver

Figure 6.3 The Nave image after a four-level gradient domain tone mapping, using three different approaches to solving the Poisson equation.

The simulation results discussed so far show that successful hardware has been implemented for the Fattal's operator using our local Poisson solver. It is important that the output image obtained using the local Poisson solver be compared to images obtained using other Poisson solvers. We compare to images produced by a full-image FFT Poisson solver which we consider a "gold standard", while noting that a full-image FFT is not realizable in real time. We also compare to images obtained using the block-based FFT Poisson solver proposed by Wang [23], which is implementable in real time.

To make a fair comparison, the same values for the attenuation parameters are used for all the three approaches. The parameter a is chosen to be 0.05 times the average of the gradients of the luminance of the HDR Nave image, and the parameter b is set to 0.9. These values are chosen so that the output images obtained using all three methods are of good visual quality. The output images are obtained using floating-point simulations for each of the solvers in MATLAB.

Figure 6.3 shows the output images obtained using the three solvers. When a full-image FFT is used to solve the Poisson equation, the resulting image is of reasonably good quality, with clear details and no halo artifacts. As might be expected, using a block-based technique to solve the Poisson equation caused blocking artifacts in the resulting tone-mapped image. Further, this approach, which uses discrete sine transforms within the 8×8 blocks, is still quite computation-intensive. In contrast, the output using our local Poisson solver has clear details and moderate contrast, and does not suffer from blocking or halos.

In results presented so far, the hardware has been successfully verified for correctness, and the output has been checked qualitatively to ensure that the use of fixed-

point mathematics has not had an adverse effect on output quality. The next section provides some quantitative measurement of output quality.

6.3 Quality Measurement

Image quality metrics like the mean square error, peak signal to noise ratio (PSNR) and structural similarity (SSIM) index [26] require a perfect image with which to compare. Since we do not have a perfect tone-mapped image, there being no ideal tone-mapping algorithm, a conclusive assessment of the quality of our hardware implementation is not possible. However, it is instructive to compare the images produced by our hardware to the closest thing there is to a "gold standard" for local tone mapping: an output image that uses a full-image FFT as the Poisson solver.

The PSNR is used for measuring the quality of our local Poisson solver and is calculated using the equation below.

$$\mathrm{PSNR} = 10 \log_{10}\left(\frac{255^2}{mse}\right), \tag{6.2}$$

where *mse* is the mean square error between the output image obtained using the full-

Table 6.2 PSNR values for floating-point and fixed-point implementations using our local Poisson solver.

HDR image	PSNR for floating-point (dB)	PSNR for fixed-point (dB)
Nave	53.8	45.2
VineSunset	50.9	41.2
GroveC	51.1	41.3
GroveD	51.4	41.5
Rosette	51.1	41.3
Memorial	51.3	47.1

image FFT and that using the local Poisson solver. The output images, obtained using the local Poisson solver and the full-image FFT, are multiplied with an appropriate scaling factor so as to cover the entire range of intensities between 0 and 255. This is to ensure that a standard intensity range be used for all the images obtained using both solvers, because each of the output tone-mapped images may have different intensity ranges.

The PSNR values for the output images using the local Poisson solver with the gold standard for both floating-point and fixed-point implementations are shown in Table 6.2. It is evident from the high PSNR values that the Fattal's operator using the local Poisson solver gives high quality output images. The values for the fixed-point implementation are also high, showing how close the approximation is to the floating-point implementation.

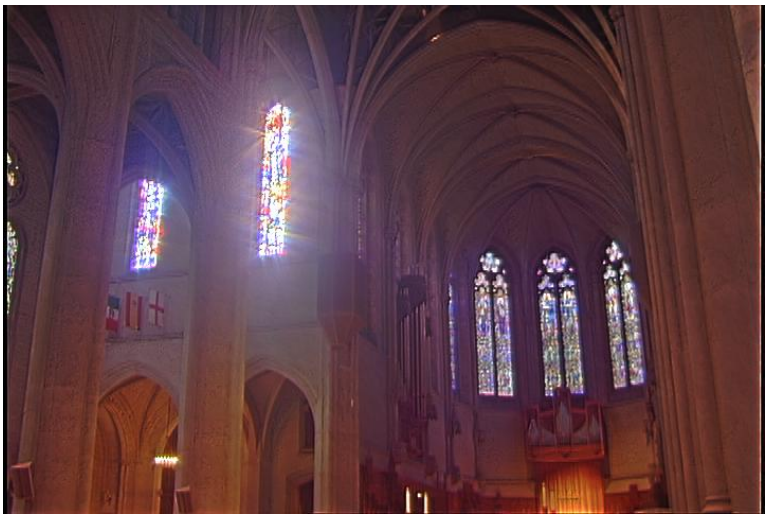
Thus, the synthesis results show that the hardware for Fattal's operator using our local Poisson solver uses small and fixed amount of hardware. It also requires very less memory and can process about 100 frames per second for a one megapixel image. The simulation results verify that the hardware is correct and produces high quality tone-mapped images. The next chapter concludes the thesis and lays a path for future work.

