Master Thesis
Computer Science

GPGPU Techniques for Real Time Tone Mapping in the Context of Virtual Night Driving

by

cand. M.Sc. Erik Bonner
Matr.-Nr. 6545572

Supervisors:

Prof. Dr. Gitta Domik
Prof. Dr. Friedhelm Meyer auf der Heide
Dr. Jan Berssenbrügge

Paderborn, September 27, 2011.

Master Thesis Nr. MA0012

GPGPU Techniques for Real Time Tone Mapping in the Context of Virtual Night Driving
September 27, 2011

Heinz Nixdorf Institut
Universität Paderborn
Fachgebiet Produktentstehung
Prof. Dr.-Ing. Jürgen Gausemeier
Fürstenallee 11
D-33102 Paderborn

Statutory Declaration:

I hereby declare that I have developed and written this thesis on my own, and no external sources were used except as acknowledged in the text and footnotes.

Paderborn, September 27, 2011

Abstract

The range of brightness that occurs in the real world far exceeds that attainable on standard display devices. As a result, computer graphics has traditionally restricted virtual illumination to the displayable range. In recent years there has been a shift in paradigm towards illumination using an intensity range commensurate with that of the real world, compressing the results to the displayable range by a post-processing operation known as *tone mapping*.

Algorithms for tone mapping, called *tone mapping operators*, can be classified as *global* or *local*. Global operators are efficient to compute at the expense of scene quality. Local operators preserve scene detail, but, due to their additional computational complexity, are rarely used with interactive applications.

This thesis proposes a local tone mapping operator suitable for use with interactive applications. The long-term goal is to use this operator with a virtual night driving simulator used at the Heinz Nixdorf Institute.

To develop a suitable tone mapping operator, a state of the art local tone mapping method was optimized using work-efficient parallel scan algorithms. Two GPU implementations of this optimized method were developed: one using CUDA and one using GLSL shaders. CUDA was chosen because it exclusively exposes GPU hardware suitable for optimizing the computations used by the proposed method. The GLSL implementation was used to determine whether performance gains over the current state of the art can be attributed to the modified algorithm or its implementation in CUDA.

Both implementations perform better than the current state of the art. The GLSL implementation produced higher frame rates than the CUDA version, at the expense of a significantly larger memory footprint. To exploit the strengths of both implementations, an additional, hybrid CUDA/GLSL method was conceived. The hybrid method achieved an optimum of both performance and memory usage. Both the hybrid and GLSL implementations operate above 60 fps for full HD resolutions on a standard PC system.

To implement all tone mapping operators presented in this thesis, an OpenGL prototyping platform was developed. Using this platform, tone mapping operators with high performance requirements can be developed, tested and compared.

Contents

Appendix

1 Introduction

1.1 Problem description

Virtual Reality (VR), the practice of simulating reality using computers, is a rapidly growing field with many areas of application. One such area is Virtual Prototyping (VP) in which Virtual Reality simulations are used for prototyping products that have not yet been built. Virtual Night Drive (VND), developed at the Heinz Nixdorf Institute, is a Virtual Reality tool for the Virtual Prototyping of automotive headlights [BER05]. Its primary focus is to support the development of headlights and headlight-based driver assistance systems by providing a realistic illumination of virtual scenery using complex beam patterns acquired from headlight prototypes. The illumination takes place as a user interactively guides a virtual car along a test track at night.

The prototypic beam pattern data that VND projects onto a virtual test track is a map of real world luminance (intensity) data with a range that spans well beyond that representable on standard display devices. Because there is no formal limit on this range, this type of data is called *High Dynamic Range* (HDR) data.

At present, as a preprocessing step, VND compresses input HDR beam pattern data into a restricted, *Low Dynamic Range* (LDR). For more accurate simulation, however, it is necessary to use original beam pattern data for illuminating the virtual test track, and, as a post processing operation on each frame, compressing the resulting HDR output so that it can be displayed on a standard monitor. Compressing HDR data into the limited, displayable intensity range is known as *tone mapping*.

Tone mapping has been the subject of intense research in recent years. There are two classes of tone mapping operators: global and local operators. Global operators apply a single transformation to all pixels in an image, while local operators consider local intensity variations during mapping. Local operators are substantially more computationally demanding than their global counterparts, but they effectively preserve local contrast that is lost by global operators. Although local operators produce favorable results, their computational complexity presents a great performance challenge for virtual reality operations with real time requirements. Therefore, until recently, interactive HDR applications have been restricted to using global tone mapping operators. Due to recent advances in graphics processing hardware, however, real time GPU-based local tone mapping is becoming possible, albeit at relatively conservative resolutions.

In order for VND to reach its full Virtual Prototyping potential, it would be highly desirable to extend its rendering system to support HDR rendering using a local tone mapping operator. Because VND is designed to be run on the *HD Visualization Center* at the HNI, on which VR applications are executed at very high resolutions, a local tone mapping operator that allows interactive frame rates at high resolutions is necessary. Since, as far as could be determined, no such operator exists, one must be developed.

1.2 Aim

The aim of this thesis is to develop a local tone mapping operator capable of tone mapping very high resolution Virtual Reality applications at interactive frame rates. To accomplish this, a promising existing tone mapping method is modified using modern, work-efficient parallel algorithms. Furthermore, an investigation into the potential for performance gains attainable by implementing this operator on the GPU using CUDA - a parallel programming language in the rapidly emerging field of General Purpose GPU (GPGPU) programming - is undertaken.

Because the long term goal of this research is the integration of local tone mapping into VND, the tone mapping operator is implemented as a self-contained GPU-based module, which can be integrated into any interactive HDR application.

1.3 Thesis structure

After the problem introduction and statement of aim are given in this chapter, Chapter 2 introduces the basic terminology and background thoery used throughout the remainder of the thesis, including a short introduction to the target applications of VP and VND, an overview of High Dynamic Rendering and coverage of basic GPU programming concepts.

The problem introduced in Section 1.1 is considered in greater detail in Chapter 3, which specifies research requirements derived from VND.

A review of the relevant previous work in high speed tone mapping is undertaken in Chapter 4. An existing tone mapping operator with potential for meeting the requirements specified in Chapter 3 is selected, and the most promising attempts at implementing it at real time frame rates are discussed.

Chapter 5 discusses the approach taken to developing a high speed local tone mapping operator that satisfies the requirements given in Chapter 3. After identifying a method for optimizing the most suitable implementation currently in the Literature, a pair of GPU-based software modules are designed for realizing the proposed method.

Focus is shifted from high level design concepts to implementational details in Chapter 6, where the procedure of implementing each module introduced in 5 is described in detail. Furthermore, a custom software platform developed to assist with evaluating tone mapping modules is introduced.

The results of the efforts of the preceding chapters are given in Chapter 7. After a visual presention of tone mapping in the context of virtual night driving, the performance of the modules developed in Chapters 5 and 6 is evaluated in detail.

Finally, Chapter 8 concludes by summarizing the work presented throughout this thesis, as well as discussing future research directions.

2 Background Theory

This chapter covers the basic background theory necessary to understand the work presented in the remainder of this thesis.

Section 2.1 introduces the concept of *High Dynamic Range* (HDR) in the context of computing imaging and graphics. This is followed by cursory discussion in Section 2.2 of image processing, concentrating on image convolution. Finally, Section 2.3 discusses programming on the GPU, including an introduction to General-Purpose GPU programming with CUDA.

2.1 HDR Concepts in Imaging and Computer Graphics

2.1.1 Color and luminance

Humans perceive the world that they inhabit using a number of senses. One of these is *sight*: light, an electromagnetic radiation that travels through space, is reflected from surfaces in the environment and subsequently perceived by the human visual system. Since each reflection alters its spectral composition, the wavelength of the light arriving at the observer's eye conveys information about the light source itself, as well as the surfaces from which it was reflected.

The human visual system is capable of perceiving electromagnetic wavelengths within a range of approximately 380 to 780 nanometers (nm) [GV97, p. 52], known as the *visible spectrum*. The visible spectrum is shown in Figure 2-1. Each wavelength in the visible spectrum constitutes a *colour*.

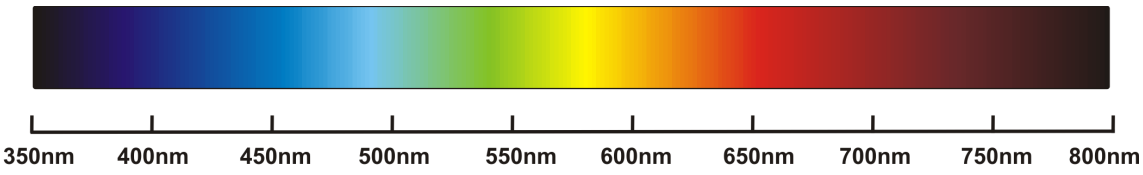


Figure 2-1: The visible portion of the electromagnetic spectrum. Different wavelengths are perceived as different colors. Wavelength is specified on the scale above in nanometers (nm) ($1\ nm = 10^{-9}\ m$*).*

Typically, light of a certain colour is a spectral distribution, i.e an additive mix of wavelengths different proportions. A colour represented by light of a single wavelength is called a *pure colour*. Experiments in the field of *colorimetry* have found that each colour in the visible spectrum can be visually matched by a linear combination of pure colors, called *primary colors*. In particular, pixels in computer display devices are colored using a linear combination of the primary colors red, green and blue.

Specifying the visual colour spectrum in terms of red, green and blue primary colour components gives rise to the *CIE-RGB* colour space. The wavelengths of the three primary components, as defined by the Interational Lighting Commision (Commission Internationale de l'Eclairage - CIE), are specified on the left in Figure 2-2. The depiction on the right in Figure 2-2 shows the RGB colour space represented as an RGB colour cube in 3D space. Each of R, G, and B axis represent the weighting of red, green and blue primary colour components, where the maximum weighting is normalized to 1.

$\lambda_R = 645.2$
$\lambda_G = 525.3$
$\lambda_B = 444.4$

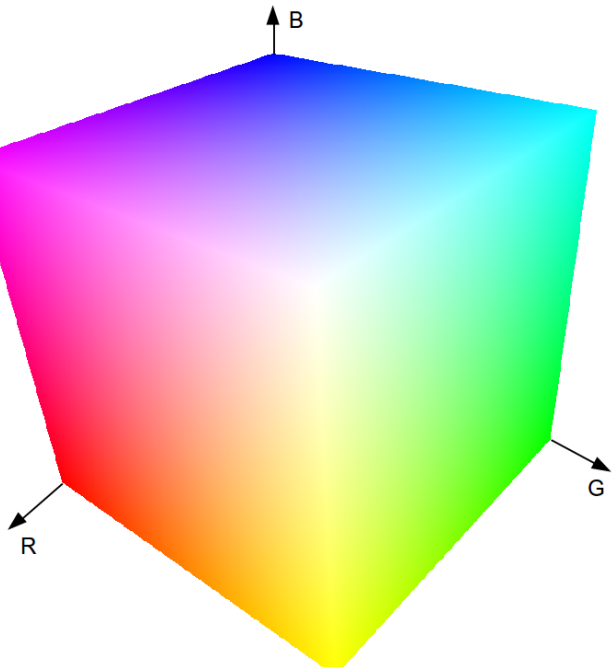


Figure 2-2: CIE-RGB colour space, represented as a unit cube. Cube image adapted from [Nie05]. Wavelength data from [RWP+06, p. 28].

The CIE-RGB colour space is consistent with the trichromatic colour perception of the human eye; in the human visual system, three types of photoreceptors, sensitive to red, green and blue wavelengths, combine their input to produce any color in the visual spectrum. The three types of photoreceptor cells do not occur in equal numbers in the eye, and, furthermore, are not equally sensitive to input light intensity. Therefore, the perceived brightness of a light source depends not only on its power, but also on its wavelength.

Photometry, the science of measuring visible light according to how it is perceived by the human eye, defines an experimentally obtained *luminous efficiency function*, $V(\lambda)$, which specifies the eye's sensitivity to light of different wavelengths [GV97, p. 67]. Figure 2-3 shows a plot of this function.

Given red, green and blue light sources of the same power, Figure 2-3 shows that the green one would be perceived as brightest of the three. The red light source would be seen as less bright than the green, and the blue one would be considered the darkest of the three.

To gauge brightness, photometry defines a measure of *luminance*, which specifies the perceived brightness of a surface when viewed from a certain direction, given in *candela per square meter* (cd/m^2) [RWP+06, p. 27]. Luminance can be computed from the red, green and blue primary component weightings of a coloured light source, by means of

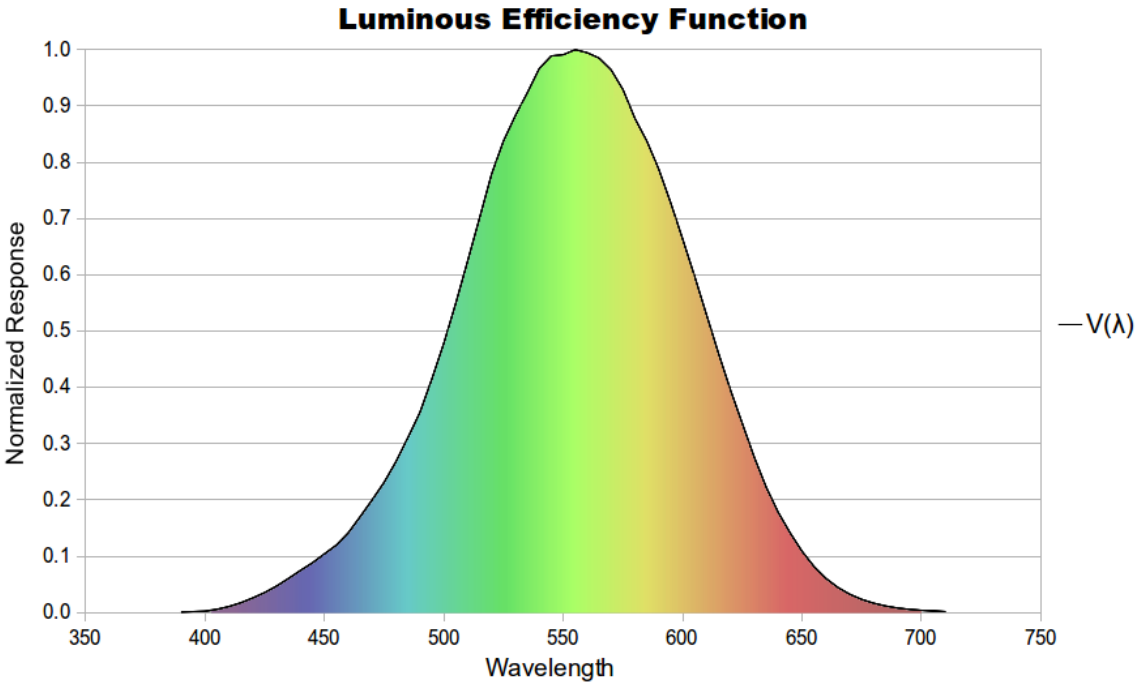


Figure 2-3: The luminance efficiency function. The x-axis shows the wavelength (λ) of the input light and the y-axis shows the normalized response of the visual system. The area under the curve is filled to indicate the perceived colour at each wavelength. For light of constant power, the curve indicates the intensity with which the light will be perceived by the visual system.

linear combination. There are a number of standards specified for the coefficients of this linear combination. For example, computing luminance from RGB components according to the ITU-R BT.709 [ITU90] standard is shown in Equation 2.1. Note that the weightings used are consistent with the relative light-efficiency function in Figure 2-3.

$$L = [0.2126, 0.7152, 0.0722] * \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.1}$$

Luminance is the standard unit of measurement used to represent brightness in HDR Rendering.

2.1.2 Dynamic range

The dynamic range of a scene denotes the ratio between its highest and lowest measurable luninance. Figure 2-4 shows, on a logarithmic gradient, the luminance of common real-world light sources and environments. Examining the values in the Figure, it can be concluded that a moonlit scene would typically have a dynamic range of 2 order of magnitude, while an indoor environment with a window through which sunlit, outdoor scenery is visible (such as the scene in Figure 2-4, would exhibit a dynamic range of about 3 order of magnitude. If objects in the outdoor scenery were particularly reflective, or the sun

itself were to be visible, this dynamic range would increase dramatically.

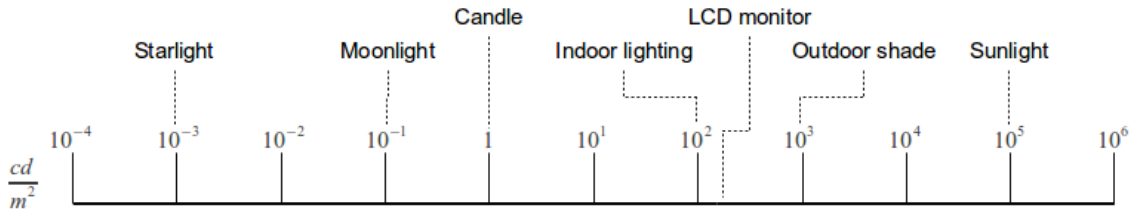


Figure 2-4: A logarithmic scale of the ambient illumination in a number of real-world environments. Values from [Wan95], [HDR].

The human visual system is capable of discerning a range of about 5 orders of magnitude in a single scene simultaneously. It can be seen in Figure 2-4 that the difference between a daylight scene and one lit by starlight is 8 orders of magnitude. Yet, both starlit and sunlit scenes can be perceived - because the eye is capable of adapting to lighting conditions that vary by close to 10 orders of magnitude [Fer01]. Because the difference between starlight and and daylight exceeds the dynamic range discernable by the human eye at one time, stars are not visible in the sky on a clear, sunny day. If, however, the sun were suddenly "turned off", after a period of adjustment, the stars would reveal their presence to the observer (who would likely nonetheless not perceive them, as they would have more pressing matters at hand).

2.1.3 HDR Imaging

Modern, general purpose LCD monitors produce a maximum luminance of about 250 to 350 cd/m^2, and therefore have a dynamic range of 2 orders of magnitude. Accordingly, most image encodings use a single byte for each of the red, green and blue color components of each pixel, restricting the luminance representable in the image to a maximum of 256 discreet levels. An image of this type contains only a fraction of the information present in the real-world scene it represents. Since their dynamic range is fixed to a constant 2 order of magnitude, such images are called Low Dynamic Range (LDR) images.

Rather than encoding images according to the limitations of the devices on which they are displayed, it is more prudent to capture them with a range of luminances representative of the scene they depict. Images encoded in such a manner have a dynamic range matching the scene they represent, and are accordingly called *High Dynamic Range* images. The science of working with HDR images is called *HDR Imaging*.

A number of HDR image encoding have arisen in recent years. Particulalry widely used encodings are the *radiance* [LS98] and *OpenEXR* [KBH07] formats. The details of how these encodings work are unimportant in the context of this thesis; it is simply necessary to note that each format uses 16-, 24- or 32-bit floating point representations to extent the range and precision of the red, green and blue (and alpha) channels of each image pixel.

Figure 2-5 shows a digital image encoded using a standard, LDR encoding. Each pixel is represented by a 3-tuple of 8 bit integer values, representing the red, green and blue colour weightings. An digital image encoded in a HDR format is shown in Figure 2-6. Using this encoding, each colour weighting is specified with a high precision, floating point value.

Furthermore, the range of intensities that can be specified extends well beyond that of $[0, 255]$ possible in traditional encodings.

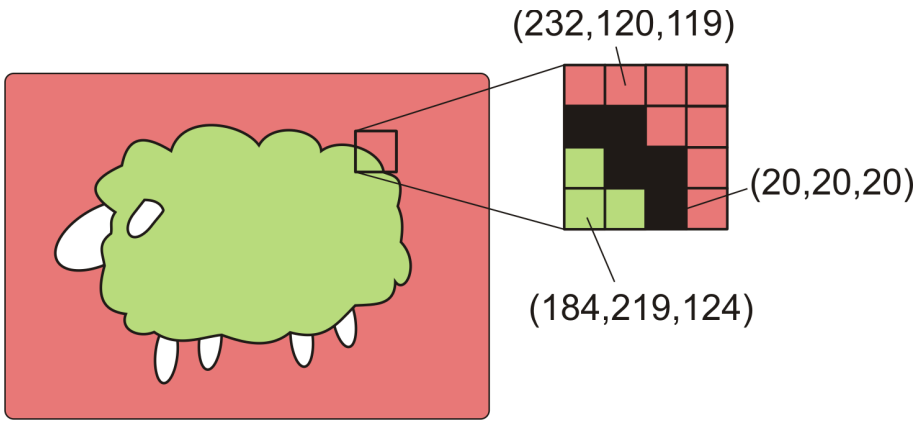


Figure 2-5: A digital image represented with a standard, LDR encoding. The RGB color channels of each pixel are specified with 8 bit integer values.

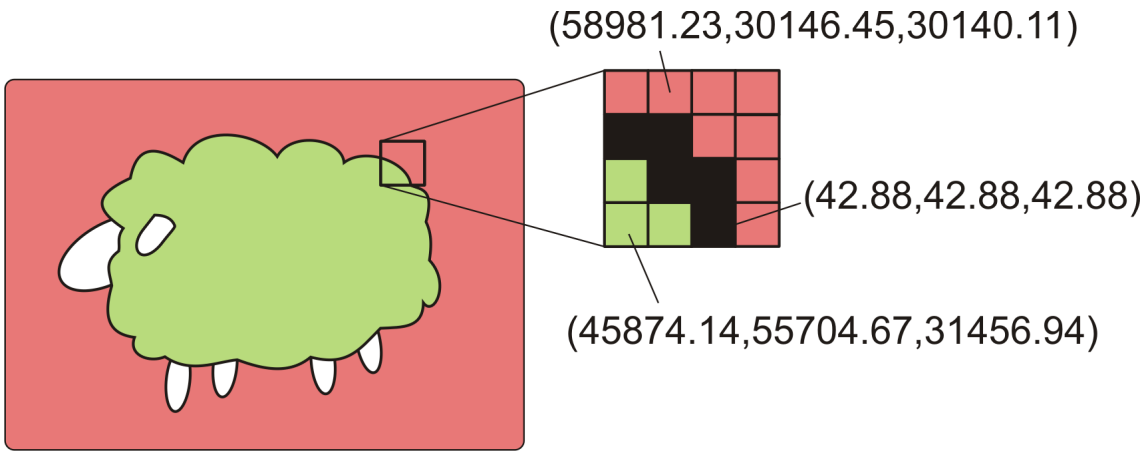


Figure 2-6: A digital image encoding using a HDR image format. The RGB color channels of each pixel are specified with high precision, floating point values.

There are a number of approaches to capturing HDR images. One common method is to capture a series of LDR images of a scene at different exposures, and consolidate them into a single, HDR scene representation. Another is to use modern, high-fidelity hardware to directly capture a HDR representation of a real-world scene. For more information on HDR image capture, see [RWP+06].

2.1.4 HDR Rendering

Similary to traditional imaging, in computer graphics, to adhere to display device brightness limitations, light sources are typically clamped to a luminance range of $[0, 1]$, where 0 represent complete absence of light and 1 represents maximum brightess. Within this range, intensity can be specified with 8 bits of precision. This is greatly inhibitive for lighting computations; using this system, a candle, a car headlight and the sun are all

clamped to 1, and are consequently considered equally bright in the simulation! Furthermore, due to the lack of precision, during lighting computation, it is difficult to distinguish between light sources of similiar intensity, thus losing fine detail in the final scene.

An NVIDIA presentation [GC] aptly formulated the following points, which are used here to relate to real-world scenery:

- "Bright things can be really bright.
- Dark things can be really dark.
- There is detail in both."

To facilitate lighting computation more representative of the real world, *HDR Rendering* removes the artificial $[0, 1]$ intensity restriction and increases bit depth per color channel to 16-, 24- or 32- bits, depending on the target application. This allows light sources to be modeled after their real-world equivalents. For example, a candle could be given a brightness of 300, a battery operated flashlight 1000 and a car headlight 10000.

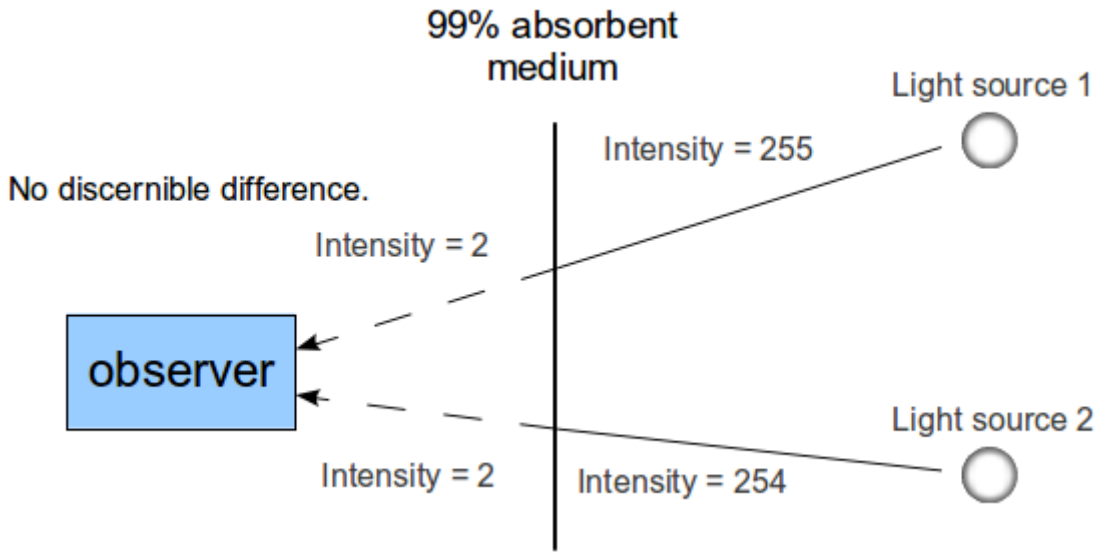
By representing intensity with unrestricted, high precision floating point values, far more accurate lighting computations are possible. Consider the following simulation. Two light sources emit light of similar intensity that passes through a 99% absorbent medium before reaching an observer. Figure 2-7 (a) depicts this situation in traditional, LDR Rendering. For simplicity of illustration, the 8 bit intensity range is represented with integers in the range $[0, 255]$. Although the light sources have different intensities, after passing through the absorbent medium, due to lack precision, the intensities of both light sources are both truncated to 2 and are no longer be discriminated from each other. Figure 2-7 (b) shows the same situation using HDR Rendering. In this case, the light source intensities now represented with full precision floating point numbers. After passing through the medium, both light sources may still be distinguished from each other. Furthermore, since the light source intensities model very bright, real-world entities, the light on the other side of the medium can still be recognized as "bright".

2.1.5 The tone mapping problem

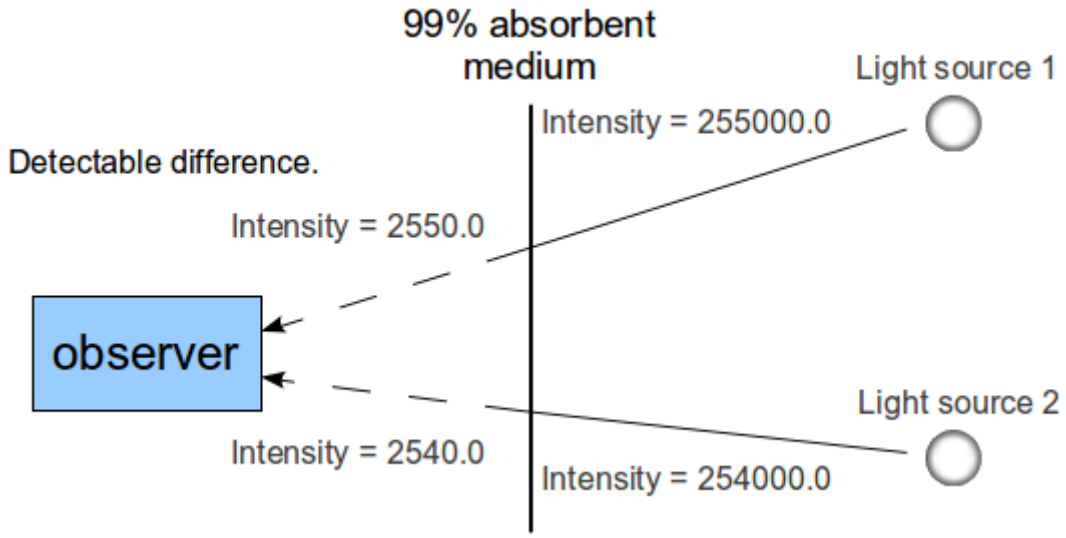
Regardless of the dynamic range used for encoding a digital image or generating a virtual scene in computer graphics, the result will ultimately be shown on an LDR display device. Therefore, until sufficient progress is made in the development of HDR display devices, mapping HDR output pixel data into LDR before sending it to the display device is unavoidable.

The question is, how can HDR pixels be compressed to LDR? Simply clamping all HDR pixels to the displayable range would map most pixels to pure black or white, resulting in a scene similar to that on the left in Figure 2-8. An intuative approach could be to compress all luminances by linearly scaling them against the miminum and maximum values in the image. This would result in a scene similar to that on the right in Figure 2-8. In both cases, a mapping from HDR to LDR has been performed. The results, however, are most probably not desirable for the majority of applications.

Typically, the mapping from HDR to LDR should be performed in such a way that the result visually matches the scene the scene being represented [RWP+06, p. 187]. This is

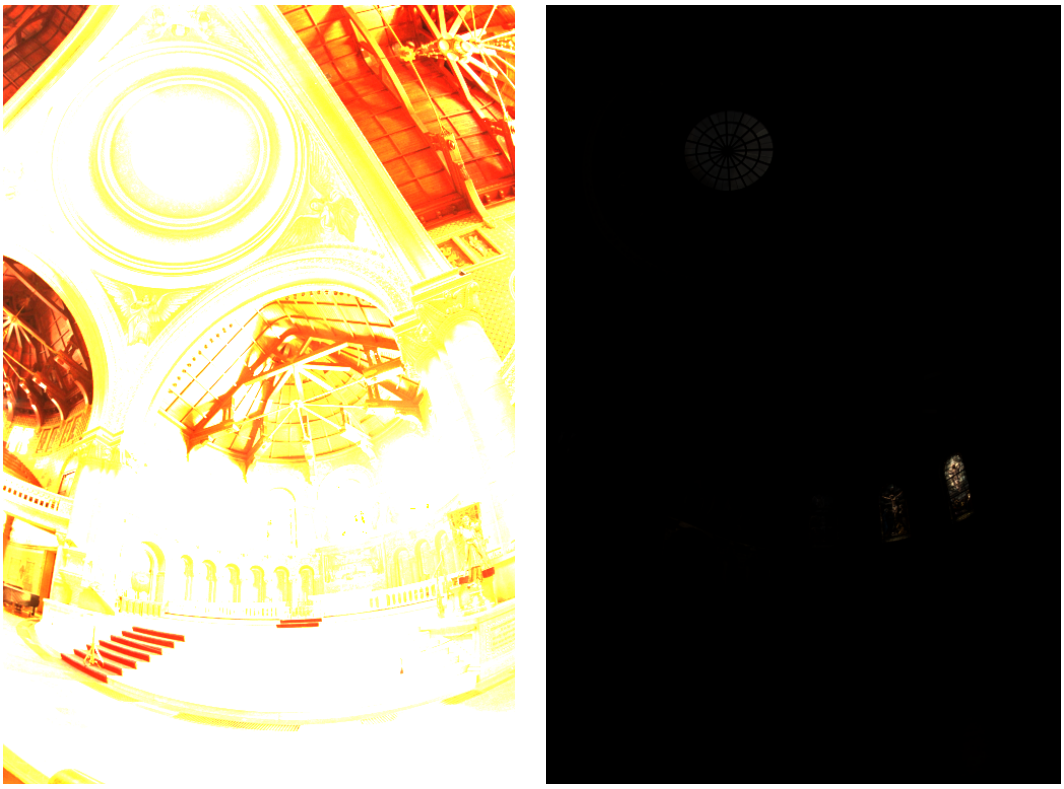


(a)



(b)

Figure 2-7: The difference in accurracy between LDR Rendering (a) and HDR Rendering (b). In both cases, two light sources of similar intensity are viewed through a 99% absorbent medium. In the LDR case, due to 8 bit intensity representation, the observer is unable to detect any difference between the two light sources. In the HDR case, intensity is represented using unrestricted, floating point numbers. As a result, the observer can detect an intensity difference between the two light sources.



Clamping Linear scaling

Figure 2-8: Two attempts at mapping the "Memorial Church" HDR radiance map (see Appendix A) to LDR. The most simple solution, on the left, clamps HDR into LDR. Another simple idea, shown on the right, is to scale luminance linearly against largest and smallest values in the image. In both cases, undesirable results are attained.

known as the *tone mapping problem*.

There is a large body of research on the tone mapping problem. An algorithm that solves the tone mapping problem is called a *tone mapping operator*. The tone mapping operators proposed in the literature can generally be classified as *global* or *local* operators[1].

Global operators compress the luminance of each pixel independently of its surroundings. Using a global operator, a dark pixel surrounded by other dark pixels is compressed in exactly the same way as a dark pixel located in a bright region. Global operators are generally efficient and highly parallelizable. The penalty of their performance benifits, however, is that detail, particulalry in regions of high contrast, is lost.

Local operators customize the compression process of each pixel to its local surroundings. By accounting for the local properties of each pixel, significant scene detail can be preserved. Individually examining each pixel's local neighbourhood during tone mapping, however, is computationally expensive. Local operators are therefore seldom used with

[1]This is true within the group of *spatial operators*, i.e operators that operate on the spatial domain. Tone mapping operators have also been proposed for the frequency and gradient domains [RWP+06]. However, since the most efficient operators for GPU implementation found during the literature review were all spatial operators, the focus of this work is on tone mapping in the spatial domain.

interactive applications.

The results of global and local tone mapping are compared in Figure 2-9. For coverage of the fundamental principles of tone mapping, see Chapter 4, which presents a detailed case study of a commonly used tone mapping operator.

Global Operator

Local Operator

Figure 2-9: Comparison of global and local tone mapping operators, applied to the HDR scene "Snow" (for radiance map source, see Appendix A). Detail in the snow is preserved better by the local operator.

2.2 Image processing

This thesis uses some basic concepts from the field of *image processing*. Image processing the science of a analysing and manipulating *digital images*. There is a large body of research and literature on the topic of image processing; this section humbly aims to provide a simplified view of the concepts referred to in later chapters. For a more comprehensive introduction to image processing, see [Tön05], [Bov08], [GV97].

2.2.1 Introduction to digital images

The fundamental structure in image processing is a *digital image*. Digital images can be represented in many forms; in this thesis, two dimensional (2D) images are of primary interest. A 2D image is simply a matrix of n-tuples, with $n \geq 1$. Each matrix entry is refered to as a *pixel*. Figure 2-10 illustrates this concept, where each pixel is represented by the n-tuple $v(x, y)$.

Two particularly common image types are *greyscale* and *color* images. The distinguishing factor between these image types is length of v. In greyscale images $|v| = 1$. In other words, each pixel is represented by a *scalar*, which specifies its intensity, typically visualized by grey value on gradient between black and white, known as a *greyscale*. In color images, each pixel is typicall represented by a 4-tuple, where each value in the tuple is referred to as a *channel*. The first three channels specify the intensity of the red, green and blue primary components used in linear combination to specify the colour to use when visualizing the pixel (see Section 2.1 for more information on colours). The fourth channel is called the *alpha* channel, and specifies pixel transparency. The number of bits used to specify each channel is called the image's *bit-depth*. The images on the left and right in Figure 2-11 are visualization of greyscale and colour images, respectively.

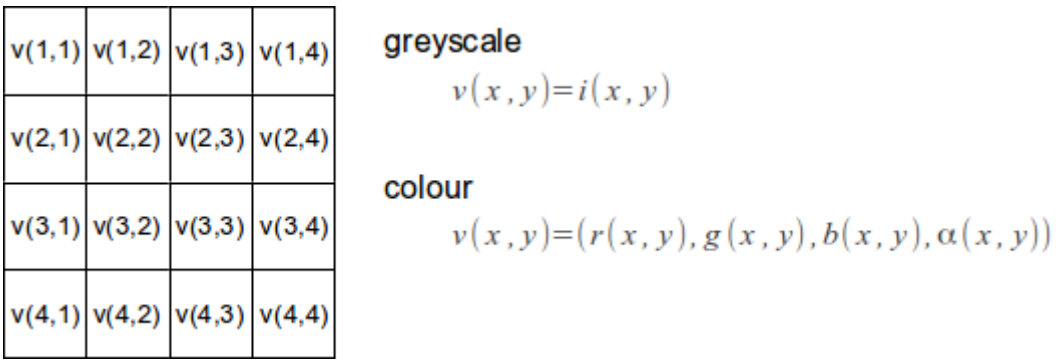


Figure 2-10: A digital image is a matrix of n-tuples. Two common types of images are greyscale images, where each matrix element is single intensity scalar, and colour images, where each matrix element is a vector specifying colour channel intensities and transparency.

Figure 2-11: Example of greyscale (left) and colour (right) image representations.

2.2.2 Image convolution

A digital image can be viewed as a discrete digital signal. As such, it can be filtered using a common operation in signal theory called *convolution*, a process known as *image convolution*. In the context of this thesis, image convolution can be seen as a modification of each pixel in an image by computing the scalar product of the union of the pixel and its local neighbourhood with a matrix of filter weights called a *filter kernel*. Figure 2-12 illustrates this process for a 3x3 filter kernel.

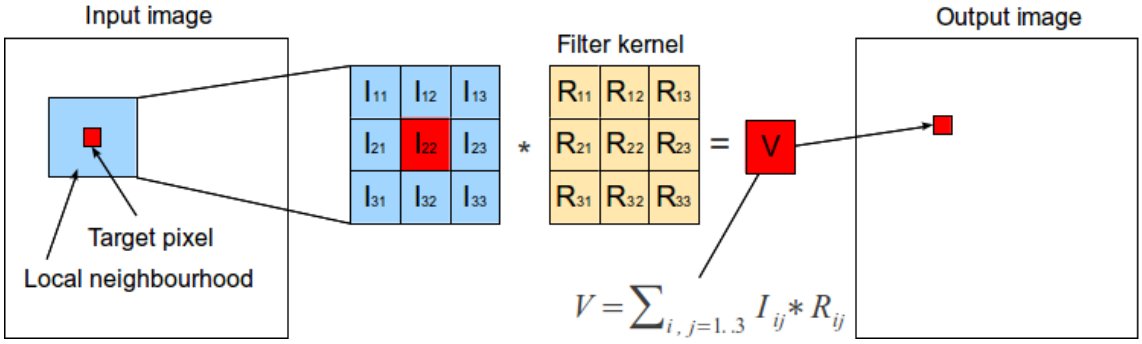


Figure 2-12: An example of image convolution. The target pixel (red) is convoluted by computing the scalar product of the region containing the pixel and its local neighbourhood (blue) with a 3x3 fitler kernel (orange). The result is written to an output image.

The process of convolving an image I by a kernel R to produce a result, or *response*, V is often denoted:

$$V = I \otimes R \tag{2.2}$$

This concept will be clarified in the following subsections, which describe the image convolutions used in this thesis.

2.2.2.1 Box filtering

Box filtering computes an equally weighted average of a given region around each pixel in an image. To apply a box filter to an image, the image is convolved by filter kernel with all elements set to $1/(n^2)$, where n is the length of one side of the filter. Figure 2-13 shows an example of computing a 3x3 box filter.

The result of applying a box filter to a digital image is a uniform blur. Figure 2-14 shows the effects of box filtering a sample image by box kernels of different dimensions.

2.2.2.2 Gaussian convolution

Like box filtering, Gaussian convolution computes the local average around each pixel. Unlike box filtering, however, the weightings in the filter kernel are not uniform. Instead filter weightings are determined by a Gaussian distribution, defined as:

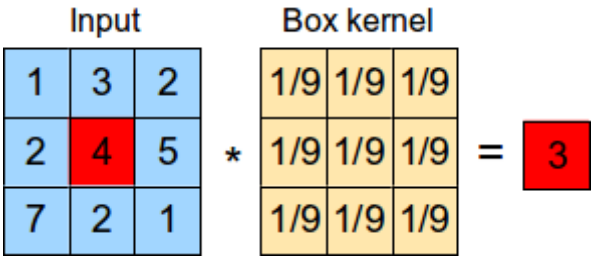


Figure 2-13: An example of computing a box filter.

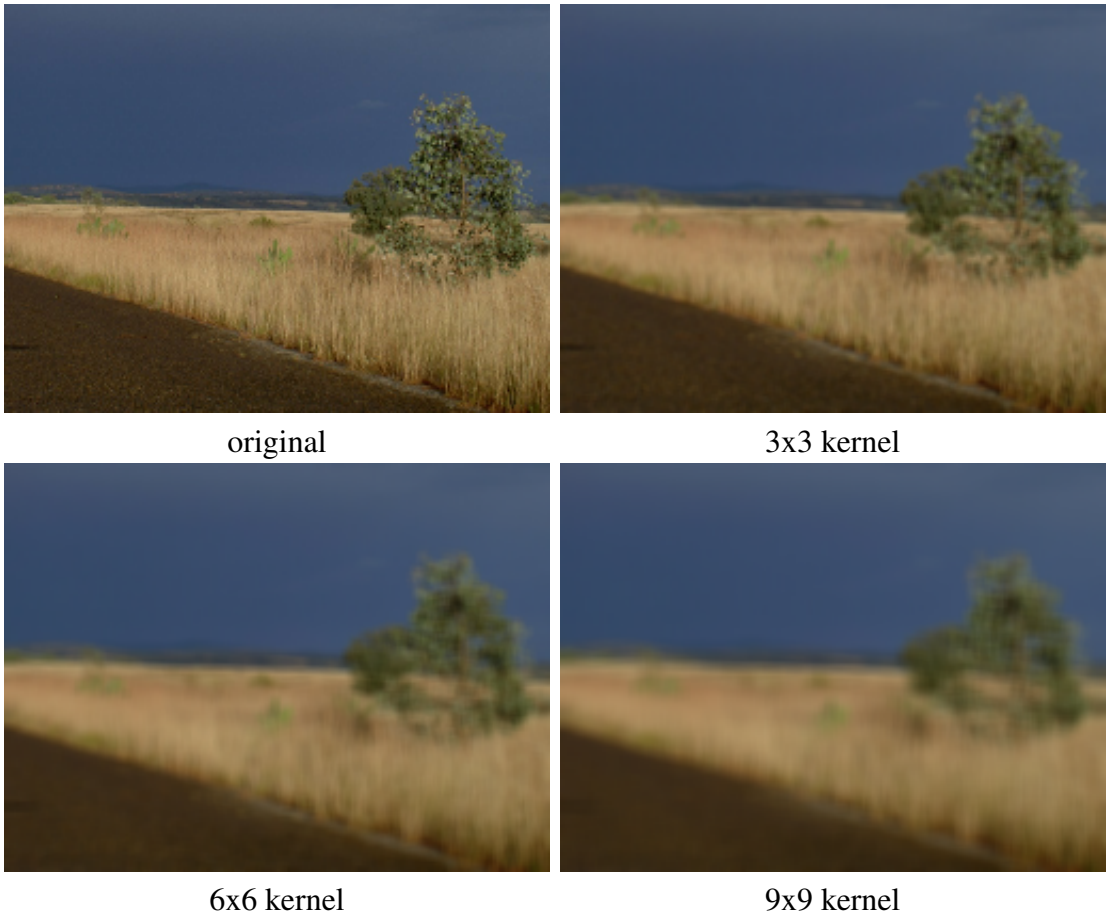


Figure 2-14: The effects of box filtering an image with a variety of kernel sizes.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\left(x^2+y^2\right)/2\sigma^2} \tag{2.3}$$

where x and y are the x and y distances from an origin at the filter kernel center, σ is the standard deviation, also known as the kernel *radius*, and $G(x,y)$ is the resulting pixel weighting. Figure 2-15 shows a plot of an example Gaussian distribution. It can be seen that the kernel weightings decrease as the distance from the origin (kernel center) increases. As a result, the blurring effect achieved by Gaussian convolution is smoother. Figure 2-16 shows the results of Gaussian blurring on a digital image.

Seperability
Convolving an image by an mxn Gaussian kernel requires mxn multiplications for each

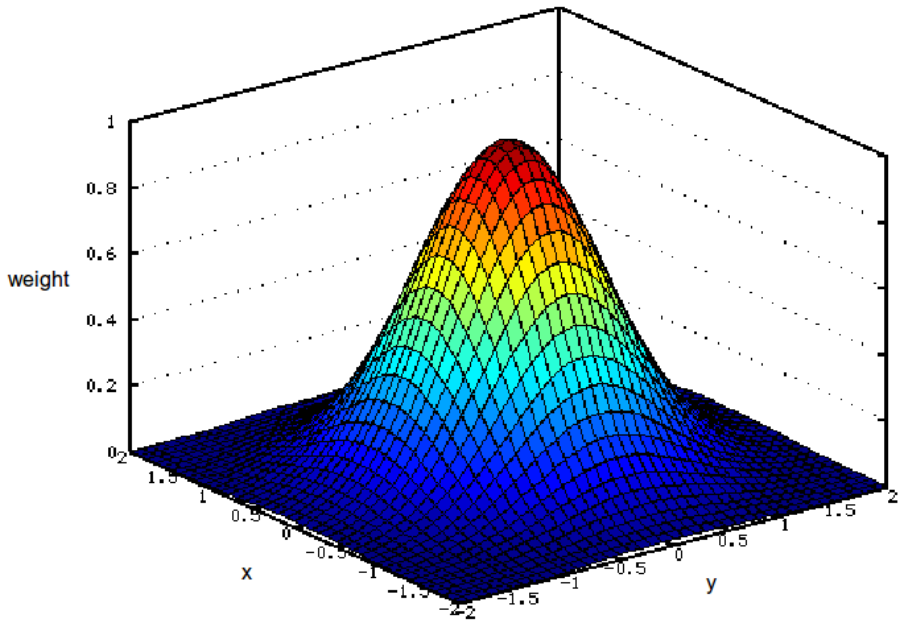


Figure 2-15: A plot of the weighting of a Gaussian kernel. Filter weights decrease as the distance from the origin increases.

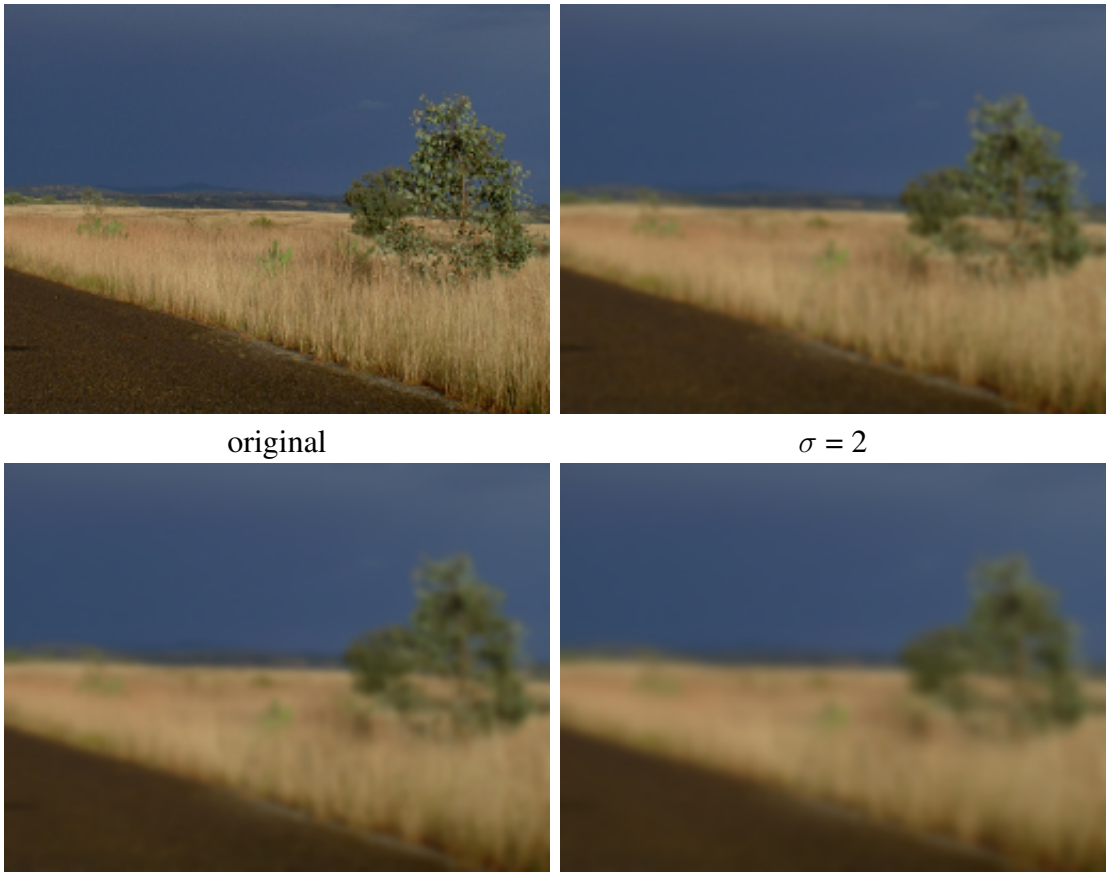


Figure 2-16: The effects of Gaussian convolution, applied to an image with a variety of kernel radii.

pixel in the image. Therefore, the amount of work required per pixel scales by $O(n^2)$. This is an expensive operation.