CHAPTER VII

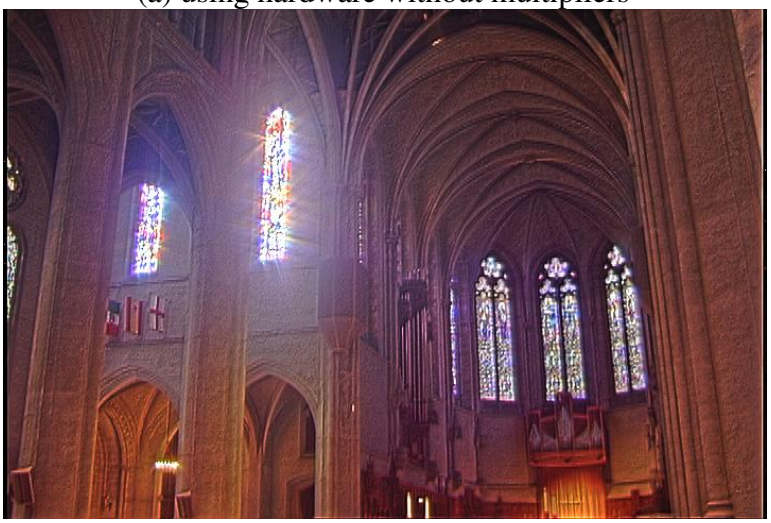
CONCLUSIONS AND FUTURE WORK

Hardware for a real-time gradient domain local tone mapping operator, Fattal's operator, has been designed and verified in simulation. A simplified Poisson solver that uses local information has been introduced; this approach to implementing an inverse gradient transformation is the key to implementing Fattal's operator on inexpensive hardware in real time. Hardware for the Fattal's operator using the local Poisson solver has been described in VHDL and synthesized for a Stratix II FPGA device using Altera's Quartus II toolset. The correctness of the hardware is verified by comparing tone-mapped images produced by the hardware in simulation with tone-mapped images produced by a fixed-point MATLAB simulation. The produced output images are of good visual quality. The quality of the local Poisson solver is also evaluated by comparing tone-mapped output images produced by using the local Poisson solver to images produced using full-image FFT and block-based FFT Poisson solvers.

The approximations used in the hardware implementation make it possible to process images in real-time on an embedded system, and do not significantly affect the quality of the output image when compared to those produced without approximations. The hardware size is fixed and independent of image size. The hardware uses significantly less memory than other approaches. High PSNR values are obtained, showing that the output images produced are of high visual quality. The hardware



(a) using hardware without multipliers



(b) using hardware with multipliers

Figure 7.1 Comparison of the Nave image obtained using hardware with and without multipliers.

implementation described takes in and produces gray scale images. However, the conversion to color is simple to implement in hardware, and should not impact the system's frame rate.

The proposed local Poisson solver processes a 3×3 window to produce a single pixel of the tone-mapped output image. One possible avenue for further research is to consider the use of different window sizes for the local Poisson solver. Using a larger

window would require more hardware, but should produce a tone-mapped output image closer to the image obtained using a full-image FFT as a Poisson solver.

We are also considering extension of the application of the local Poisson solver to other gradient domain problems that require a real-time implementation. These applications could include finite element analysis, removing photography artifacts [4] and shadows [5], image stitching [7], and gradient domain painting [8]. All these applications are based on gradient domain computations and involve taking an inverse gradient transformation, which can be posed as solving the Poisson equation.

The hardware described so far for implementing Fattal's operator using local Poisson solver includes multipliers. The hardware cost can be further reduced by designing an implementation that is completely free of multipliers. The hardware implementation remains the same as described in Chapter 5 except the attenuation factor computation and gradient manipulation; these are the only hardware blocks that have multipliers. For these parts of the calculation multipliers are avoided by moving the calculations into the logarithmic domain, where multiplications become additions. In making these modifications to the hardware, it is important to observe the quality of the output images to assess any adverse impact.

A preliminary simulation experiment was conducted in MATLAB to assess the potential of the multiplier-free version of the hardware. The Nave image is used for this purpose. The values for the attenuation parameters are fixed at a=1.5 and b=0.9, as chosen previously. Figure 7.1 compares the tone-mapped output images obtained from simulation of the original and multiplier-free computations. The tone-mapped output image from the multiplier-free version has a good visual quality and is similar to the

image obtained using hardware with multipliers. A multiplier-free version of hardware has the added benefit that it could be used to vary the values of the attenuation parameters at run-time, thus adding more flexibility to the system. Further investigation of a multiplier-free version of the hardware would require describing the system completely in hardware to determine the hardware cost and attainable frame rate, and to investigate the benefits of varying the values of the attenuation parameters at run-time.

BIBLIOGRAPHY

[1] E. Reinhard, G. Ward, S. Pattanaik, and P. Debevec. *High dynamic range imaging*, Morgan Kaufmann, 2005.

[2] R. Fattal, D. Lischinski, and M. Werman, "Gradient domain high dynamic range compression," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 249-256, July 2002.