Fortunately, Gaussian kernels are *seperable*. This means that a mxn kernel matrix with weights given by the 2D Gaussian distribution specified by Equation 2.3 can be replaced by two, 1D vector kernels, of length m and n, with weights specified by the 1D Gaussian distribution [Tön05]:

$$G(x) = \frac{1}{2\pi\sigma^2} e^{-x^2/2\sigma^2} \tag{2.4}$$

By applying the two mx1 and 1xn 1D convolutions consecutively to the input image, the same result will be obtained as if it were convoluted by the original mxn matrix, with a total of m+n multiplications. Therefore, by exploiting the seperability of Gaussian kernels, the asymptotic complexity of convoluting each image pixel is reduced to $O(n)$.

2.2.2.3 Efficient box filtering with Summed-Area Tables

Like the Gaussian kernel, a box filter kernel is seperable. Therefore, the per pixel complexity of convolution can be reduced from $O(n^2)$ to $O(n)$. It is possible, however, to further optimize this operation to constant complexity ($O(1)$) per pixel.

Constant per-pixel box filtering complexity is achieved with the help of a special data structure called a *Summed-Area Table* (SAT) [Cro84]. A SAT S is a table computed from a input image I such that each element $S(i, j)$ in S contains the sum of all elements $I(p, q)$, with $p \leq i$ and $q \leq j$ [SO08]. Formally, from an mxn image I, a SAT of equivalent dimension is constructed, where each entry in S is given by:

$$S(i, j) = \sum_{p=1}^{i} \sum_{q=1}^{j} I(p, q) \tag{2.5}$$

Figure 2-17 shows a SAT constructed from an example 4x4 input image.

original

1	4	0	2
0	2	1	5
3	1	4	2
4	7	0	3

SAT

1	5	5	7
1	7	8	15
4	11	16	25
8	22	27	39

Figure 2-17: An example 4x4 image and corresponding SAT. Image source [SO08].

Using an image's SAT, the sum of all pixels within any rectangular image region can be computed with four fetches. For example, to compute the sum of the blue region in the example 4x4 image on the left in Figure 2-18, the delimiting elements A, B, C and D marked in the image's corresponding SAT on right are summed:

$$Sum = A - B - C + D \tag{2.6}$$

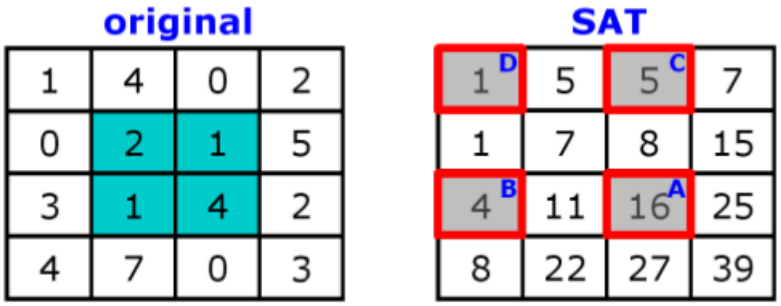


Figure 2-18: Computing the average of the region marked in blue by applying Equations 2.6 and 2.7 to the elements marked in the SAT on the right. Image source [SO08].

The average of the region can be obtained by computing:

$$Average = \frac{Sum}{w' * h'} \tag{2.7}$$

where w' and h' are the region dimensions.

Since box filtering equates to computing a series of region averages in an image, given an image and its corresponding SAT, a box filter convolution can be applied with constant complexity per pixel.

Chapter 5 investigates efficient parallel methods for generating SATs on the GPU.

2.3 GPU Concepts

This section discusses concepts related to programming *Graphics Processing Units* (GPUs). GPUs are powerful and cheap computing devices present in each PC, laptop and workstation, as well as many portable devices. Originally designed to execute the fixed-functionality graphics pipeline (Section 2.3.1), they have evolved over the years to become fully programmable, parallel processing tools.

2.3.1 Shader programming

GPUs were originally designed to generate 2D and 3D graphics for display on the computer monitor. For many graphics applications, this was accomplished by mapping mathematical scenes, represented by theoretical constructs such as points or polygons, via a sequence of transformations in GPU hardware to an array of pixels for display on the screen. This set of transformations was known as the *fixed-functionality graphics pipeline*, since the functionality of each transformation was hardwired into the GPU harware.

With each new generation of GPUs, the fixed-functionality graphics pipeline became increasingly configurable to meet the evolving demands of developers. In the early

2000s, key stages of the graphics pipeline became fully programmable, leading to the *programmable graphics pipeline*.

Figure 2-19 shows a schematic of the programmable graphics pipeline. Vertices from the virtual scene are processed by the **Vertex Processor**, which transforms them from their local coordinate systems to consistent, global coordinate system. The transformed vertices, together with their associated primitives are then clipped against the virtual camera's view frustrum by the **Clipper and Primitive Assembler**, which also performs occlusion operations. Once the visible primitive in the global coordinate system have been identified, the **Rasterizer** converts the visible vertex and primitive information to *fragments*, which are essentially uncoloured pixels that are still not guarenteed to make it onto the screen. Fragments are processed by the **Fragment Processor**, which assigns them colour. The output of the fragment shader is a set of pixels, which are shown on the display device. For more information on the basics of computer graphics, see [Ang11].

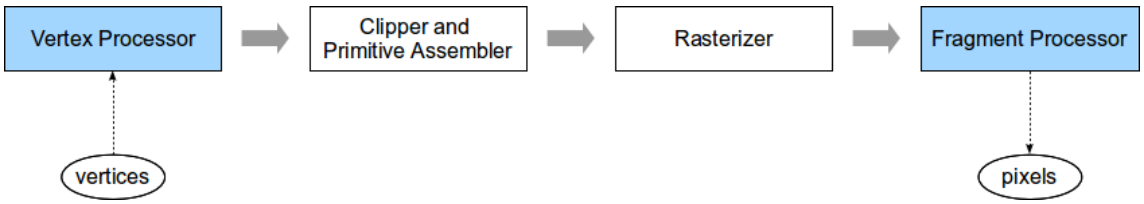


Figure 2-19: The programmable graphics pipeline. Programmable stages are marked in blue.

The vertex and fragment shaders shown in Figure 2-19 are *programmable*, which means that developers can replace their standard functionality with custom procedures. This opens many possibilities; for example, a blur effect can be added the output scene by programming the fragment shader to apply one of the image convolutions described in Section 2.2.

There are a number of languages specified for programming shaders, which are called *shader programming languages*. Some of the most common of these are: Microsoft's *High Level Shader Language* (HLSL), NVIDIA's *C for graphics* (Cg) and the *OpenGL Shading Language* (GLSL). Shader languages are special purpose languages, designed specifically for manipulating vertices and fragments.

2.3.2 General-Purpose GPU (GPGPU) programming

GPUs are designed for processing primitives and producing pixels for display on the screen. Many of the applications for which they are used have real time performance requirements, which means they have to produce images at up to 60 frames per second, for increasingly high resolutions. Furthermore, the problems that GPUs are designed to solve are very specific and highly parallelizable. As a result, GPUs compute using the *Single Instruction Multiple Data* (SIMD) model, where a collection of specialized, high-performance Streaming Processors (SPs) perform the same instructions on multiple data simultaneously.

Due to their specialized, high performance, parallel architecture, GPUs have become extremely powerful computing devices. Figure 2-20 compares the floating point operations per second and memory throughput of a number of modern GPUs to CPUs.

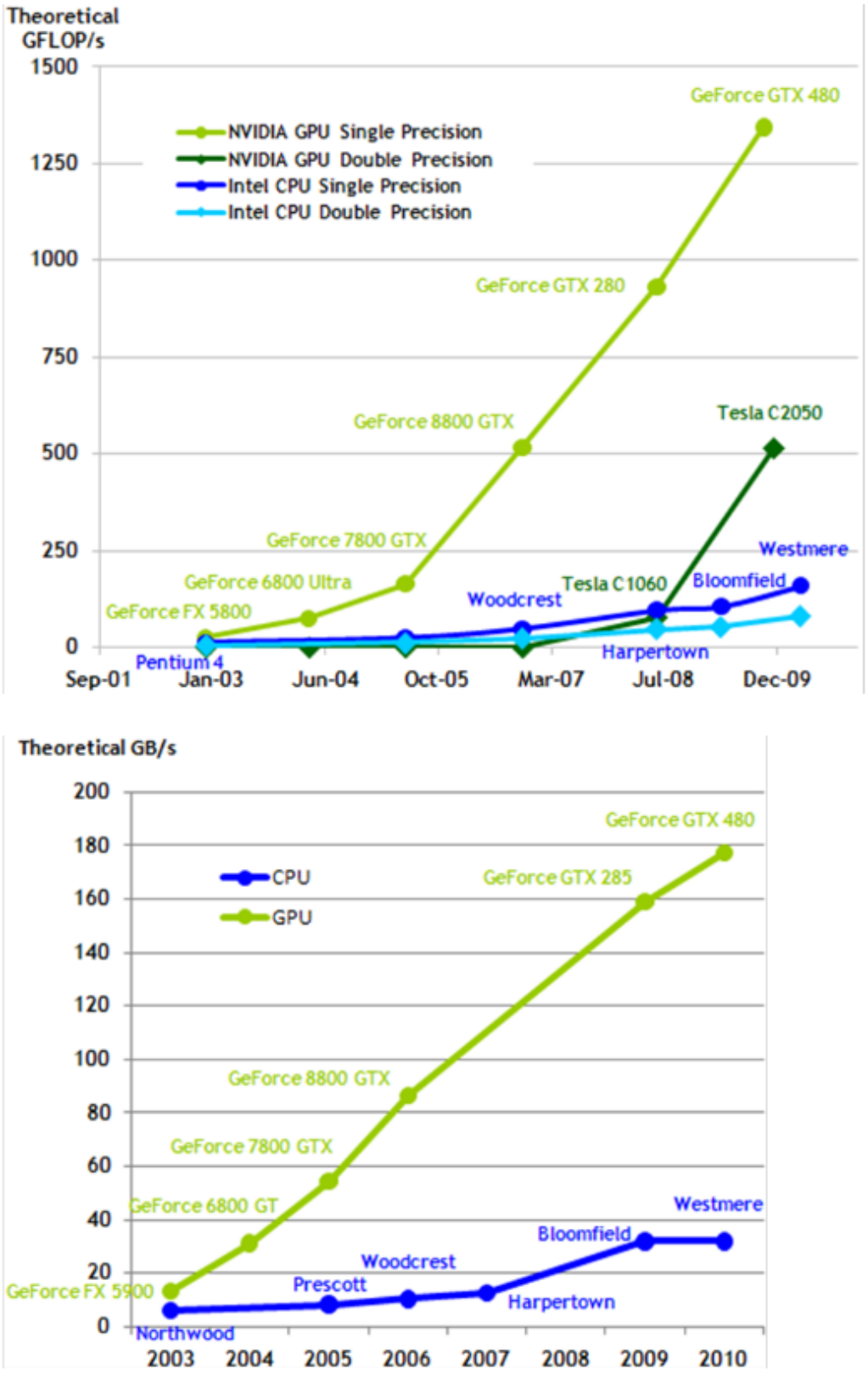


Figure 2-20: Comparison of Floating-Point Operations per Second and memory bandwidth of recent GPUs and CPUs. Image from the NVIDIA CUDA Programming Guide [NVI11].

Motivated by the processing power offered by GPUs, developers, particulalry the scientific community, began using GPUs to solve general-purpose problems unrelated to computer graphics. This was often awkward, however, because it was necessary to formulate problems in terms of graphics-related concepts, such as vertices, textures and fragments, in order to solve them with shader languages. Using the GPU to solve general-purpose, non-graphical problems is known as *General-Purpose GPU (GPGPU) programming*.

In late 2006, NVIDIA introduced the *Compute Unified Device Architechture* (CUDA) - a scalable, parallel programming model for GPUs, that allows programmers to bypass the graphics API and solve general-purpose problems on the GPU directly in C or C++.

2.3.3 The CUDA programming model

CUDA is an extension of the C and C++ languages. A CUDA program is specified as a special C or C++ function called a *kernel*. Kernels are executed concurrently as a collection of parallel *threads*. Groups of threads are organized as *blocks* and blocks in turn comprise a *grid*. Threads within a block cooperate using barrier synchronization primitives and common access to a shared memory space exclusive to each block. A grid is a set of independently executing thread blocks. Figure 2-21 illustrates this concept.

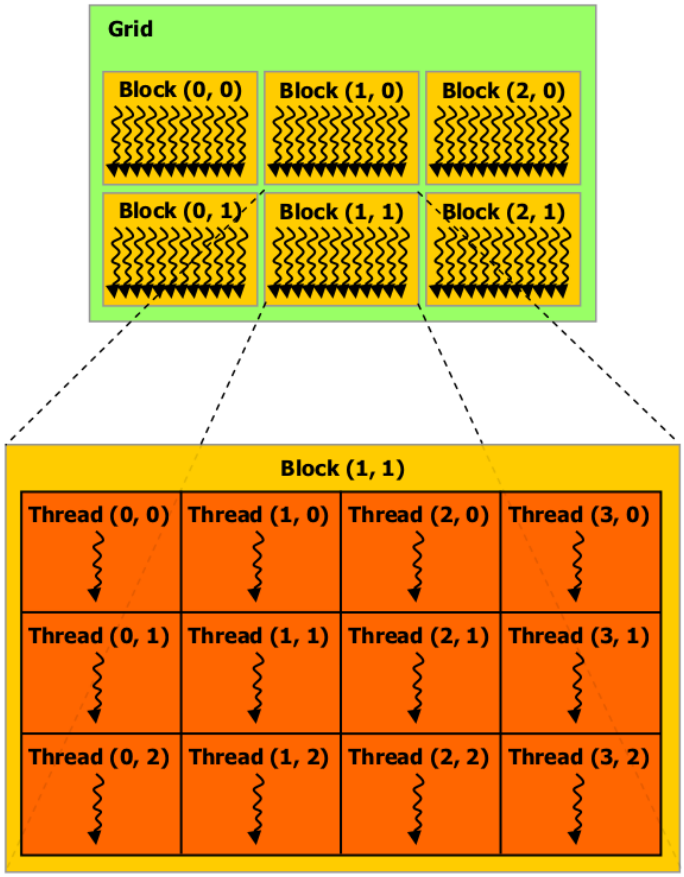


Figure 2-21: CUDA thread heirarchy [NVI11].

The dimensions of the processing grid, as well as of each block, are specified when the kernel is invoked. Typically, a kernel is invoked such that there is one thread for each

element of data to be processed. For example, when using CUDA for image processing, the thread grid is set up with a number of threads commensurate with the number of image pixels, and each thread is mapped to an input pixel by matching its position within the grid to the pixel's position within the image.

2.3.3.1 Memory heirarchy

The CUDA memory heirarchy is shown in Figure 2-22. Threads have access to multiple memory spaces during their execution. Each thread has set of local registers and a private local memory space that it uses for storing local variable and computing local results. Within each block, each thread has access to a common, shared memory space called *shared memory*. Shared memory is only slightly slower than registers. Finally, all threads in the system have access to a global, high-latency memory space called *global memory*.

Registers and shared memory are *on-chip* memory, while gobal memory is *off-chip*. On-chip means that the memory is locate directly on the chip with the processing unit. Off-chip memory is stored in a random access DRAM memory unit, which resides off the processor chip, so global memory requests must be fetched across a bus. As a result, registers and shared memory are very fast but with small capacity, and global memory is large but slow.

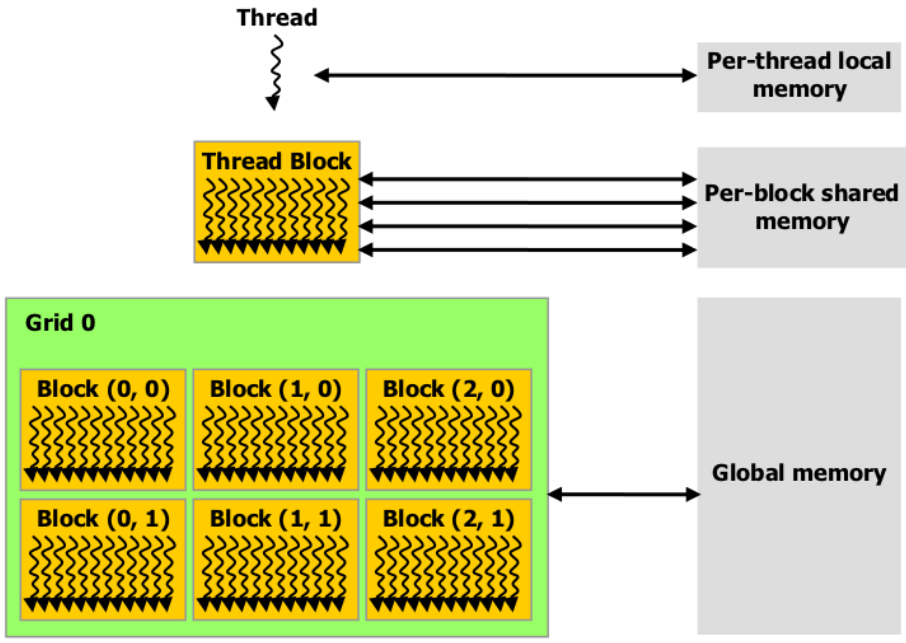


Figure 2-22: CUDA memory heirarchy. Adapted from [NVI11].

There are two additional, read-only memory types available to all threads: *texture memory* and *constant memory*. Constant memory resides in the global memory space, and, unlike global memory, is *cached* on chip. Texture memory is similarly cached, where the caching procedure is optimized for 2D *spatial locality*, i.e data located close together in 2D will be cached.

A summary of the memory types is given in Figure 2-23.

Memory	Cached	Access	Scope	Capacity
Register	No	Read/Write	Single thread	Bytes
Shared	No	Read/Write	Block	Kilobytes
Global	No	Read/Write	All threads	Mega-/Gigabytes
Constant	Yes	Read	All threads	Mega-/Gigabytes
Texture	Yes (spatially)	Read	All threads	Mega-/Gigabytes

Figure 2-23: Summary of CUDA memory spaces.

2.3.3.2 Performance considerations

Avoiding thread divergence
Within each block, threads are further logically divided into *warps*, which consist of 32 threads each. All threads within a warp execute in lock-step, i.e all 32 threads execute the same instruction at the same time, operating on different data. If threads of warp diverge on a data-dependent branch, the warp must be re-executed for each executional path taken, each time disabling all threads not following that path. In a very unfortunate case, the warp would be re-executed 32 times, thereby losing all parallelism. This situation is known as *warp divergence*.

Coalescing global memory access
Global memory requests are serviced via 32-, 64-, or 128-byte memory transactions. When the threads of same half-warp (16 threads executing in lock-step) issue a memory request, their collective request can be satisfied with a single memory transaction, provided that:

- The request size per thread is 4, 8 or 16 bytes. 4 byte requests will be serviced with a single 64 byte transaction; 8 byte requests with a single 128 byte transaction and 16 bytes requests with two 128 byte requests.
- All threads are accessing memory within a segement of global memory whose starting address is a multiple of the transaction size.
- Threads request words in order: thread k in the half-warp requests word k in transaction.

The procedure of optimizing the response to the requests of a half-warp into a single memory transaction is known as memory *coalescing*. If the requirements above are not met, coalescing will not take place and the half-warp will be serviced with 16 seperate, 32-byte memory transactions.

It should be noted that more recent GPU-capable GPU architechtures more relaxed in their coalescing requirements. However, since the GPU used for the majority of development and testing in this thesis is one of the earlier models (see Appendix A), all of the conditions above apply.

Avoiding shared memory bank conflicts
To achieve high bandwidth, shared memory is organized into 16 equally sized modules, called *banks*, where sequential 32-bit words are organized into sequential banks. Memory requests that fall into seperate banks can be serviced simultaneously. However, if n simultaneous requests fall into the same bank, these requests must be serviced sequentially.

This is known as an n-way bank conflict. Bank conflicts can only occur within each half-warp. Therefore, it is important to design shared memory requests in a half-warp to fall into seperate banks.

3 Requirements Specification

The long term goal of this research is to enable HDR Rendering in Virtual Night Drive. To accomplish this, a suitable tone mapping operator must be conceived, which can meet the rendering requirements implied by VND.

This chapter begins, in Section 3.1, by examining the current LDR rendering process incorporated by VND and highlighting necessary modifications required for more accurate, HDR headlight simulation. Section 3.2 discusses the requirements on the tone mapping operator to be developed in this thesis in the context of its intended future integration into VND.

3.1 Enabling HDR Rendering in VND

At present, the headlight prototyping procedure used by VND consists of two major stages: one preprocessing step, performed by an offline converter, and an interactive simulation, which uses the output of the offline converter to illuminate the virtual environment as the user guides a vehicle about a test track. This procedure is overviewed in Figure 3-1.

The VND rendering process in Figure 3-1 is divided into major phases, and the location of each phase within the overall simulation is indicated by the color of the diagram region in which it is drawn. Furthermore, the dynamic range of data that each phase operates on is specified in the far right column of the diagram.

Phase 1 in Figure 3-1, performed by an offline converter, involves processing and simplifying the raw headlight data provided by the client headlight manufacturer, saving the results into a collection of textures suitable for interactive projection into a virtual environment. The data passed to the converter is a map of headlight beam pattern intensity data containing luminances of up to $10^5 cd/m^2$. This HDR data is tone mapped as a part of the preprocessing process, producing a set of LDR textures that can easily be used for projection and lighting computation in the interactive driving simulation.

Phases 2 and 3, shown in the bottom, light grey region in Figure 3-1, are performed during the interactive driving simulation. Phase 2, performed on the GPU using a vertex shader, projects the LDR headlight textures produced by Phase 1 from each virtual car headlight onto test track. This projected region is illuminated in Phase 3, using fragment shaders, according to the LDR illumination data stored in the textures produced by Phase 1.

It would be desirable to remove tone mapping from the preprocessing step and use the original, HDR headlight data for environment illumination, allowing more accurate lighting computations. Illuminating the virtual test track with HDR beam pattern data will result in HDR output from the illumination fragment shader (Phase 3 in Figure 3-1). In order to display this frame data, tone mapping must be performed as an additional, post-

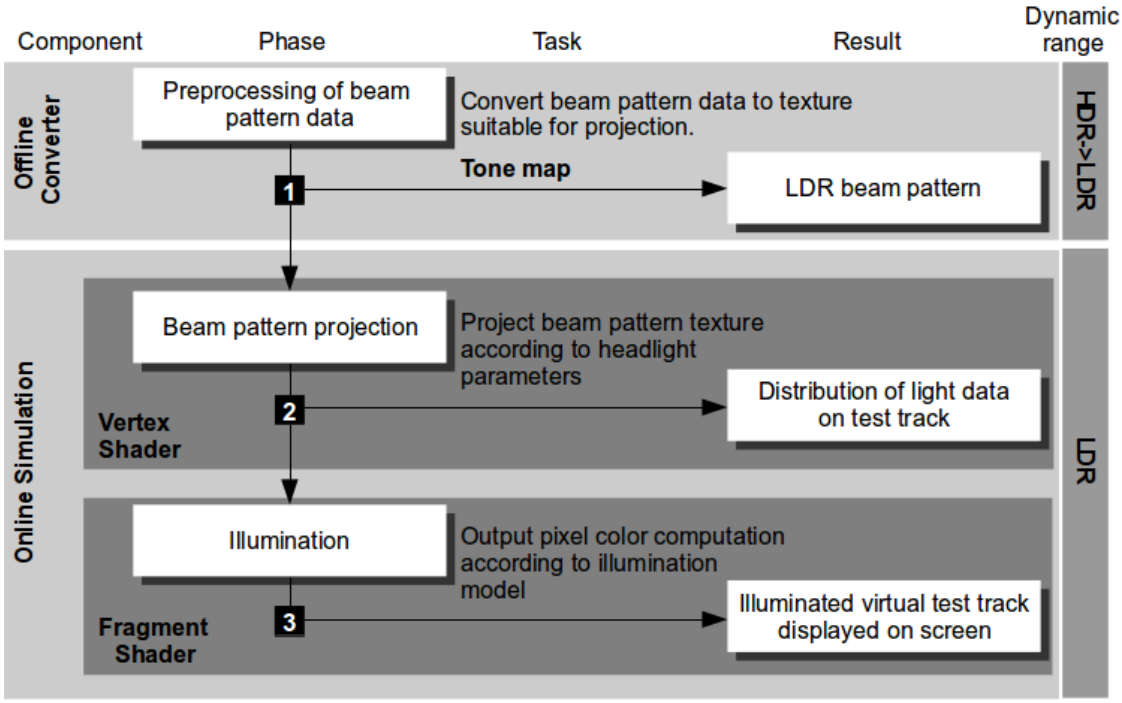


Figure 3-1: The current rendering process in VND. Input HDR beam pattern data is compressed into LDR format as a preprocessing step. As a result, only LDR illumination is performed during simulation. Figure inspired by [BER05, p. 78].

processing phase. This situation is shown in Figure 3-2. Because each pixel produced by the illumination fragment shader (Phase 3) must be tone mapped, and this process must take place as rapidly as possible, Phase 4 in Figure 3-2 should be implemented on the GPU.

3.2 Tone mapping requirements

The goal of this thesis is to implement a tone mapping operator that can, in future, be used in Phase 4 in Figure 3-2. Consequently, it is important that the use of this operator does not affect VND's ability to meet its original requirements. Therefore, the requirements on the tone mapping operator can be extrapolated from those of VND.

Four key requirements on the tone mapping operator have been identified. These requirements are specified below.

Requirement 1: Preservation of local details in high contrast regions
Driving simulation in VND takes place in a night time environment, where the only source of light in the scene is that of the test vehicle's headlights. Projecting the original, HDR beam pattern data into the environment will lead to a very high contrast range in the resulting scene. In particular, when a headlight itself becomes visible in the view frustum, such as when the test vehicle is approached by oncoming traffic, the dynamic range of the scene can reach 5 orders of magnitude.

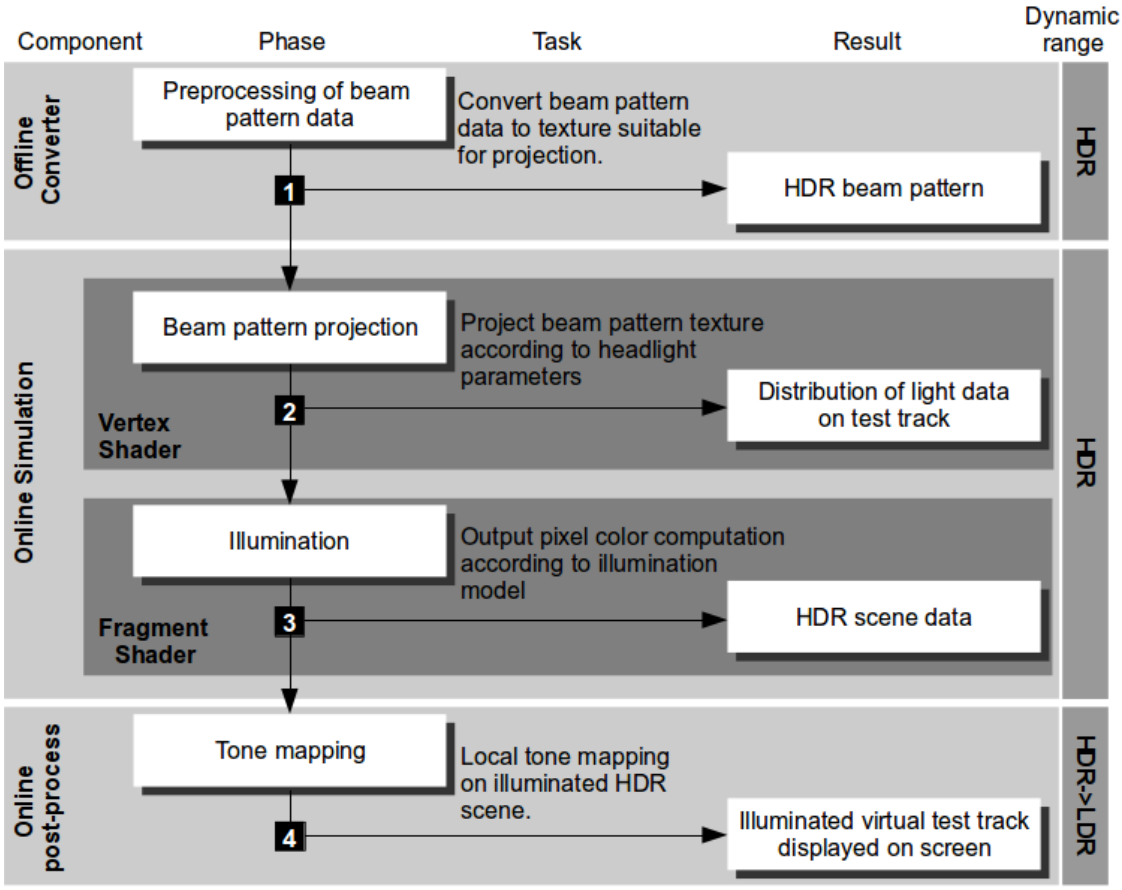


Figure 3-2: To support HDR environment illumination in VND, the tone mapping operation originally performed in Phase 1 (Figure 3-1) must take place in an additional, post-processing module, which tone maps each output frame in real time.

Because VND is designed for headlight protyping, it is important that the fine detail in the environment illumination caused by any given headlight prototype are clear, even in very high contrast scenes. Since global tone mapping operators notoriously favour performance at the expense of local detail in high contrast regions, this suggests that the appropriate operator for VND will be *local*.

Requirement 2: Interactive frame rates at high resolutions
VND is designed to be run on two classes of platforms. The first is that of standard PCs equipped with mid-range GPUs, which will typically run VND at resolutions up to 1920x1200. The second is that of high definition Virtual Reality visualization systems. In particular, VND has been designed to run in the *HD Visualisation Center* at the Heinz Nixdorf Institute. As described in Section A.2, the HD Visualisation Center is a system of interconnected computers that drive multiple high resolution projectors, which together create a highly immersive simulation enironment. The largest of the projectors generate images at a resolution of 3840x2160.

Since VND is an interactive application, the introduction of a tone mapping operator can not interfere with its interactivity, which entails maintaining a minimum frame rate

of 30 fps[1]. Therefore, the tone mapping implementation must be capable of supporting **interactive frame rates**, i.e a **minimum of 30 fps**, for the following configurations:

i. PC systems equipped with mid-range GPUs at resolutions up to **1920x1200**.

ii. The HD Visualization Center at a resolution of **3840x2160**.

Although 30 fps is the minimum frame rate required for interactivity, in order for VND to maintain its current high level of responsiveness, a framerate of 60 fps is necessary[2]. Therefore, the tone mapping operator should be able to maintain a **desired frame rate of 60 fps** on the system configurations i. and ii. specified above.

Clearly, this requirement directly contradicts Requirement 1; preserving detail in high contrast regions during the tone mapping requires advanced local adjustment at each pixel, penalizing performance. It is possible that the most suitable tone mapping operator for VND will need to find a reasonable balance between these requirements.

Requirement 3: Implementation as a self-contained post-processing module
Although the primary focus of this paper is developing a tone mapping operator for future integration into VND, there is no reason to restrict the area of application solely to VND. The growing trend towards using HDR in VR and other interactive applications [PB10] [YNC+06] [McT06] [Che08], coupled with continued progress in computing hardware capabilities, implies that there will be an increasing requirement for powerful, post-processing local tone mapping procedures. Therefore, the tone mapper developed in this paper should be implemented as a self-contained module, which can easily be integrated into existing interactive applications with tone mapping requirements, of which VND is but one instance.

Requirement 4: Efficient post-processing of OpenGL applications that use shaders
VND is based on OpenGL and makes heavy use of vertex and pixel shaders. Hence, however the tone mapping module is internally implemented, it must efficiently interoperate with OpenGL applications that use shaders. This requirement must be taken into consideration particularly when choosing a technology for implementation of the tone mapping module.

3.3 Summary

This chapter examined the current VND implementation and discussed the necessary conditions for introducing HDR rendering. With the intention of future integration into a future HDR version of VND, the requirements on a suitable tone mapping operator were specified. These requirements were inferred from those of VND.

[1] The definition of interactivity depends on the area of application. When interacting with Computer Aided Design (CAD) systems, for example, a frame rate of 5-10 fps is sufficient. Virtual Reality applications, on the other hand, aim to provide the user with a fluid impression of a dynamically changing virtual environment. To achieve this, a frame rate of 25-30 fps is necessary (motion pictures are typically shown on television at 25 fps).

[2] When driving through a virtual scene at high speed, the position of the viewer can change significantly in the 33 ms required to render each frame at 30 fps. This results in a reduction in simulation fluidity that is particularly noticeable on high resolution displays. By targeting a simulation rate of 60 fps or more, the limiting factor on rendering speed is shifted from the simulation itself to the refresh rate of common display devices.

The remaining chapters of this thesis investigate developing a tone mapping module that meets the requirements specified here.

4 Related Work

The tone mapping problem has been the focus of extensive research in recent years. During this time a multitude of global and local methods, operating in the spatial and frequency domains [RWP+06], of varying complexity, and with varying performance and visual results have been proposed. Despite the enormous variety, a property common to all methods is that global operators are much faster to compute, while local operators preserve more detail in high contrast image regions. Because preservation of high contrast detail is one of the requirements specified in Chapter 3, the focus of this research is on local tone mapping.

Some of the most widely cited local tone mapping algorithms are the *Fairchild iCAM method* [FJ02], *Pattanaik Multiscale Observer model* [PFF+99], *Ashikhmin Spatially Variant operator* [Ash02], *Pattanaik Gain Control method* [PY02], *Fattal Gradient Domain Compression method* [FLW02], *Tumblin and Turk Low Curvature Image Simplifier (LCIS) method* [TT99] and *Reinhard Photographic Tone Reproduction method* [RSS+02]. Due to their complexity, a review of these methods lies outside the scope of this thesis. For a comprehensive survey of the most common local tone mapping methods, see [RWP+06], [DCW+02].

Reinhard's Photographic Tone Reproduction method [RSS+02] has been chosen as the basis for this investigation, because its visual output consistently scores highly in evaluation studies [CWN+08] [LCT+05], it requires few parameters, and a number of recent optimizations and GPU adaptions show promising performance gains. Reinhard's local operator focuses on enhancing local contrast during compression, satisfying Requirement 1 in Chapter 3, and requires few parameters, making it favourable for implementation as a self-contained post-processing module (Requirement 3). Furthermore, if need be, the amount of detail preserved by the operator can be reduced to increase performance, which creates leeway for finding a possible balance between visual quality (Requirement 1) and performance (Requirement 2).

Section 4.1 explains Reinhard's photographic tone reproduction method in detail. Section 4.2 then presents a series of successive adaptions of Reinhard's operator to the GPU, with each adaption more closely approaching the goal of real time local tone mapping at high resolutions. Finally, Section 4.3 reviews the papers presented here, and discusses the outlook from the current state of the art, with respect to the goals outlined in Chapter 3.

4.1 Reinhard's method for photographic tone reproduction

The tone mapping method presented here, *Photographic Tone Reproduction for Digital Images* proposed by *Reinhard et al.* [RSS+02], is the foundation for the work in this thesis. Throughout the remainder of this thesis, this is referred to as *Reinhard's method* or *Reinhard's operator*.

Reinhard's method takes inspiration from formal techniques in photography, which are based on Ansel Adam's *Zone System*[Ada80, Ada81, Ada83]. With these photographic techniques in mind, a simple and computationally efficient global luminance compression operator has been developed, as well as a complex, more computationally demanding local tone mapping operator required for scenes with particularly high dynamic ranges. Section 4.1.1 describes the procedure for global luminance compression, and Section 4.1.2 presents the additional measures that must be taken for local tone mapping.

4.1.1 Global luminance compression

The main idea behind global tone mapping is to map all pixel luminances in an image to the range $[0, 1]$. In order to do this, a global image luminance characteristic must be determined, which can then be used as the basis of the mapping. As a simple example, such a characteristic could be the maximum luminance value of the image and the consquent mapping could be to divide the luminance of each pixel by this value. This would, however, lead to a linear tone mapping, which, as can be seen in Figure 2-8, produces undesirable visual results. Instead, Reinhard's method uses the notion of an *image key*.

The key of an image is defined as its the log-average luminance:

$$\overline{L}_w = exp(\frac{1}{N}\sum_{x,y} log(\delta + L_w(x,y))) \tag{4.1}$$

where $\overline{L}_w$ is the image key, $L_w(x,y)$ is the luminance for pixel (x,y), N is the total number of pixels in the image and δ is a small constant to ensure that the log function is still defined for input luminances of 0.

Scenes that are relatively bright, such as the sunset depicted in Figure 4-1 (a), are attributed with high key values (Figure 4-1 (b)), whereas darker images, such as indoor environments (Figure 4-1 (c)), have lower key values (Figure 4-1 (d)).

Once the scene key has been determined, it is used to compute a *scaled luminance* value for each pixel in the scene:

$$L(x,y) = \frac{\alpha}{\overline{L}_w} L_w(x,y) \tag{4.2}$$

where $L(x,y)$ is the scaled luminance of pixel (x,y), L_w is as defined in equation 4.1, $\overline{L}_w$ is the scene key and α is a parameter to which, through application of the above equation, pixels with luminances surrounding or equal to the scene key are mapped. This mapping of luminances near the scene key to the parameter α is analogous to setting camera exposure in photography. Therefore, α can be considered an exposure parameter. For "normal key" scenes, Reinhard chooses a default α value of 0.18, which is consistent with the ("middle-grey" [RSS+02]) value used in standard photography. For high key scenes, lower values of α are used, and vice-versa for low key scene. Typically, the value of α ranges from 0.045 up to 0.72, depending on the scene. Figure 4-2 demonstrates the effect of varying α for a sample HDR image.

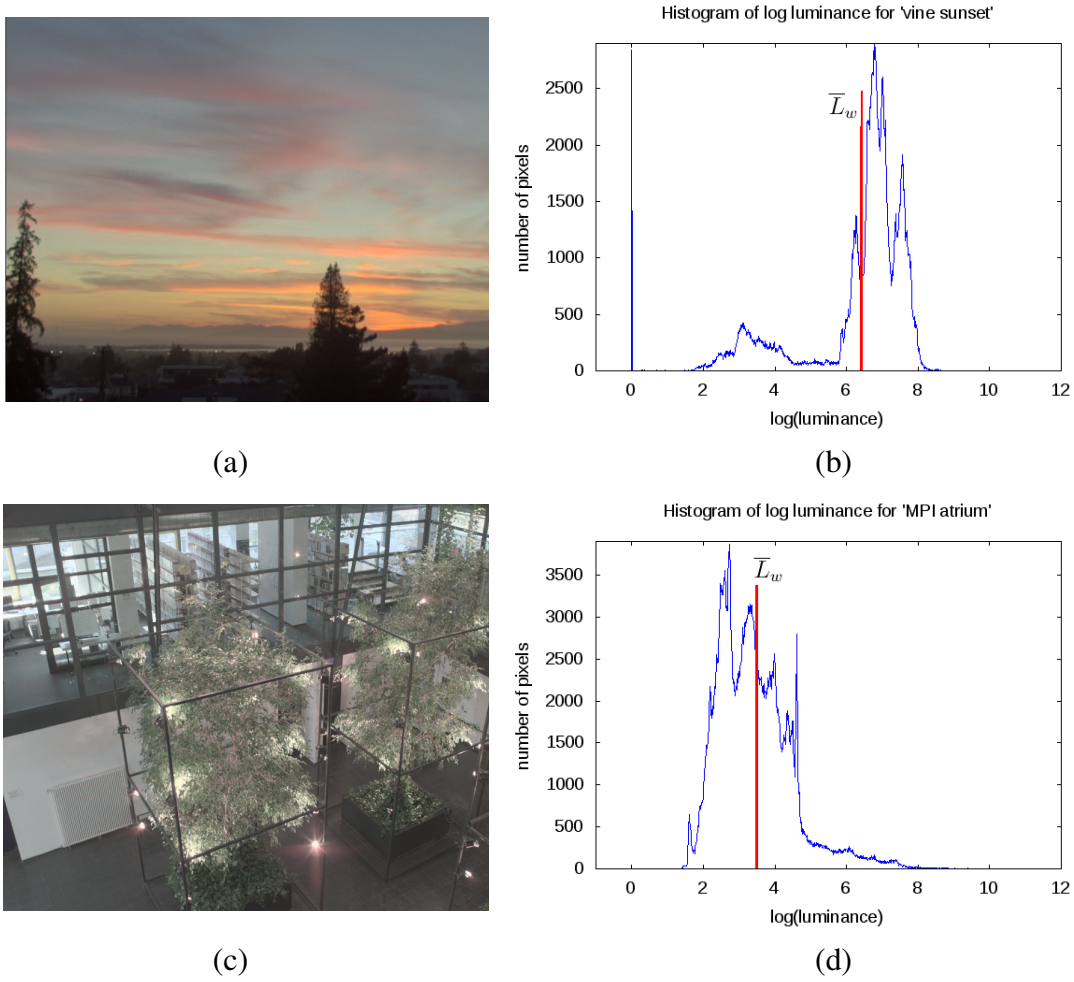


Figure 4-1: A high key HDR scene ("vine sunset") (a) and its luminance histogram (b), compared to a low key HDR scene ("MPI Atrium") and its corresponding luminance histogram. In both histograms, the scene key, $\overline{L}_w$, is marked as a vertical red line. See Appendix A for more information on the radiance maps used.

Finally, the scaled luminances computed in Equation 4.2 for each pixel are mapped into the interval [0, 1] using the following tone mapping operator:

$$L_d(x,y) = \frac{L(x,y)}{L(x,y)+1} \tag{4.3}$$

where $L(x,y)$ is the scaled luminance for pixel $p(x,y)$ computed in Equation 4.2 and $L_d(x,y)$ is the final, tone mapped LDR output luminance.

Equation 4.3 compresses high luminances by approximately 1/L, while low luminances are scaled by a factor close to 1. Figure 4-3 shows a plot of this function, which converges asymptotically to 1. Luminances surrounding the global average (which were mapped to the key value α by Equation 4.2) are compressed nearly linearly, whereas large luminances are compressed more forcefully. As a result, slight luminance variations (detail) in regions of near-average luminance are preserved, while similar detail in high luminance, such as cloud patterns in a bright sky or terrain lit by bright car headlights in a

key value 0.09 key value 0.18 key value 0.36 key value 0.72

Figure 4-2: Demonstration of how the selection of key value affects tone-mapping of the "Memorial Church" (for radiance map source, see Appendix A).

night scene, will be lost during compression.

4.1.2 Local dodging-and-burning

While the global luminance compression operator given in Equation 4.3 is sufficient for many HDR images, images with detail in regions of high luminance require additional measures to avoid this detail being lost during luminance compression. In his solution, Reinhard draws inspiration from a well established method in the field of tranditional photography, known as *dodging-and-burning*. Dodging-and-burning is a process in which, during development of a photographic print, light is withheld from regions with high relative brightness (dodging), while relatively dark regions are subjected to additional exposure (burning). This has the effect of bringing whole regions of extreme relative brightness or darkness closer to the scene average, while preserving their internal local detail.

For example, if a photographer were working with an indoor scene in which a window to brightly lit daytime scenery were visible, developing the print with uniform exposure would cause the scenery in the window to appear extremely bright, while the indoor details in the foreground would appear undesirably dark. Instead, by dodging the bright scenery in the window and burning the dark scenery in the foreground, details throughout the entire scene would become more easily viewable.

The key concept behind Reinhard's local tone mapping operator is an automatic implementation of the dodging-and-burning procedure. The first step in this procedure is to identify the appropriate regions of relative brightness or darkness to which automatic dodging-and-burning should be applied. Hence, for each image pixel $p(x, y)$, the largest surrounding neighbourhood of approximate isoluminance, known as that pixel's *local region* $L'(x, y)$, must be found. An example of a pixel and its corresponding local region is shown in Figure 4-4.

In order to identify the local region surrounding a pixel $p(x, y)$, a measure of average luminance in the area surrounding that pixel is required. Convolving the image by a Gaussian kernel (Section 2.2.2.2) centered at (x, y) with a radius s provides a suitable weighted average (where the weighting favours pixels closer to the center) of the lumi-

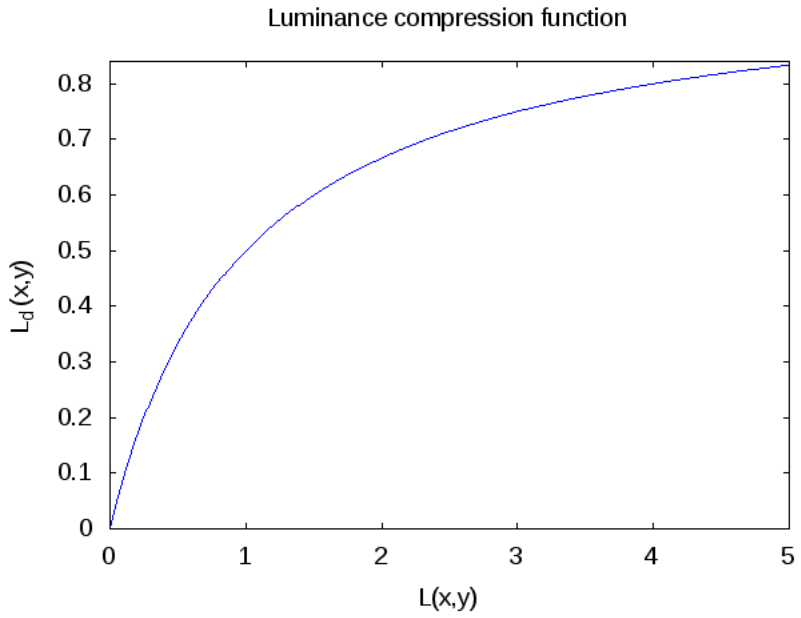


Figure 4-3: Plot of scaled luminance $L(x, y)$ against the resulting tone mapped LDR result $L_d(x, y)$ (Equation 4.3) for a given pixel $p(x, y)$. Low scaled luminance values (representing near-average scene luminances) are compressed approximately linearly, while high scaled luminance values (representing scene luminances many times that of the scene average) are compressed more forcefully.

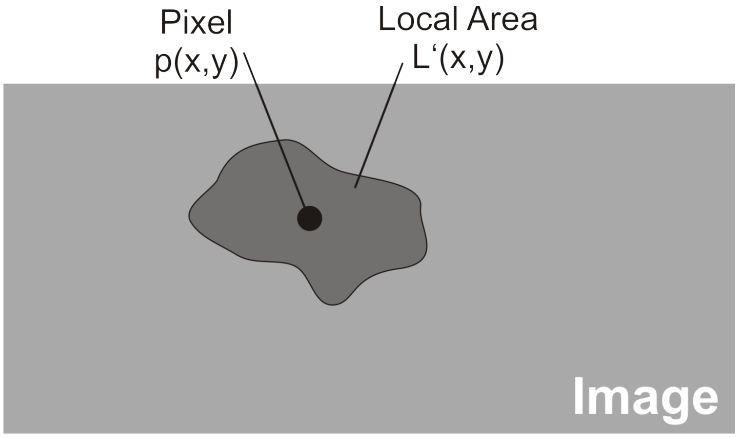


Figure 4-4: The first step in automatic dodging-and-burning is to find the local region $L'(x, y)$ for each pixel $p(x, y)$

nances within a circular region of radius s surrounding the target pixel. By evaluating the weighted average of a succession of circular regions of increasing radius, and comparing each result with its predecessor, the largest region of approximate isoluminance surrounding the target pixel can be found. Therefore, for each pixel $p(x, y)$ in the scaled luminance map produced by Equation 4.2, a series of response functions V are computed:

$$V(x, y, s_i) = L(x, y) \otimes R(x, y, s_i) \tag{4.4}$$

where $L(x, y)$ is the scaled luminance at pixel $p(x, y)$, $R(x, y, s_i)$ is a Gaussian kernel of scale s_i (see Section 2.2.2.2) and $V(x, y, s_i)$ is the response generated by the convolution. The scale of each kernel is 1.6 that of its predecesor, i.e s_{i+1} = $1.6s_i$. By default, a total of 8 scales ($i = 1 \rightarrow 8$) are computed for each pixel.

Computing a series of Gaussian convolutions with increasing kernel radius, such as those in Equation 4.4, for a pixel $p(x, y)$ is known as constructing a *Gaussian pyramid* for that pixel. Figure 4-5 illusrates this concept. Starting from the target pixel (scale 0), each layer represents a Gaussian convolution centered at the target pixel and of a larger than its predecessor. By comparing adjacent levels in the pyramid, any sudden changes in average luminance existing between them can easily be identified. Reinhard defines a center-surround function for identifying such changes:

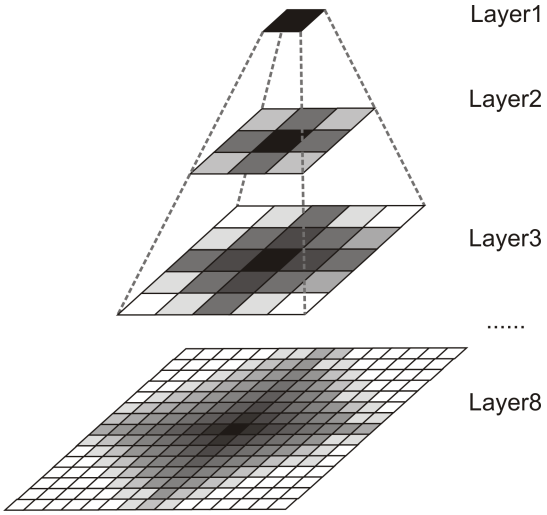


Figure 4-5: A Gaussian pyramid.

$$W(x, y, s_i) = \frac{V(x, y, s_i) - V(x, y, s_{i+1})}{2^{\phi}\alpha/s_i^2 + V(x, y, s_i)} \tag{4.5}$$

where the center $V(x, y, s_i)$ and surround $V(x, y, s_{i+1})$ are as defined in equation 4.4, α is the same as in equation 4.2 and ϕ is a free sharpening parameter.