[3] F. Hassan, L. Vytla, and J. E. Carletta, "Exploiting redundancy to solve the Poisson equation using local information," *IEEE International Conference on Image Processing*, to appear, 2009.

[4] A. Agrawal, R. Raskar, S. K. Nayar, and Y. Li, "Removing photography artifacts using gradient projection and flash-exposure sampling," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 828-835, July 2005.

[5] G. Finlayson, S. Hordley, and M. Drew, "Removing shadows from images," *European Conference on Computer Vision*, vol. 2353, pp. 823-836, May 2002.

[6] Y. Weiss, "Deriving intrinsic images from image sequences," *International Conference on Computer Vision*, vol. 2, pp. 68-75, August 2001.

[7] P. Perez, M. Gangnet, and A. Blake, "Poisson image editing," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 313-318, July 2003.

[8] J. McCann and N. S. Pollard, "Real-time gradient domain painting," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1-7, August 2008.

[9] P. E. Debevec and J. Malik, "Recovering high dynamic range radiance maps from photographs," *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 369-378, August 1997.

[10] E. Reinhard, M. Stark, P. Shirley and J. Ferwerda, "Photographic tone reproduction for digital images," *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 267 - 276, July 2002.

[11] J. Tumblin, and G. Turk, "LCIS: a boundary hierarchy for detail-preserving contrast reduction," *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 83-90, August 1999.

[12] F. Durand, and J. Dorsey, "Fast bilateral filtering for the display of high dynamic range images," *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 257-266, July 2002.

[13] P. Choudhury, and J. Tumblin, "The trilateral filter for high contrast images and meshes," *Proceedings of the 14th Eurographics Workshop on Rendering*, vol. 44, pp. 186-196, June 2003.

[14] P. Ledda, L. P. Santos, and A. Chalmers, "A local model of eye adaptation for high dynamic range images," *Proceedings of the 3rd International Conference on Computer Graphics, Virtual Reality, Visualization and Interaction in Africa*, pp. 151-160, November 2004.

[15] S. N. Pattanaik, J. A. Ferwarda, M. D. Fairchild, and D. P. Greenberg, "A multiscale model of adaptation and spatial vision for realistic image display," *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 287-298, July 1998.

[16] M. D. Fairchild, and G. M. Johnson, "Meet iCAM: An image color appearance model," *Proceedings of the IS&T/SID 10th Color Imaging Conference*, pp. 33-38, 2002.

[17] N. Goodnight, R. Wang, C. Woolley, and G. Humphreys, "Interactive time-dependent tone mapping using programmable graphics hardware," *Proceedings of the 14th Eurographics Workshop on Rendering*, vol. 44, pp. 26-37, June 2003.

[18] G. Krawczyk, K. Myszkowski, H-P. Seidel, "Perceptual effects in real-time tone mapping," *Proceedings of the 21st Spring Conference on Computer Graphics*, pp. 195-202, May 2005.

[19] C-T. Chiu, T-H. Wang, W-M. Ke, C-Y. Chuang, J-R. Chen, R. Yang, R-S. Tsay, "Design optimization of a global/local tone mapping processor on ARM SOC platform for real-time high dynamic range video," *IEEE International Conference on Image Processing*, pp. 1400-1403, October 2008.

[20] F. Hassan and J. E. Carletta, "An FPGA-based architecture for a local tone mapping operator," *Journal of Real-Time Image Processing*, vol. 2, no. 4, pp. 293-308, December 2007.

[21] M. Kazhdan, H. Hoppe, "Streaming multigrid for gradient-domain operations on large images," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1-10, August 2008.

[22] S. J. Kasbah, I.W. Damaj, and R. A. Haraty, "Multigrid solvers in reconfigurable hardware," *Journal of Computational and Applied Mathematics*, vol. 213, no. 1, pp. 79-94, March 2008.

[23] T-H. Wang, W-M. Ke, D-C. Zwao, F-C. Chen, "Block-based gradient domain high dynamic range compression design for real-time applications," *IEEE International Conference on Image Processing*, vol. 3, pp. 561-564, December 2007.

[24] Y. Zhuang and X-H. Sun, "A High-order Fast Direct Solver for Singular Poisson Equations," *Journal of Computational Physics*, vol. 171, no. 1, pp.79-94, July 2001.

[25] A. Agrawal and R. Raskar, "Gradient domain manipulation techniques in vision and graphics," *ICCV 2007 Course*, 2007.

[26] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600-612, April 2004.

[27] F. Hassan, "Real-time embedded algorithms for local tone mapping of high dynamic range images," Ph.D Thesis, *University of Akron*, 2007.

[28] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden, "Pyramid methods in image processing," *RCA Engineer*, vol. 29, no. 6, 1984.

[29] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes - the art of scientific computing," Cambridge University Press, 1992.

[30] J. Demmel, "Solving the discrete Poisson equation using Jacobi, SOR, Conjuagte Gradients and the FFT," Retrieved from Lecture Notes Online Website: http://www.eecs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html.

[31] G. Strang, "Introduction to applied mathematics," Wellesley-Cambridge Press, 1986.