By evaluating $W(x, y, s_i)$ for increasing scales until a certain threshold is exceeded, the largest region around a pixel with no large contrast changes can be found. Formally, a threshold ϵ is defined, and the largest scale s_{max} is found such that

$$|W(x, y, s_{max})| < \epsilon \tag{4.6}$$

holds. The Gaussian response computed for s_{max} is then used for the final, local tone-mapping:

$$L_d(x, y) = \frac{L(x, y)}{V(x, y, s_{max}) + 1} \tag{4.7}$$

where $L(x, y)$ is the scaled luminance value for image pixel $p(x, y)$, $V(x, y, s_{max})$ is as defined in Equation 4.4 and $L_d(x, y)$ is the final, tone mapped LDR output luminance. Naturally, in order tone map an entire image, L_d, and hence the entire procedure described above, must be computed for each pixel in the input image.

The additional detail that becomes visible when using automatic dodging-and-burning is demonstrated in Figure 2-9, Chapter 2.

4.1.3 Performance

Although it produces desirable visual results, Reinhard's local photographic tone reproduction method, in the form described here, remains too computationally intensive for use in real time applications. Table 4-6 shows the time required for tone mapping a single HDR image of various resolutions. Results were attained by measuring the execution times of an open source C++ implemenation of Reinhard's local operator, which was executed on the system detailed in Appendix A.

Image resolution	**Time (ms)**
720x480	1000
1024x1024	3000

Figure 4-6: Performance of a CPU implementation Reinhard's operator.

Clearly, the execution times in Figure 4-6 greatly exceed the maximum time of ca. 30 ms allowed per frame required for interactive frame rates (defined in Chapter 3 as 30 fps).

4.2 Local tone mapping on the GPU

The first generation of GPUs with fully programmable vertex and fragment shaders introduced a new level of massively parallel computational power to the consumer market. For example, in 2003 NVIDIA's GeForce FX card could compute at a sustained rate of 51 GFLOPS, which was about 8 times that of the fastest Pentium 4 CPU available at the time [GWH05]. Additionally, the introduction of vertex and fragment shading languages allowed scientists to use GPUs to solve problems that were not strictly graphics related. Because its major bottleneck is a per-pixel convolution by Gaussian kernels of multiple scales, and convolving an image by a Gaussian kernel is a highly parallelizable process, Reinhard's local photographic tone reproduction method is well suited for implementation on the GPU.

4.2.1 A GPU pipeline for local photographic tone reproduction

The first adaption of Reinhard's local tone mapping operator to the GPU was proposed by *Goodnight et al.*[GWW$^+$03] in 2003. By leveraging the massiviley parallel processing powers of the GPU, they were able to achieve a significant speedup over traditional, CPU-based implementations.

Goodnight's proposed GPU pipeline is detailed in Figure 4-7. The shader-based tone mapping system (right dashed rectangle) is designed as a post-processing module which can be used by any OpenGL application (left dashed rectangle). The tone mapping system is made up of a collection of shader programs (circular blocks) and intermediate data storages (rectangular blocks). Rendering targets are referred to as *buffers*, and are denoted by solid grey rectangles. After its processing pipeline has been executed, *Buffer 0* contains a scaled luminance map (Equation 4.2), which can then either be used directly for global tone mapping (by applying Equation 4.3 inside the *Operator global/local* oval) or for local dodging-and-burning (by passing it to the processing pipelines of *Buffer 1* and *Buffer 2*). The details of Goodnight's GPU implementation of automatic dodging-and-burning are described in the following subsection.

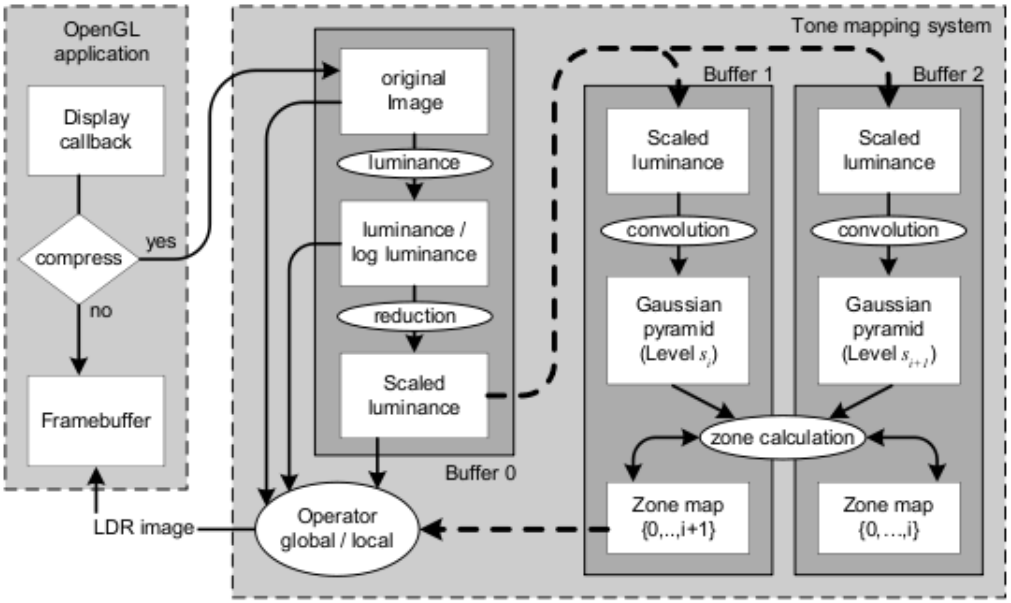


Figure 4-7: The pipeline for implementation of Reinhard's local tone mapping operator on the GPU. Image source [GWW+03].

4.2.1.1 Automatic dodging-and-burning

Although a convenient and intiuative approach to automatic dodging-and-burning would be to compute all levels of the Gaussian pyramid, store each level in a seperate texture and pass all textures to a shader program, which could, in a single pass, determine the suitable local region of each pixel, such an approach is in practice infeasable. For higher resolutions, storing several adjacent full resolution copies of the convolved input image would require copious amounts of video memory. Additionally, the shader program in this solution would necessarily contain conditionals for each level of the Gaussian pyramid; when executed on a SIMD architecture, conditionals are very expensive [GWW+03]. Therefore, a more efficient multi-pass method has been proposed.

The appropriate area of approximate isoluminance surrounding each pixel required for local tone mapping (see Section 4.1) can be determined by computing a so-called *scale*

map[1] for the input image. A scale map is a texture of equal dimensions as the input image, in which each texel stores the average luminance of the approximately isoluminant neighbourhood of its corresponding input pixel. Given an input image and its corresponding scale map, it is straightforward to write a shader program (the *Operator global/local* elipse in Figure 4-7) that uses Equation 4.7 to produce a locally tone mapped output image.

The advantage of using a scale map is that it can be accumulated using multiple offscreen rendering passes, where each pass i adds only the pixels belonging to scale i to the zone map (Equation 4.4). Because membership to a given scale is determined by examining two adjacent levels of the Gaussian pyramid (Equation 4.5), each pass requires only two pyramid levels as input and contains no expensive branching code. Figure 4-8 illustrates the first three passes of a zone map computation. The first pass takes the first and second levels of the Gaussian pyramid as input, and outputs a partial scale map with only the pixels identified as belonging to the first scale set. The second pass takes the second and third levels of the Gaussian pyramid, as well as the partial scale map produced in the first pass, and outputs a new partial scale map with only the pixels belonging the the first and second scales set. This process continues for as many passes as required. In order to implement Reinhard's local operator to its default resolution of 8 scales, 8 passes are required.

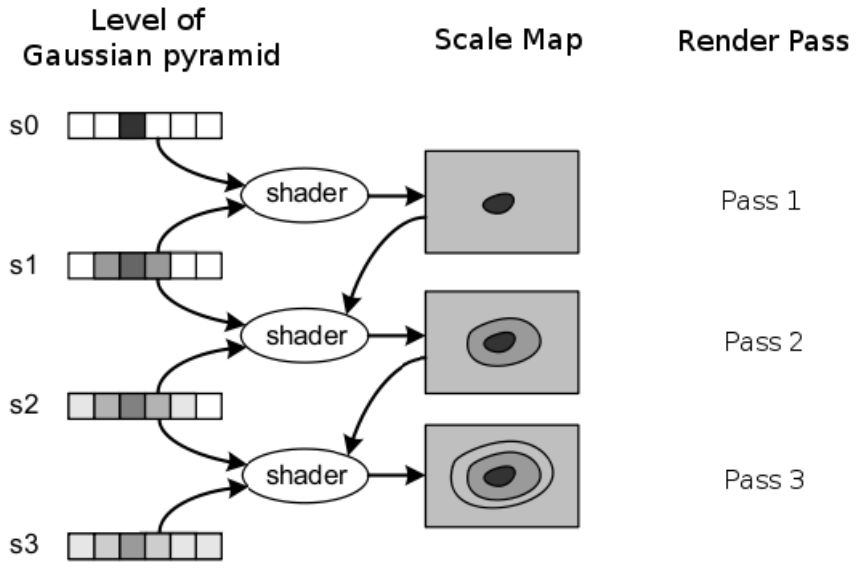


Figure 4-8: Zone map computation. Diagram adapted from [GWW+03].

Gaussian convolution

Although potential memory usage and code branching pitfalls were avoided by computing a scale map using the multi-pass method described in section 4.2.1.1, there still remains a major bottleneck in the computation: for each pass in the scale map computation, the scaled-luminance map must be convolved by a Gaussian kernel of an appropriate scale. By exploiting the seperability of Gaussian convolutions (Section 2.2.2.2), *Goodnight et al.*

[1] Although *Goodnight et al.*[GWW+03] use the term *zone map*, we refer to this as a *scale map* in order to avoid confusion with the term *zone* in photography, which is used frequently throughout Reinhard's paper [RSS+02].

were able to implement each convolution by an nxn kernel as a two pass $O(n)$ operation, rather than the $O(n^2)$ required for non-seperable convolutions of equivalent dimensions. Additionally, by packing the convolution kernels, as well as the input image luminances, into collections 4-vectors when passing them to the GPU, convolution could be efficiently implemented using highly optimized hardware dot-product instructions.

Performance

Despite their optimization efforts, at the time of publishing, *Goodnight et al.* could only tone map a 512x512 input image at interactive frame rates ($\geq$30fps) using two scales (see Figure 4-9), which is insufficient for preserving many details in images with very high dynamic ranges. Figure 4-10 compares an image tone mapped using two scales with one tone mapped using the default of eight. Significant loss of detail can clearly be seen in the bright, snowy regions. Although their results were published several hardware generations prior to the time of our research, and would certainly be more impressive if the pipeline were run on today's GPUs, the main pipeline bottleneck - repeated Gaussian convolution of high resolution textures - remains a significant challenge for modern hardware. Therefore, it is likely that a modern implementation of the proposed pipeline, without additional optimization, would not deliver interactive frame rates for very high resolutions.

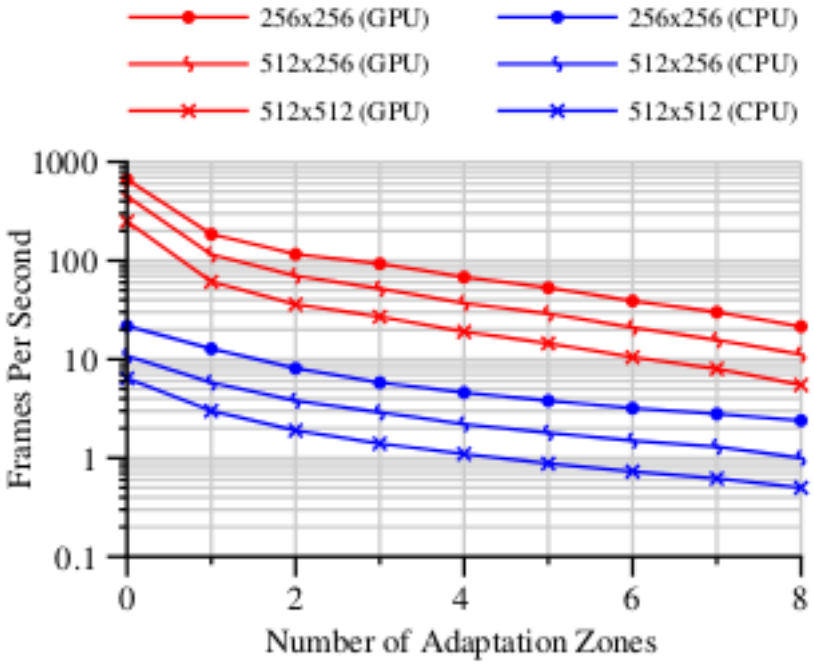


Figure 4-9: Performance of Goodnight et al.'s [GWW+03] GPU implementation at the time of publishing.

4.2.2 Convolution optimization by texture resampling

In 2005 *Krawczyk et al.*[KMS05] proposed further optimizations to the pipeline introduced by [GWW+03] (section 4.2.1). Downsampling the input texture by a carefully selected factor, convolving the input by a scaled approximation of the desired Gaussian kernel, then upsampling the convolved texture to its original dimensions led to signifi-

(a) (b)

Figure 4-10: An HDR scene tone mapped with 2 scales (a), which is the maximum that [GWW+03] could achieve in real time for a 512x512 texture, compared to the same scene tone mapped with the default of 8 scales (b). It can be seen that significant detail in the snow is lost when operating with only 2 scales. See Appendix A for more information on the radiance map used.

cant performance gains. Figure 4-11 shows the process for generating each level in the Gaussian pyramid.

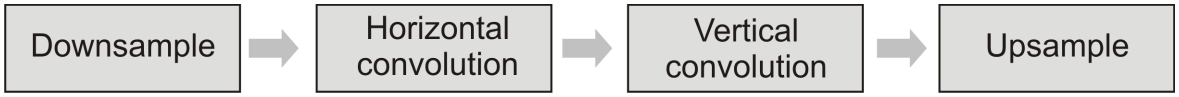


Figure 4-11: An optimization of Gaussian convolution proposed by Krawczyk et al. [KMS05], which allows full 8-scale local tone mapping at interactive frame rates. Each input texture is downsampled by a carefully selected factor, convoluted horizontally and vertically by a down-scaled approximation of the horizontal and vertical components of the Gaussian kernel, then upsampled back to its original dimensions.

Using this technique, HDR image streams could be tone mapped using 8 scales at interactive frame rates. The frame rates achieved on an NVIDIA GeForce 8800GTX GPU with 768 MB of memory are shown below[2].

Image resolution	Frame rate (fps)
256x256	189
512x512	157
1024x1024	74

Figure 4-12: Performance of local tone mapping with texture downsampling.

The significant performance gains won through approximate convolution of downsampled textures, however, came at the cost of visual quality. Both the downsampling and

[2]Results shown are taken from [SO08], because the performance measurements given in [KMS05] are distorted by additional perceptual effects simulation.

upsampling operations used in this procedure have the effect of blurring sharp edges (and thereby the local region boundries which the subsequent automatic dodging-and-burning attempts to identify), which results in noticable halo artificats surrounding high contrast edges in the final output [SO08]. Furthermore, implementation of the proposed method using shaders requires storage of several intermediate results using textures, increasing the memory footprint of the operator.

4.2.3 Approximating Reinhard's local operator using GPU-based Summed-Area Tables

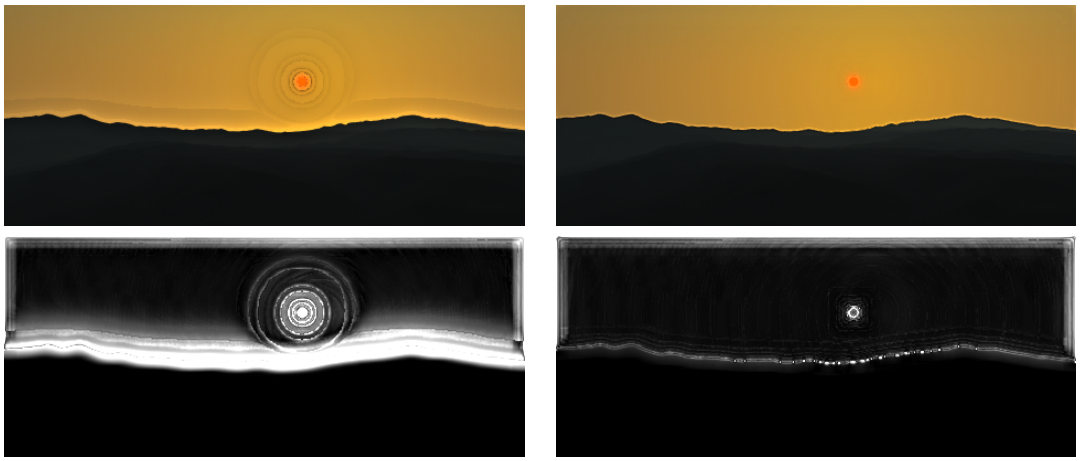
Noting that the primary bottleneck of Reinhard's local operator is repeated image convolution by a Gaussian kernel for each input image pixel, which is essentially repeated computation of weighted averages surrounding each pixel, *Slomp and Oliveira* [SO08] proposed an approximation of Reinhard's method in which Gaussian convolution is replaced by simple box filtering. Conceptually, the only modification to Reinhard's original algorithm is that Equation 4.4 has been updated to:

$$V(x, y, s_i) = L(x, y) \otimes Box(x, y, s_i) \tag{4.8}$$

where $Box(x, y, s_i)$ is a box filter kernel centered at (x, y) with radius s_i. Additionally, in order to account for the fact that box filter pixel weighting is uniform, while Gaussian kernels weight pixels closer to the center higher than those near the kernel edges, the threshold value ϵ in Equation 4.6 modified to $\epsilon = 0.025$, rather than the default of $\epsilon = 0.05$ used in Reinhard's original method.

By computing the S-CIELAB [ZW97] spatial perceptual error metric between their results and gold standard images produced by a software implementation of Reinhard's original local operator, *Slomp and Oliveira* showed that approximating Reinhard's automatic dodging-and-burning procedure using box filters and a different ϵ-threshold introduces very little visual error to the tone mapping. Figure 4-13 shows a comparison of the perceptual error introduced using *Slomp and Oliveira*'s method and compares it to that caused by *Krawczyk et al.*'s texture resampling optimization described in Section 4.2.2. It can clearly be seen that *Slomp and Oliveira*'s method produces favourable visual results. Furthermore, a recent study by *Linnemann et al.* [LWR$^+$09], in which a population of test subjects were asked to compare images of simple objects tone mapped using Slomp's method with the same objects in real life, showed that Slomp's method produces very good visual results.

The box filtering used in Slomp's method is performed very efficiently with the help of *Summed-Area Tables (SATs)*. As described in Section 2.2.2.3, given an image's SAT, the average pixel value in any given rectangular region of the image can be computed with 3 additions, 1 subtraction and 1 division, i.e in constant ($O(1)$) time. Therefore, the challenge of high speed box filtering on the GPU lies in efficiently constructing SATs for each input frame. *Slomp and Oliveira* use a method proposed by *Hensley et al.* [HSC$^+$05] for efficient parallel SAT construction on GPU hardware by means of *recursive doubling* [DR77]. Using Hensley's method, the SAT for a wxh image can be computed using a total of $log_k(w) + log_k(h)$ parallel steps, where k is the number of pairwise additions per step. The parameter k is adjustable: Raising k increases the work per parallel step



(c) (d)

Figure 4-13: A HDR image tone mapped using Krawczyk's texture resampling method (a), compared to the same image tone mapped using Slomp's box filter approximation (b). Krawczyk's method clearly produces visual artifcats, or halos, surrounding high contrast images. The perceptual error, measured using the S-CIELAB metric [ZW97] with the same scene tone mapped using Reinhard's original operator as the ground truth, is shown for Krawczyk's method (c) and Slomp's method (d) bottom row. It can be seen Slomp's method produces considerably less error than that of Krawczyk. Images taken from [SO08].

and decreases the number of steps, while reducing k has the converse effect. *Slomp and Oliveira* found an optimal balance when $k = 8$.

Image resolution	Frame rate (fps)
256x256	385
512x512	243
1024x1024	102

Figure 4-14: Performance of local tone mapping using Slomp and Oliveira's box filtering approximation of Reinhard's local operator.

Figure 4-14 shows the resulting frame rates for a number of images of varying resolutions. Using their high speed box filter approach, *Slomp and Oliveira* were able to attain substantially higher frame rates than Krawczyk's texture resampling method, which, until Slomp's publication, was previously the fastest GPU implementaton of Reinhard's local operator. In addition to better performance, Slomp's method also produces less visual error to that of Krawczyk. Of the local Reinhard operator implementations presented in this chapter, this method comes the closest to fullfilling Requirements 1 and 2 in Chapter 3, making it of great interest to the research in this thesis.

4.3 Summary and discussion

This chapter presented the relevant work in the field of real time local tone mapping. A suitable method, which consistently produces high quality visual results and is well suited for GPU-based acceleration, was selected from a multitude of existing local tone mapping methods. The chosen method, Reinhard's tone mapping operator [RSS+02], which is the foundation upon which the research in this thesis is based, was explained in detail in Section 4.1. Reinhard's operator has both global and local components (Sections 4.1.1 and 4.1.2, respectively.), where the local component preserves many details lost by its global counterpart, at the expense of significantly greater computational overhead. The performance bottleneck of Reinhard's local operator is caused by repeated, per-pixel convolution of input images by 8 scales of a Gaussian pyramid (Figure 4-5). As a result, when executed on a modern, multicore CPU, the local operator runs about 2 orders of magnitude too slowly for use in interactive applications. Therefore, all subsequent attempts to execute Reinhard's local operator at interactive frame rates has utilized the massively parallel computing power of GPUs.

The first adaption of Reinhard's local operator to the GPU was presented by *Goodnight et al.* [GWW+03] (Section 4.2.1). *Goodnight et al.* considerably reduced the overhead of applying 2D Gaussian kernels by using 1D separable kernels implemented on the GPU. They also optimized convolution computation by packing Gaussian kernels, as well as their target images, into 4-vectors and using special, optimized dot product instructions in the convolution shaders. This produced interactive results at low resolutions, as long as the height of the Gaussian pyramid was limited to 2 scales, which negatively impacted visual results.

Krawczyk et al. [KMS05] (Section 4.2.2) presented an optimization of *Goodnight et al.*'s approach by down-sampling input images by a carefully selected factor, convoluting with an appropriate Gaussian filter, then up-sampling back to original dimensions. While this resulted in a significant speedup, it was prone to "halo artifacts" [Kra07, p.15] as well as prohibitive memory requirements for high resolution images.

More recently, *Slomp and Olivera* [SO08] (Section 4.2.3) presented a method for implementing a slightly modified version Reinhard's local tone mapping operator using box filters, rather than Gaussian filters, which was efficiently implemented using the method proposed by *Hensley et al.* [HSC+05] for high speed construction of Summed-Area Tables (SATs) on the GPU. Their tone mapping performed superiorly to Krawczyck's in terms of computation speed and memory usage, while producing fewer halo artifacts.

Chapter 3 outlined the research goals of this thesis. Of particular importance are interactive frame rates at very high resolutions and high quality visual results. In light of these goals, *Slomp and Oliveira*'s method is the most interesting candidate at present. Their reported results, however, are only given for image resolutions up to 1024x1024 which is far from the VND operating resolutions of up to 1920x1200 on standard PC systems. It is likely that the performance of their operator will drop below acceptable frame rates for input images of the target VND resolutions, particularly in the HD Visualization Center.

Figure 4-15 shows where each of the methods presented in this Chapter lie in a Quality vs. Performance plot, as well as where the goal of this thesis lies. *Slomp and Oliveira*'s shows great potential, but there is room for further optimization to their method. Therefore,

approximating Reinhard's local operator by SAT-based box filtering is the starting point for the investigation undertaken throughout the remainder of this thesis.

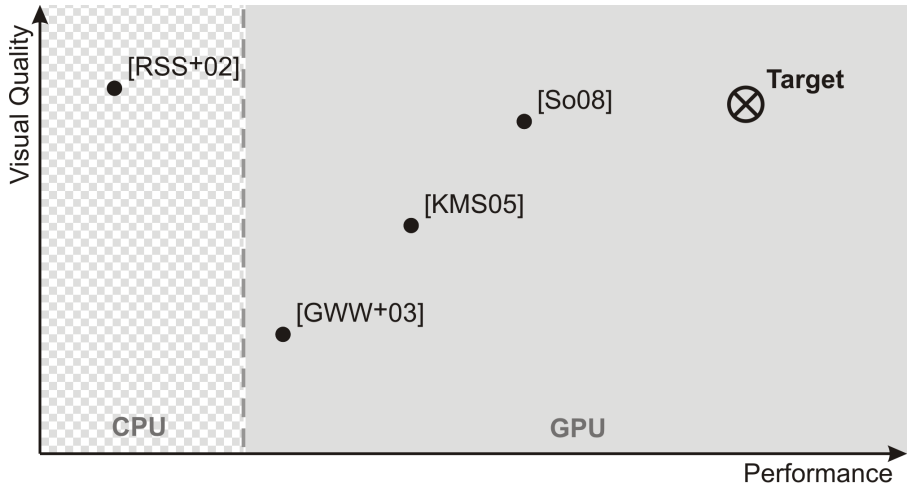


Figure 4-15: Performance vs. visual quality of the methods presented in this chapter. Reinhard's local tone mapping operator [RSS+02] is the base method, which all subsequent papers have attempted to optimize. Goodnight et al. [GWW+03], Krawczyk et al. [KMS05] and Slomp and Oliveira [SO08] all proposed adaptions of Reinhard's operator to the GPU. The black target indicates where the ideal operator would be placed in this context.

5 Approach

Chapter 4 presented the chosen tone mapping operator - Reinhard's local photographic tone reproduction method - as well as a number of recent implementations of this operator on the GPU. The most promising of these implementations is that of *Slomp and Oliveira* (Section 4.2.3), which achieves substantial performance gains by approximating the Gaussian convolution central to Reinhard's method with box filters.

Section 5.1 begins by reviewing *Slomp and Oliveira*'s tone mapping pipeline and analysing it for potential points of optimization. Section 5.2 discusses ideas for accelerating SAT generation - a central operation in *Slomp and Oliveira* method. It is shown that the SAT generation method used by *Slomp and Oliveira* is not optimal, and a more recent, efficient alternative is presented. It is then revealed that parallel *scan* computation, the base operation used in SAT generation, performs extremely well when implemented in CUDA. Encouraged by these results, Section 5.3 investigates the possibility of implementing a CUDA tone mapping module by conducting a prelimary study on CUDA's suitability for high speed OpenGL post-processing. Given the positive results reported in Section 5.3, Section 5.4 moves on to present a CUDA post processing module developed for tone mapping interactive HDR OpenGL applications. Finally, to evaluate the benefits of using CUDA, Section 5.5 presents the same module implemented in shaders.

5.1 *Slomp and Oliveira*'s method

5.1.1 Method description

The full process with which *Slomp and Oliveira*'s implementation of Reinhard's operator tone maps a HDR image is shown in Figure 5-1. In the example, a HDR scene (in this case the "memorial" scene), represented in a 16-bit-per-channel RGBA format, is given. The example illustrates how a single HDR image is tone mapped. Interactive applications would need to process each output frame in the same manner as shown here. Each of the 5 main tone mapping steps in the Figure are described below.

Step 1: Extract luminance. The luminance for each pixel in the input HDR scene is computed from its red, green and blue components. Computing luminance from an input RGB image produces an image containing only monochrome luminance values, called a *luminance map*. For the memorial scene in the example, the majority of pixel luminances are well above the maximum displayable value of 1.0. These values are clamped to white when rendered normally, producing the nearly entirely white image in the figure.

Step 2: Identify scene key. In this step, the luminance map generated in Step 1 is analysed and the *scene key*, defined as the scene's average log luminance (Section 4.1.1), is determined.

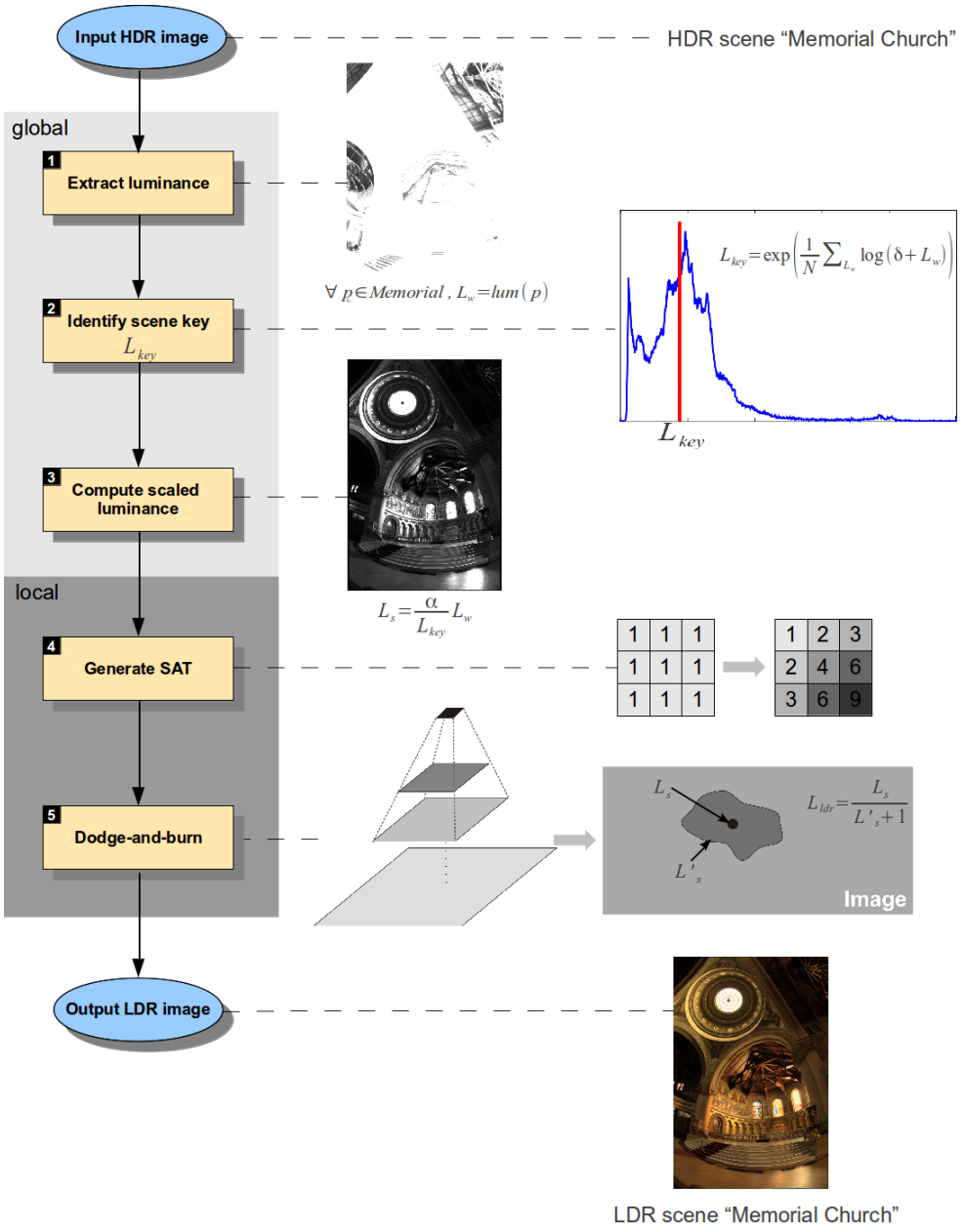


Figure 5-1: The five major steps of Slomp and Oliveira's approximation of Reinhard's local operator. Steps 1-3 are global operations (applied uniformly for all pixels) and Step 4 prepares support data structure Step 5, which is a local operation (computed individually for each pixel). The example HDR image used is the "Memorial Church" scene detailed in Appendix A.

Step 3: Compute scaled luminance. Given the luminance map (Step 1) and its corresponding scene key (Step 2), Step 3 computes the scaled luminance (Equation 4.2) for each pixel in the luminance map, producing a *scaled luminance map*. As can be seen in the figure, a large portion of the pixels in the luminance map have now been mapped into the visible range (with pixels equal to the scene key mapped to the camera exposure parameter α from Section 4.1.1). At this point, if a global tone mapping were desired, it could easily be achieved by applying Reinhard's global tone mapping function (Equation 4.3) to each pixel in the scaled luminance map. Since local tone mapping is of primary

interest, however, the realm of of local pixel operations is entered (light grey region in the figure) by proceeding with Step 4.

Step 4: Generate SAT. A Summed-Area Table (SAT) is generated from the scaled luminance map produced in Step 3. This operation is performed in $\lceil log_k(w) \rceil + \lceil log_k(h) \rceil$ passes using general-purpose shader programs.

Step 5: Dodge-and-burn. The SAT from Step 4 is used to efficiently compute a series of regional averages of increasing scale for each pixel in the scaled luminance map. This is visualized by the pyramid on the bottom right of Figure 5-1, where the top level represents a single pixel in the scaled luminance map, and each level below it represents the average scaled luminance of a rectangular region surrounding that pixel. Because the SAT enables computation of regional averages of any size in constant time, no complete box filter pyramid is explictly computed prior to tone mapping. Instead, using the SAT, these averages are dynamically computed to the necessary scale for each pixel during the dodging-and-burning procedure.

By computing high speed box filters, the *Local Region* (Section 4.1.2) surrounding each pixel in the scaled luminance map is identified. Once identified, the Local Region is used to compress the luminance of its corresponding pixel into the displayable range $[0, 1]$ (Equation 4.7). This step deviates minorly from Reinhard's method in that, because box filters are used in the search for Local Region, the Local Region identified for each pixel is rectangular. In the case of Reinhard's original operator, Gaussian filters, which have a circular distribution, are used for this search. Therefore, the resulting Local Regions in Reinhard's method are circular. It has been shown (Section 4.2.3) that this has little impact on visual quality.

5.1.2 Optimization opportunities

This section investigates the potential for optimization of *Slomp and Oliveira*'s method, which is, as far as could be determined, the fastest implementation of Reinhard's method to date. Potential peformance bottlenecks are found by analysing the steps described in Section 5.1.1.

Steps 1-3 in the light grey region in Figure 5-1 are computed globally for the entire input. If, rather than proceeding to step 4, the output of step 3 were compressed according to Equation 4.3, the resulting LDR image would be tone mapped using Rheinhard's global operator (Section 4.1.1). It is well established that Reinhard's global operator easily performs at interactive frame rates when implemented on the GPU (consider *Goodnight et al.*'s [GWW+03] results (Section 4.2.1) for 0 scales). Therefore, attention is focused on the local steps (4-5) in Figure 5-1.

Step 4 involves constructing a SAT from the scaled luminance map produced in Step 3. This operation must be performed once per input frame - at least 30 times per second for interactive applications.

Slomp and Oliveira use a method proposed by *Hensley et al.* [HSC+05] for SAT generation, whereby SATs are computed on the GPU by means of *recursive doubling* [DR77]. Using *Hensley et al.*'s method, when given a wxh image, the image's SAT can be constructed in $\lceil log_k(w) \rceil + \lceil log_k(h) \rceil$ parallel steps, where k is an adjustable parameter rep-

resenting the number of pairwise additions per step (Section 4.2.3). At this point a clear opportunity for optimization presents itself: more recent work has shown that parallel prefix sum computation - an operation central to SAT construction - can be computed more (work-)efficiently than by the recursive doubling method used in *Slomp and Oliveira*'s method. Section 5.2 provides a detailed discussion of *Hensley et al.*'s method and presents a more efficient alternative.

Finally, in **Step 5**, the Local Region for each pixel is identified by iteratively comparing box filter averages of ascending scale centered at each pixel. Evaluation of each layer of the pyramid can be accomplished in O(1) computation time using the SAT generated in Step 4. Thus, unlike in Reinhard's original method, it is unlikely that arithmetic computation will be a limiting performance factor in this step. Consequently, the scale maps used by *Goodnight et al.* [GWW+03] and *Krawczyk et al.* [KMS05] (Section 4.2), will likely be unnecessary here, as their focus is to minimize the overhead involved in expensive Gaussian convolution computation.

Since Step 5 access the input SAT four times for each box filter computed, and up to eight such computations are possible for each pixel of each input frame, it is important to optimize memory access. Chapter 6 examines optimization of SAT memory organization and access.

5.2 Optimizing SAT generation

The central approach of this thesis is identifying and implementing methods for high speed SAT generation, thereby accelerating *Slomp and Oliveira*'s local tone mapping implementation.

Section 5.2.1 begins by introducing the concept of the *all-prefix-sum*, also known as *scan*, which is the basic building block of SATs. The method by which they are computed in parallel in Slomp's tone mapping is then analysed in detail in Section 5.2.3 and subsequently shown to be non work-efficient. An alternative, work-efficient parallel prefix sum construction algorithm is then introduced in Section 5.2.4. Finally, the potential for additional performance gains by implementing the work-efficient parallel prefix sum algorithm in CUDA are discussed in Section 5.2.5.

5.2.1 The building block of SATs: all-prefix-sums

All-prefix-sums are a simple and widely used construct in computer science. The definition of all-prefix-sums, as stated in [Ble90], is given below.

Definition *Given an ordered set of* n *elements*

$$[a_0, a_1, ..., a_{n-1}]$$

and a binary associative operator $\otimes$*, the* all-prefix-sums *operation returns the ordered set*

$$[a_0, (a_0 \otimes a_1), ..., (a_0 \otimes a_1 \otimes ... \otimes a_{n-1})].$$

Although this operation appears inherently sequential, efficient parallel implementations have been proposed (see Sections 5.2.3 and 5.2.4). When applied to an array of data, the all-prefix-sum operation is also referred to as a *scan* [HSO07]. We will use the term *scan* for the remainder of this document.

The Summed-Area Table (defined in Section 2.2.2.3) of a given matrix is simply a two dimensional scan of its rows and columns, where the operator $\otimes$ is addition.

5.2.2 Sequential scan

Computing a scan on an input array $data$ of size n sequentially is straightforward. The procedure for sequential scan computation is given below.

Algorithm 1 Sequential scan on array $data$ of size n.

```
for i = 1 → n − 1 do
    data[i] ← data[i] + data[i − 1]
end for
```

Algorithm 1 performs exactly $n - 1$ addition operations to complete its task. In the ensuing sections, a *work-efficient* parallel scan algorithm is searched for. The term *work-efficient* indicates that the amount of work performed by a parallel algorithm shows the same asymptotic behaviour as that of its sequential counterpart. In this case, a work-efficient parallel scan algorithm will not execute more than $O(n)$ addition operations. The asymptotic bound of an algorithm's work is denoted as its *work complexity*.

5.2.3 The non-work-efficient parallel scan used by *Slomp and Oliveira*

The SAT generation technique used in *Slomp and Oliveira*'s implementation is that of *recursive doubling*, which was first implemented on the GPU by *Hensley et al.* [HSC+05]. Using Hensley's method, a scan can be computed on a data array of length n using n processors in $\lceil log_k(n) \rceil$ parallel steps, where k is the customizable number of elements added per step.

Method overview
We begin by considering the case in which each processor computes one pair-wise addition per pass, i.e $k = 2$. Given an input array $data$ containing n elements, the parallel algorithm listed in Algorithm 2 is executed.

An important property of the array $data$ in the above algorithm is that any out of bounds array accesses return 0, i.e $\forall(j < 0 \wedge j \geq n), data[j] == 0$. Because 0 is the addition identity, all accesses outside of array bounds will have no affect on the final outcome.

The steps executed by Algorithm 2 during scan computation are shown in Figure 5-2 for an example input of 8 elements, labeled A through H. The first step ($i = 0$) updates each

Algorithm 2 Parallel scan using recursive doubling on array $data$ of size n.

for $d = 0 \rightarrow \lceil log_2(n) \rceil - 1$ **do**
  for all $j \in [0, n-1]$ **in parallel do**
    $data[j] \leftarrow data[j] + data[j - 2^d]$
  end for
end for

element by adding to it the contents of its left neighbour ($2^i = 2^0 = 1$). Because the immediate predecessor of the first array element, A, lies outside of the input array, and all out of bounds elements are evaluated as 0, A will remain unchanged. At this point, A contains its final value and the remaining elements B through H contain partial sums. The next iteration ($i = 1$) sets each element to the sum of itself and the element two addresses to its left ($2^i = 2^1 = 2$). After this step, A and B both contain their final values, while elements C through H contain partial sums. This process continues until the scan is complete. At any given iteration i, the first 2^i elements will contain their final values, and the remainder partial sums requiring further processing.

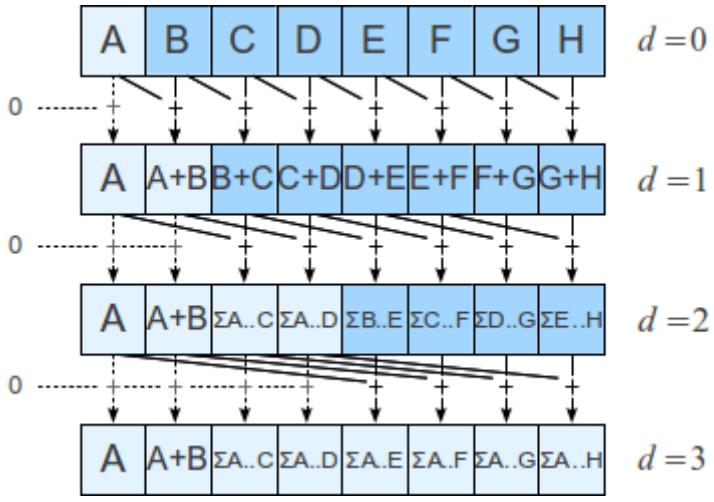


Figure 5-2: Recursive doubling with $k = 2$ applied to an 8 element example array. Light blue elements contain final values, while dark blue elements contain partial results requiring further processing. Each out-of-bounds array access returns 0, which has no effect on the element with which it is added (dashed lines). For any iteration d, the left most 2^d elements contain their final values. Therefore, a total of $\lceil log_2 n \rceil$ passes are required to reduce an array of size n, where each pass performs n additions (including the addition of final results with 0).

To reduce the number of passes required for scan computation, the amount of work performed per pass can be increased. Algorithm 3 shows the algorithm for scan computation in the general case, where k elements are added per step. The motivation behind increasing the per-pass complexity lies in the fact that, in practice, there is a certain system-dependent overhead involved in setting up each parallel computation step. When the computation performed per processor per step is sufficiently small, the overhead of invoking each parallel pass outweighs that of the actual computation done per pass. On the other hand, increasing k in the Algorithm 3 has the effect of increasing the amount

of sequential work per processing unit, thereby reducing the parallelism of the algorithm. *Slomp and Oliveira* found an optimal balance when $k = 4$ [SO08].

Algorithm 3 Recursive doubling with k work complexity per step.

for $d = 0 \rightarrow \lceil log_k(n) \rceil - 1$ **do**
  for all $j \in [0, n-1]$ **in parallel do**
    $data[j] \leftarrow \sum_{r=0}^{r \leq k} data[j - r * k^d]$
  end for
end for

Constructing SATs on the GPU

Hensley's algorithm is intended for implementation on GPU hardware, which introduces additional practical considerations. In particular, when implemented using shader programs, the array $data$ in Algorithm 2 would be represented with a texture. At present, shader programs are unable to read and write to the same texture in a single pass, so a common technique called *ping-ponging* must be incorporated. Ping-ponging is a procedure whereby a shader is called repeatedly, and its input and output textures are swapped between each pass, which causes the output of a each pass i to become the input of its following pass $i + 1$. Ping-ponging is illustrated in Figure 5-3.

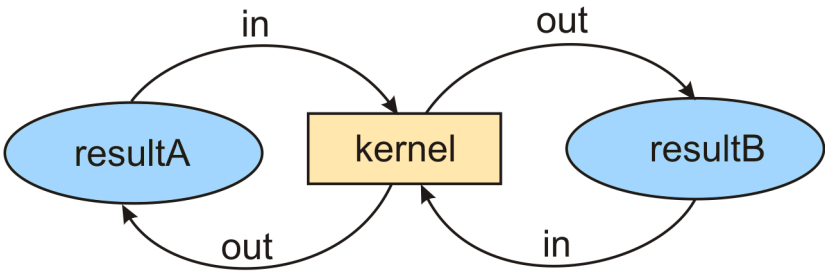


Figure 5-3: Texture ping-ponging with shaders. Because shaders are unable to read and write from the same texture during a single pass, a result is accumulated using two textures, resultA and resultB, which swap roles as input and output for the shader program for each pass.

In order to construct a SAT from a 2D wxh texture on the GPU, Hensley incorporates texture ping-ponging to compute scans by means of recursive doubling for each row of the input texture in parallel. Because each recursive doubling step is applied to all rows simultaneously, a texture with each row containing a scan of its corresponding input row is generated in $\lceil log_2(w) \rceil$ passes (for the sake of simplicity, we consider the case when $k = 2$). This texture is then passed as input to a simliar procedure, which computes a parallel scan for each column of its input simultaneously in $\lceil log_2(h) \rceil$ passes. The resulting texture contains the SAT of the original input texture. The algorithm described here, Hensley's shader SAT generation algorithm, is given in Algorithm 4.

Performance and work-complexity

Consider the case when $k = 2$. Each of the $\lceil log_2(n) \rceil$ parallel steps in Algorithm 2 performs a total of n additions. Therefore, the work complexity of this operation is

Algorithm 4 Generating a SAT for input texture $inTex$ of dimensions wxh.

t_a = $inTex$

// Horizontal pass
for $i = 0 \rightarrow \lceil log_2(w) \rceil - 1$ **do**
  for all $(x, y) \in t_a$ **in parallel do**
    $t_b(x, y) \leftarrow t_a(x, y) + t_a(x - 2^i, y)$
  end for
  $swap(t_a, t_b)$
end for

// Vertical pass
for $i = 0 \rightarrow \lceil log_2(h) \rceil - 1$ **do**
  for all $(x, y) \in t_a$ **in parallel do**
    $t_b(x, y) \leftarrow t_a(x, y) + t_a(x, y - 2^i)$
  end for
  $swap(t_a, t_b)$
end for

// Texture t_a contains result

$\Theta(nlog_2(n))$. Similarly, in the general case, each of the $\lceil log_k(n) \rceil$ parallel steps computes $n(k-1)$ additions:

$$\lceil log_k(n) \rceil \, (n(k-1)) = O(n * k * log_k(n))$$

Figure 5-4 shows a plot of the number of addition operations performed per pass for $k = 2$, $k = 4$ and $k = 8$ in Algorithm 3, and compares this to the additions computed by the sequential scan algorithm (Algorithm 1). By examining its work complexity, it can be concluded that *Hensley's algorithm is not work-efficient*, because $O(n * k * log_k(n))$ exceeds the linear work complexity of its sequential counterpart.

It should be further considered that, for any given pass i, the first k^i elements of the input array already contain their final values. During this and all subsequent passes, these values are copied by their assigned processing unit, tying up system resources for unnecessary work. This problem is compounded by the fact that the GPU typically provides fewer processors than fragments being processed, so fragments are batched into groups that are processed sequentially. Particularly for large resolutions, performance will be impaired by the sequentialization caused by processing fragments already containing their final result.

5.2.4 A work-efficient parallel scan

The previous section discussed the parallel scan algorithm used in *Slomp and Oliveira*'s tone mapping, and showed that these scans are not work-efficient and suffer from an unnecessary level of sequentialization for high resolution input datasets. This section

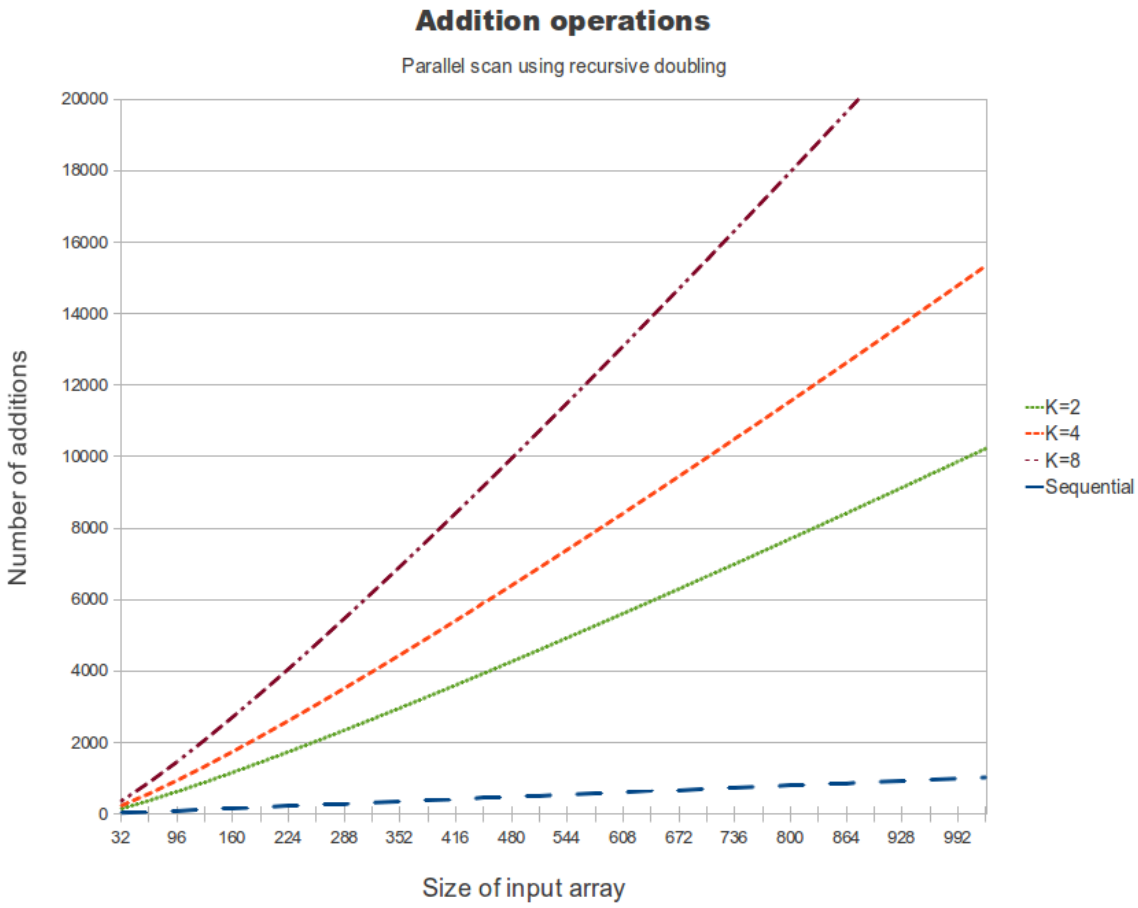


Figure 5-4: Total number of addition operations performed by Hensley's parallel scan algorithm (Algorithm 3) for $k = 2$, $k = 4$ and $k = 8$, compared to the amount of addition performed by a sequential scan. A significant discreptancy between the parallel and sequential scans is apparent. This expected, as the parallel algorithm has a work complexity of $O(n * k * log_k(n))$, while the sequential scan has $O(n)$.

introduces a more recent, work-efficient parallel scan algorithm proposed by *Sengupta et al.* [SLO06]. Sengupta's method consists of two phases. The first, *reduce*, phase constructs a tree from an input array, where the union of all leaf nodes is equal to the input array, the root node contains the sum of all leaf nodes, and the remaining internal nodes contain partial sums. The second phase, called the *down-sweep* phase, traverses the tree generated during *reduce* from top-down, using its contents to generate a scan of the input array. This section describes each phase in detail, then provides an analysis showing that this method is work-efficient.

Reduce phase

The *reduce* phase takes an input a_0 of length n, denoted by $|a_0| = n$, and constructs $log_2(n)$ partial sums, where each partial sum a_d is half the size of its predecessor a_{d-1} and each element i in a_d contains the pairwise accumulation of neighbouring elements $2i$ and $2i + 1$ in a_{d-1}. Algorithm 5 shows the pseudo-code for this operation.

Figure 5 shows how the reduce operation is applied to an input array of 8 elements. Each

Algorithm 5 Reduction phase in work efficient parallel scan.

for $d = 1 \rightarrow log_2(n)$ **do**
  for $i = 1 \rightarrow (n/2^d - 1)$ **in parallel do**
    $a_d[i] \leftarrow a_{d-1}[2i] + a_{d-1}[2i+1]$
  end for
end for

partial sum a_d contains the pairwise sums of the elements in a_{d-1}. In general, each element in a_d is the sum of 2^i elements of the input a_0 (bottom level in the figure). The root node at level 3 ($d = log_2(8) = 3$) contains the sum of all elements in the input array.

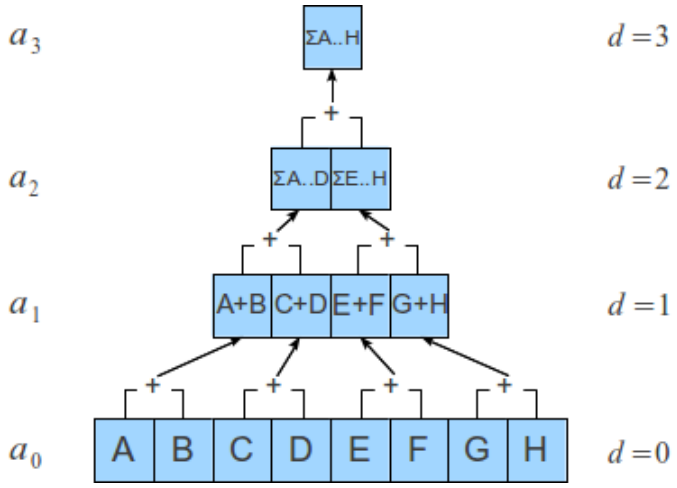


Figure 5-5: The reduce phase applied to an 8 element input array. In this example, the sum of all input elements is created in $log_2(8) = 3$ passes, performing a total of $n - 1 = 7$ additions.

Down-sweep phase

The down-sweep phase begins at the root of the tree in Figure 5-5 and traverses to the leaves (level $d = 0$), accumulating the partial sums at each level to produce a complete scan in a_0. Algorithm 6 shows the pseudocode for this operation.

When implementing Algorithm 6 using shaders, each partial sum must be stored in a texture. Furthermore, because shaders are unable to read from and write to the same texture in a single pass, an additional set of $log_2(n) - 1$ textures must be allocated for storing the intermediate results generated during the down-sweep. This process is shown in Figure 5-6 for an input array of 8 elements. The tree constructed during the reduce phase (Figure 5-5) is shown on the left of the dashed line. A set of additional textures corresponding to each level of the reduce tree, with the exception of the tree root, can be seen to the right of the dashed line (marked in dark blue). Each pass d of the down-sweep operates on the array a_d of partial sums generated during the reduce phase, as well as the result of the previous (level $d + 1$ in the Figure) down-sweep pass. The output of the final down-sweep pass is a complete scan of the original input array a_0.

As described in Section 5.2.3, a rectangular SAT can be generated from a 2D input texture by computing scans for its rows and columns.

Analysis

Algorithm 6 Down-sweep phase in work efficient parallel scan.

for $d = (log_2(n) - 1) \rightarrow 0$ **do**
  for $i = 0 \rightarrow (n/2^d - 1)$ **in parallel do**
    if $i > 0$ **then**
      if i is odd **then**
        $a_d[i] = a_{d+1}[\lfloor i/2 \rfloor]$
      else
        $a_d[i] = a_d[i] + a_{d+1}[\lfloor (i/2) \rfloor - 1]$
      end if
    else
      $a_d[i] = a_d[i]$
    end if
  end for
end for

Unlike Hensley's scan algorithm described in Section 5.2.3, Sengupta's scan does no wasteful computation in each pass. Furthermore, because the partial result textures contain only values for which computation is *necessary*, the problem encountered by Hensley's algorithm, where high resolution inputs reduce parallism by sending large volumes of unnecessary fragments to the shader, is not encountered here.

During the reduce phase, each of the $log_2(n)$ passes halves the size of its input. As a result, each pass d operates on an array a_d of size $|a_d| = n/2^d$, where n is the size of the input array. When reducing an input array of size m by half, a total of $m/2$ additions are required. Hence, a given pass d involves $n/2^{d+1}$ addition operations. The total number of additions W_R performed by the reduce sweep on an input array of size n is:

$$\begin{aligned} W_R &= \frac{n}{2^1} + \frac{n}{2^2} + ... + \frac{n}{2^{log_2(n)}} \\ &= n \left(\sum_{d=1}^{log_2(n)} \frac{1}{2^d} \right) \\ &= n \left(\sum_{d=0}^{log_2(n)} \frac{1}{2^d} - 1 \right) \\ &= n \left(\frac{1 - (1/2)^{log_2(n)+1}}{1 - (1/2)} - 1 \right) \\ &= n - 1 \end{aligned}$$

Simlarly, the down-sweep phase requires $log_2(n)$ steps. By analysing Algorithm 6, it can be seen that, for a given pass d, an addition operation is performed whenever i greater than 0 and even. It follows that, for an input array with even length m, a single pass computes $m/2 - 1$ additions. Each a_d in Algorithm 6 has length $|a_d| = n/2^d$, hence each pass d computes a total of $|a_d|/2^d - 1 = n/2^{d+1} - 1$ addition operations. The total number of additions W_D performed by the down-sweep can now be deduced:

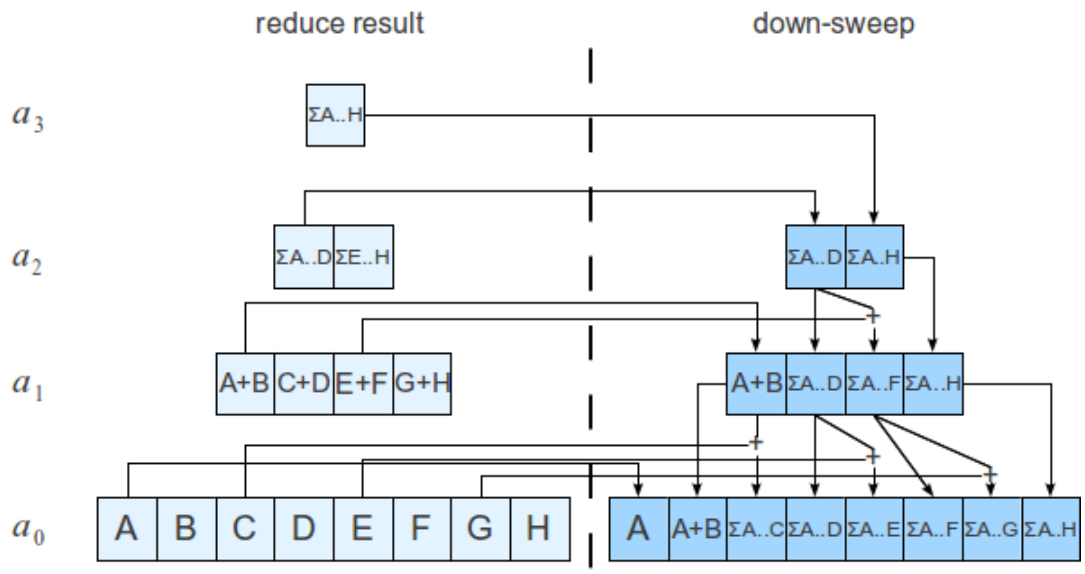


Figure 5-6: The down-sweep phase of Sengupta's work-efficient parallel scan algorithm applied to the results of the reduce phase shown in Figure 5-5 (light blue arrays to the left of the center line). The down-sweep commences at the top of the reduce tree and moves towards the leaves, generating a new set of partial results at each step (dark blue arrays to the right of the dashed line). When implemented with shaders, these new partial results must to stored in additionally allocated textures.

$$
\begin{aligned}
W_D &= \frac{n}{2^1} - 1 + \frac{n}{2^2} - 1 + ... + \frac{n}{2^{log_2(n)}} - 1 \\
&= n \left(\sum_{k=1}^{log_2(n)} \frac{1}{2^k} \right) - log_2(n) \\
&= n \left(\sum_{k=0}^{log_2(n)} \frac{1}{2^k} - 1 \right) - log_2(n) \\
&= n - log_2(n) - 1
\end{aligned}
$$

The total number of additions W_T performed by the complete scan algorithm is:

$$
\begin{aligned}
W_T &= W_R + W_D \\
&= 2n - log_2(n) - 2 \\
&= O(n)
\end{aligned}
$$

It follows from the above computations that *Sengupta's method is work-efficient.*

Figure 5-7 shows the total number of addition operations required to compute a scan for input arrays of sizes up to 1024 elements. For comparison, the work complexity of

Hensley's method (Section 5.2.3) and the sequential scan given in Algorithm 1 are also shown. It can be seen that Sengupta's method has the same asymptotic behaviour as the sequential scan.

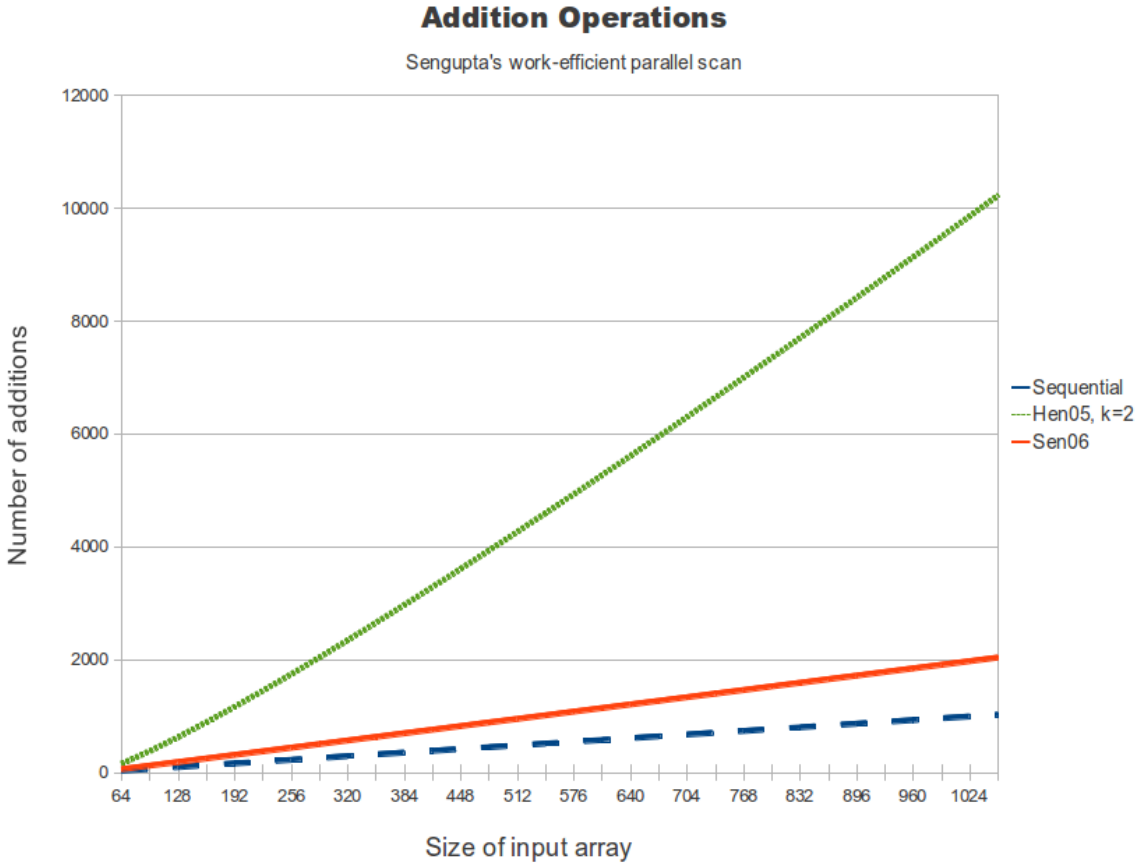


Figure 5-7: Total addition operations required to scan input arrays of sizes up to 1024 elements. Sengupta's work-efficient parallel scan algorithm shows the same linear asymptotic behaviour as the sequential scan, while Hensley's parallel scan algorithm (Section 5.2.3, shown here with $k = 2$) is bound by $O(nlog_2n)$. The work difference between Hensley's and Sengupta's methods increases monotonically with the size of the input array.

Although Sengputa's method is work-efficient, implementions with shader programs do not make optimal use of modern GPU hardware. Because shaders are unable to read and write to the same texture in a single pass, each partial result generated during the reduce and down-sweep scan phases must be stored in seperate textures, causing a significant memory footprint for large input textures. When computing a scan on an array of n elements, where each element is b bytes long, this overhead amounts to a total of

$$\left(2n \sum_{k=0}^{log_2(n)} \frac{1}{2^k} - 1\right) b = (4n - 3)b$$

bytes.

The intended area of application of the parallel scan operation discussed here is SAT construction for high resolution HDR images. Each SAT constructed during tone mapping

has the same dimensions as the input HDR scene. Because each SAT element is an accumulation of all its predecessors in both x and y directions, and the SAT is constructed from potentially large HDR floating point values, entries in the bottom-left of the SAT can become very large. *Hensley et al.*[HSC+05] showed that, given a wxh image, where each pixel is represented with P_i bits of precision, the number of bits of precision per pixel required for a SAT to be accurately generated from the image is:

$$P_s = log_2(w) + log_2(h) + P_i \tag{5.1}$$

To meet the high resolution requirements set out in Chapter 3, Equation 5.1 indicates that at least 38-bits of precision per pixel are required to construct a SAT from a 16-bit per pixel 2048x2048 input image. Fortunately, since the SAT in this method is constructed from a scaled luminance map, which, due to the the scaling operation (Equation 4.2), has a reduced range compared to the original HDR input, 32 bits of floating point precision per SAT entry will suffice.

When using a shader implementation of Sengupta's method, given a 2048x2048 input HDR image, the memory footprint of the SAT construction phase of the tone mapping procedure (Step 4 in Figure 5-1) will be $4 * (4 * 2048 * 2048 - 3)$ bytes, which amounts to 64 MB. Because SAT generation is only one part of the tone mapping procedure, and, according to the goals layed out in Chapter 3, the tone mapper will be implemented as GPU module suitable for integration into interactive applications, which may themselves use large amounts of video memory, it is important to keep the memory footprint of SAT generation as low as possible.

5.2.5 Leveraging GPU hardware with CUDA

General Purpose GPU (GPGPU) programming languages such as CUDA [GGN+08] (Section 2.3.3) are well suited for general computations, such as scan generation, on the GPU. One major advantage that GPGPU languages have over traditional shader languages is that they provide access to GPU hardware capabilites unavailable to shaders. Of particular interest in the context of scan generation is the ability to read from and write to the same global memory location during one single program (*kernel*) execution (Figure 5-8); access to high-speed, per-processor memory called *shared memory* (Figure 5-9); and inter-processor cooperation enabled by thread synchronization and atomic operations (Figure 5-10).

Harris et al. [HSO07] introduced a modification of Sengputa's method, which performs the reduce and down-sweep computations *in-place*, i.e without allocating additional memory for storage of partial results. Furthermore, Harris achieves significant speedup by dividing the input data into chunks, assigning the chunks to different GPU processors, using the processors to efficiently compute partial results with the help of high speed, on-chip *shared memory*, and then updating the input array with the partial results. This process is repeated until the input array contains a complete scan. Using this method, Harris reported scan speeds of up to 7 times those achieved by a shader implementation of Sengupta's method.

Harris' reported scan performance, as well as a memory footprint 25% that of a shader

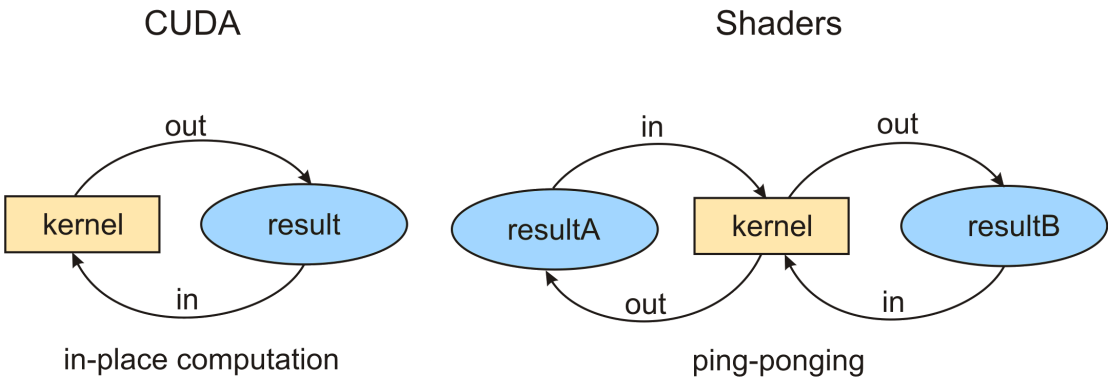


Figure 5-8: When iteratively accumulating a result, where the output of iteration i is the input of iteration $i+1$, shaders must ping-pong between two seperate textures due to their inability to read from and write to the same texture in a single execution. CUDA programs do not have this restriction.

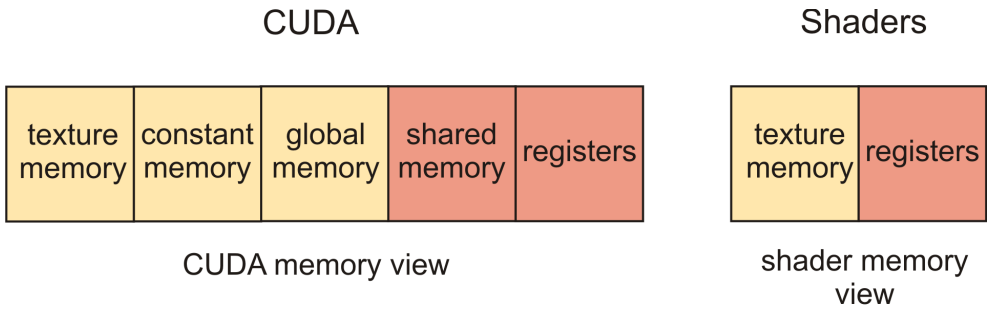


Figure 5-9: CUDA programs have a substantially more advanced view of GPU memory than shaders. In particular, CUDA programs have access to high speed, per-processor, on-chip memory called shared memory. Shared memory plays a crucial role in reaching performances unattainable by shaders.

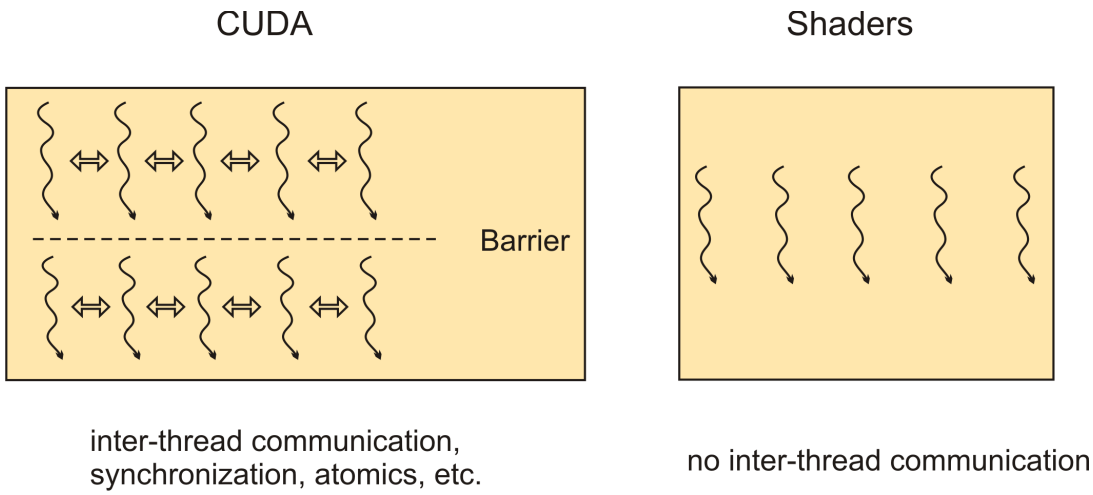


Figure 5-10: CUDA provides a more sophisticated parallel programming model than shaders. While multiple streams of execution run completely independently with shaders, CUDA provides primitives for inter-thread communicaton, thread synchronization and atomic operations.

implementaton of Sengupta's method, make scan computation with CUDA an attractive starting point for an investigation into high speed local tone mapping using box filters.

5.3 Using CUDA with OpenGL for post processing

The previous section concluded with the observation that parallel scans can be computed significantly faster using GPGPU languages than with shaders, due to less restricted access to GPU hardware. This leads to the possibility that implementing *Slomp and Oliveira*'s method (Section 4.2.3) in CUDA, using modern, work-efficient scan computation methods for SAT generation, may precipitate performance gains, particularly for high resolution scenes. In order to comply with the goals outlined in Chapter 3, a CUDA-based tone mapper must be implemented as a configurable, self-contained module that can be integrated into any existing OpenGL application. Therefore, a CUDA tone mapping module would need to be developed, which uses the GPU to post-process output from its host OpenGL application. Because CUDA itself has no functionality for rendering results on the screen, the post-processed result would need to be passed back to the host application for final rendering.

At the time of writing, it is unclear how efficiently CUDA can share the GPU with an OpenGL application for post-processing applications. A 2008 study by *Lönroth and Unger* [LU08] showed that any potential performance gains achievable by using efficient CUDA SAT-based box filter implementations to create a "depth of field" effect as a post-process, were lost due to the overhead of switching context between OpenGL and CUDA twice for each frame. Although OpenGL-CUDA context switching overhead was attributed to the fact that early versions of CUDA required a copy of frame data across the system bus to the CPU for each context switch, an issue that has since been resolved, there remain signs that prohibitive overhead exists when post processing with CUDA. For example, the sample CUDA post processing program *postProcessGL* supplied with the CUDA 3.2 SDK, which uses CUDA to apply a Gaussian blur to the output of a simple OpenGL program, runs at significantly lower frame rates than equivalent programs using pure OpenGL with GLSL shaders.

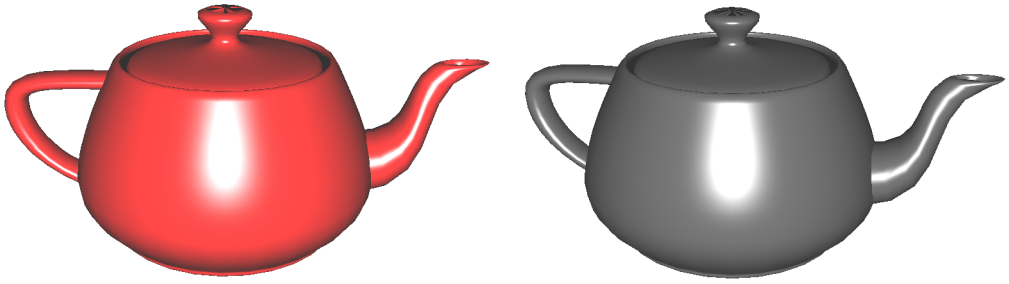
In order to determine whether the current CUDA release (version 3.2 at the time of writing) is capable of efficient OpenGL post-processing, a small study comparing a simple CUDA post-processing program with a an equivalent shader program has been undertaken. Section 5.3.1 describes the experiments performed, Section 5.3.2 provides the results, and Section 5.3.3 discusses the results in the context of the overall goal, real time tone mapping with CUDA.

5.3.1 Procedure

A simple OpenGL program was created, which renders a *Utah teapot* directly to floating point texture. Standard GLSL shader programs were used to illuminate the teapot using per-fragment Phong shading. Figure 5-11 (a) shows the contents of the output texture. Experiments were designed such that this texture is processed by post-processing programs, implemented both in CUDA and GLSL, which extract the monochrome per-pixel luminance from the original image, shown in Figure 5-11 (b). The results are then

passed back to OpenGL for final display. Figure 5-12 illustrates the setup used for the experiments. By comparing the number of frames that can be rendered per second in this manner, an evaluation of CUDA's suitability for post-processing OpenGL applications can be attained.

The platform used for testing is described in Section A.1.



(a) (b)

Figure 5-11: Contents of offscreen texture rendered by OpenGL (a) and the result of post processing with CUDA or GLSL (b).

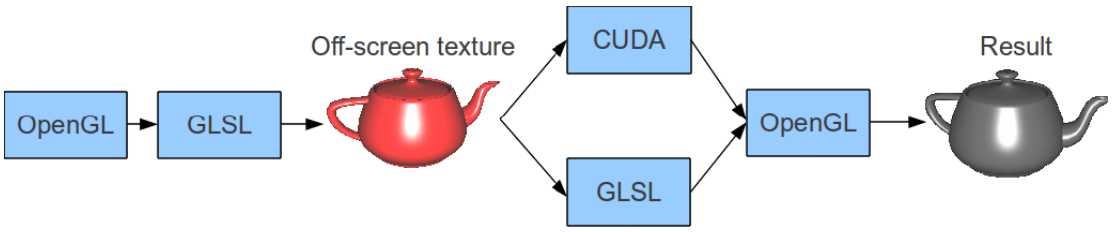


Figure 5-12: Experiment setup. A Utah teapot is produced with OpenGL and illuminated and rendered to texture using GLSL shaders. The resulting texture is then processed by either CUDA or GLSL programs, and the results are passed back to OpenGL for output on the display. Performance is evaluated as average time required to render each frame.

5.3.2 Results and discussion

Each experiment was executed for a total of 32 seconds, and average frames per second (fps) for the final 30 seconds of execution were logged. The first 2 seconds of each experiment were not logged so that various program initialisation operations would have no effect on the average frame rate measurements. Figure 5-13 shows a comparison of the framerates logged during 30 seconds of experiment execution with an output resolution of 1024x1024. The average frame rates, as well as average time per frame, throughout 30 seconds of execution can be found in Figure 5-14.

It can be seen that GLSL shaders perform marginally better than CUDA for post-processing. This can be attributed to the fact that the current release of CUDA is incapable of writing directly to texture. Instead, a CUDA post-processing program must write its result to a *Pixel Buffer Object (PBO)* - an OpenGL handle to pixel data residing on the GPU - and copy, or *unpack*, this data to the target texture. Although this copy takes place entirely on the GPU, avoiding the costly transfer of data across the system bus that made

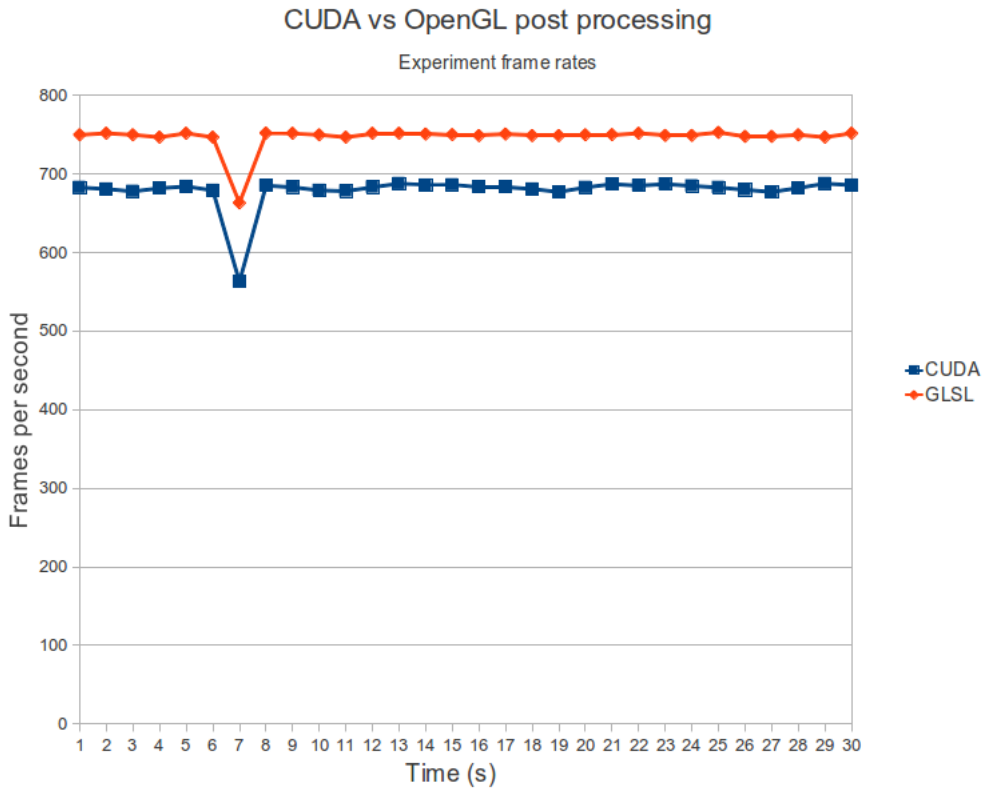


Figure 5-13: Frame rates measured over 30 seconds of post-processing a 1024x1024 OpenGL application. The blue dataset shows the frame rates when post-processing application output using CUDA and the red dataset shows the framerates when performing an equivalent operation using shaders.

	Average FPS	**Average time per frame (ms)**
CUDA	679	1.47
GLSL	747	1.34

Figure 5-14: Average frame rates and frame render times for the simple post-processing experiment.

post-processing with earlier versions of CUDA infeasible, this operation still introduces a small overhead. As can be derived from Figure 5-14, however, this overhead, 0.13 ms per frame for a 1024x1024 image on the test system, is near-negligble. As GPU memory speed and clock frequencies continue to rise, the time to copy memory between two locations residing in GPU memory will continue to fall.

5.3.3 Conclusion

The results of the small study presented here show that modern CUDA releases have become sufficiently mature to be used for post-processing OpenGL applications without incurring significant performance penalties.

5.4 A CUDA tone mapping module

Despite the promising parallel scan computation performance outlined in the Chapter 5, as far as could be determined, no investigation into real-time tone mapping using CUDA (or other GPGPU languages) has been undertaken. Therefore, a CUDA post-processing module has been developed, which applies local tone mapping in real time to a stream of HDR images produced by any OpenGL-based application.

Figure 5-15 shows how the CUDA module is used for tone mapping interactive HDR applications. Rather than rendering its output to the standard screen framebuffer, the target application (*OpenGL Application* in the top of the figure) uses special OpenGL functionality called *Render To Texture (RTT)* to render its output to a 16-bit per pixel floating point HDR texture. Attempting to render this texture directly using traditional OpenGL rendering would result in a white screen; the HDR texels are represented by 16-bit floating point values in the range $[0.0, 65535.0]$, and traditional OpenGL rendering requires texels in the range of $[0.0, 1.0]$, clamping all texels outside of this range to 0.0 (black) or 1.0 (white). Instead, the HDR texture is passed to the CUDA tone mapping module, which performs Steps 1-5 outlined in Section 5.1.1 to produce a LDR texture with all texels compressed into the displayable range of $[0.0, 1.0]$. To optimize the speed and memory consumption of the crucial SAT-generation step (Step 4 in Section 5.1.1), the CUDA tone mapping module uses Harris' efficient in-place scan computation algorithm. Once tone mapping is complete, the CUDA module writes the result into a standard, 8-bit-per-channel LDR texture, which is read back by the OpenGL and displayed on the monitor.

The conceptual, black box description used thus far may give the false impression that the CUDA module is a single, highly complex program that takes an input HDR texture and executes on the GPU until a tone mapped, LDR texture is produced. In reality, the module is a system of parallel CUDA programs, known as *kernels*, together with CPU-based sequential code for invoking and manipulating them. Figure 5-16 shows the internal module design. Each yellow rectangle represents a CUDA kernel, which is executed on the GPU. The elipses represent datasets, each containing at least as many elements as texels in the input texture, which are produced by or passed to the kernel programs. All such data resides on the GPU throughout module execution, i.e none of the data represented by a blue elipse is transfered across the bus connecting the GPU and its host computer system.

Prior to invoking the CUDA tone mapping module, the host OpenGL application renders its HDR output to a rectangular 16-bit-per-channel floating point texture of dimensions wxh. Once control is passed to the CUDA module, the first kernel in Figure 5-16 is invoked and passed the input HDR texture. This *Compute luminance* kernel derives the world luminance value, L_W, for each input pixel using Equation 2.1 defined in Chapter 2, producing a *luminance map* where each entry $p(x, y)$ is set to $log(L_W(x, y))$.

The next step is to determine the *scene key* of the input HDR scene (Section 4.1.1), defined as the log-average of all scene luminances. To find the scene key, which is equivalent to the the average value of the luminance map, the luminance map is passed to the *Reduction* kernel. Given an input of size n, the Reduction kernel produces an output with $n/2$ elements, where each element is an accumulation of two elements in the input dataset. Therefore, the Reduction kernel is invoked $log_2(w * h)$ times, repeatedly reducing its own

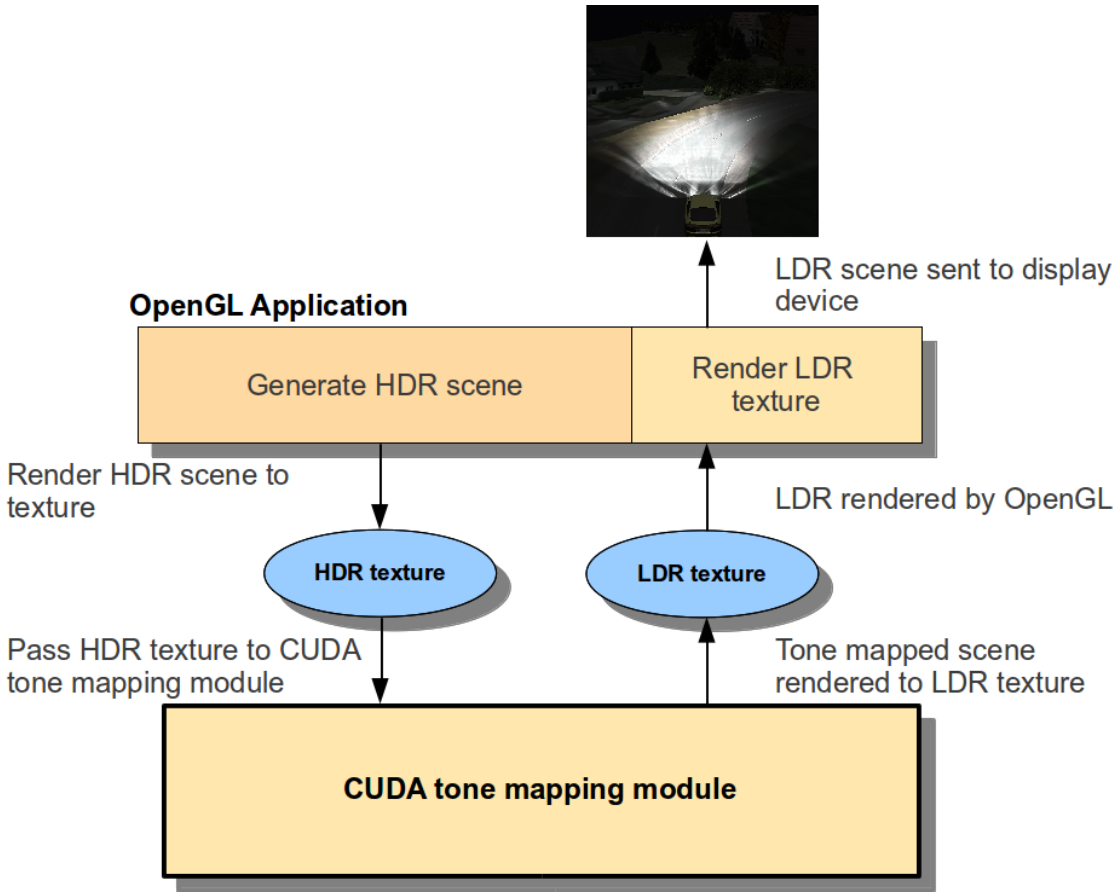


Figure 5-15: Integration of the CUDA tone mapping module into an OpenGL application.

output until a single value representing the sum of all elements in the luminance map is produced. By dividing this value by the total input dataset size, $w * h$, the scene key is attained.

The scene key is then used by the *Compute scaled luminance* kernel to compute the *scaled luminance* (Section 4.1.1) for each element in the luminance map, producing a *scaled luminance map*. Constant-time box filtering operations are then enabled by constructing a SAT from the scaled luminance map, an operation performed by the *Generate SAT* submodule.

Finally, given the luminance map, scaled luminance map, scaled luminance SAT and original input HDR texture, the *Dodge-and-burn* kernel uses high speed, SAT-based box filters to identify the Local Region for each pixel in the luminance map, and locally tone maps each pixel in the original input HDR texture accordingly. The output is written to a displayable LDR texture, which is then rendered by the OpenGL application.

The individual kernels shown in Figure 5-16 are described in detail in Sections 5.4.1 to 5.4.5.

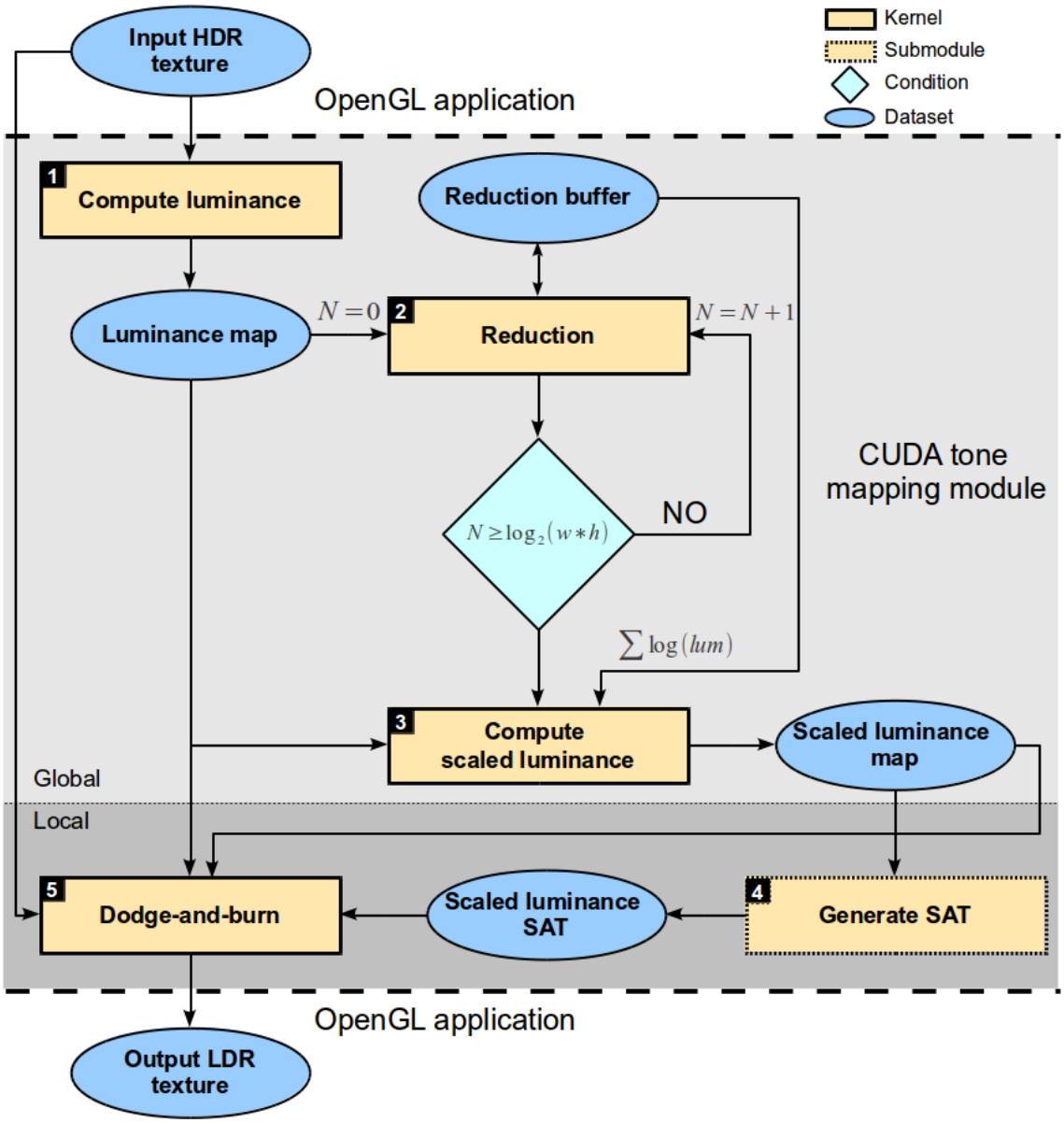


Figure 5-16: The design of the CUDA tone mapping module. Blue ovals represent datasets, which may be textures or arrays in gobal GPU memory, and solid yellow rectangles represent the CUDA kernels that process these textures. The dashed yellow rectangle, denoted "Generate SAT", represents a sub-module, which is itself a module of sequential code and CUDA kernels. The condition in the green diamond in the center of the diagram ensures that the reduction kernel is executed $\log_2(w*h)$ times, where w and h are the dimensions of the input HDR texture. Kernels in the light, "global" zone apply the same operation to all input elements, and can be used for both global and local tone mapping operators. Kernels in the darker zone, on the other hand, are required specifically for local tone mapping. The dashed line represents the border between host OpenGL application and the tone mapping module.

5.4.1 Luminance extraction

Since all subsequent steps in the tone mapping procedure operate on luminance values, the first kernel in the CUDA module computes the luminance of each input RGBA pixel in parallel and stores the results in a *luminance map*, denoted by L_M, in gobal memory. The luminance value L_W for each pixel in $p(x, y)$ in the input array is computed using Equation 2.1 defined in Chapter 2. Once L_W has been computed for each pixel, its logarithm is written to the luminance map:

$$L_M(x, y) = log(L_W(x, y) + \delta) \tag{5.2}$$

where δ is a small offset to avoid undefined results when $L_W(x, y) = 0$. Although δ is often set to a very small value close to 0 in many implementations, such as those by *Linnemann et al.* [LWR+09] and *Reinhard et al.* [RSS+02], in this work a value of $\delta = 1$ has been chosen. Because $log(1) = 0$, setting $\delta = 1$ causes 0-valued input pixels to produce 0-valued output values, causing no interference to later global metric (such as global average) computations on the luminance map.

Storing the logarithm of L_W, rather than L_W itself, in the luminance map greatly simplifies the subsequent task of identifying the scene key. Given luminance map containing luminance logarithms, the scene key can be computed simply by computing the average value of the luminance map. The original L_W value can easily be obtained from the luminance map using the exponential function: $L_W = e^{L_M}$.

5.4.2 Reduction

Having computed the luminance map, the next step in the tone mapping procedure is to identify the *scene key*. As described in Section 4.1.1, the scene key $\overline{L}_w$ is defined as the log-average luminance of the input image. Because the luminance map computed by the luminance extraction kernel already contains the logarithm of scene luminance values L_W, the definition of $\overline{L}_w$ (Equation 4.1, Section 4.1.1) can be simplified to:

$$\overline{L}_w = exp(\frac{1}{N} \sum_{x,y} L_M(x, y)) \tag{5.3}$$

where N is the number of elements in the luminance map (in this case wxh) and L_M is as defined in Equation 5.2.

To compute the sum of all elements of a dataset, a parallel *reduction* operation is required (described in Section 5.2.4). For this purpose, a CUDA reduction kernel has been used, which, given an input dataset with n elements in global memory, fills the first $n/2$ elements of the dataset with pairwise sums of its original values. After invoking the reduction kernel $log_2(w * h)$ times on a given dataset, the first element contains the sum of all elements in the dataset prior to the first kernel invocation.

In order to avoid overwriting the luminance map, the Reduction kernel, when invoked for the first time, writes its output into a buffer residing in global GPU memory (*Reduction buffer* in Figure 5-16). Each subsequent invocation computes in-place in the reduction

buffer, until the first buffer element contains the sum of all elements in the luminance map. This value is then copied to the host system and used to compute the scene key $\overline{L}_w$ using Equation 5.3, which is passed as a parameter to the next kernel.

5.4.3 Computing scaled luminance

Given the scene key $\overline{L}_w$ and luminance map L_M, the scaled luminance computation kernel computes the scaled luminance for each pixel in the luminance map, using the following computation (adapted from Equation 4.2 in Section 4.1.1).

$$L_S(x,y) = \frac{\alpha}{\overline{L}_w} exp(L_M(x,y)) \tag{5.4}$$

where α is an exposure parameter (described in Section 4.1.1) and L_S is the resulting *scaled luminance map*. Once computed, the scaled luminance map is stored in a wxh array in global GPU memory.

5.4.4 SAT generation

To compute box filters of arbitrary size in constant time, a Summed Area Table must be constructed. The SAT generation submodule takes the scaled luminance map and computes its SAT.

The fundamental procedure used by the SAT generation submodule is a work-efficient scan algorithm proposed by *Harris et al.* [HSO07]. Harris' scan algorithm is essentially a modification of Sengupta's algorithm (described in Section 5.2.4), which computes the *reduce* and *down-sweep* phases interatively inside a single array, rather than requiring additional arrays for storing partial results.

The Summed-Area Table for a 2D image is generated by computing a scan for each row of the input image, then computing a scan for each column of the result. By invoking a CUDA thread grid of equal dimensions to the wxh scaled luminance map and assigning each entry a thread, scans for each row in the scaled luminance map can be produced in parallel in a total of $log_2(w)$ steps. Similarly, column scans on the result can be computed in parallel in $log_2(h)$ steps.

Although computing row-scans on the input followed by column-scans on the result produces the desired SAT correctly and work-efficiently, additional considerations must be made during implementation with CUDA. Even if an algorithm is perfectly designed, its CUDA implementation may perform far below expectations if the programmer does not take care to use hardware resources correctly. In the case of SAT construction, the second step of computing simultaneous column scans causes *non-coalesced* (Section 2.3.3) accesses to global memory, which have a severe impact on kernel throughput. To avoid this, Harris proposed to transpose the result of the first row-scan operation, then compute a second row-scan on the transposed image. By adjusting all subsequent SAT indexing to account for transposed coordinates, a second tranpose to bring the completed SAT rows and columns back to their correct positions is not required. Figure 5-17 illustrates this procedure for an example 3x3 array with elements set to 1.

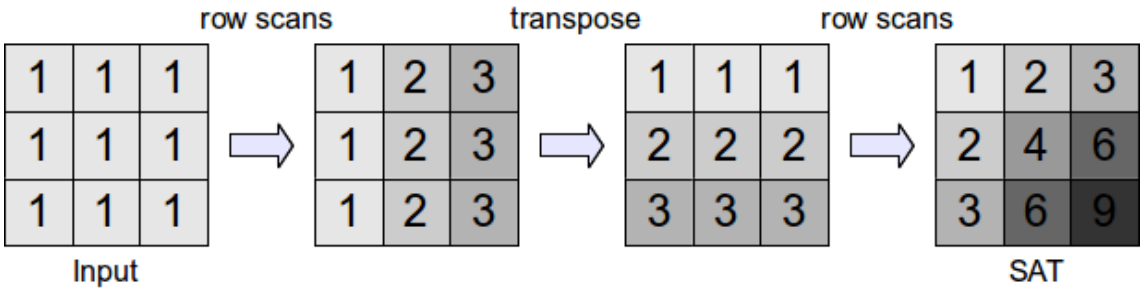


Figure 5-17: An example of a SAT generated from a simple 3x3 input array. Scans are computed for each row of the input simultaneously. Then, in order to optimize memory access patterns, the resulting array is transposed before the process of computing row scans is repeated. The result is a transposed SAT of the input array.

5.4.5 Dodging-and-burning

The final, dodging-and-burning kernel, takes the original input HDR image passed as input to the CUDA module, the luminance map L_M, the scaled luminance map L_S and the Summed-Area Table L_{SAT}, and uses them to compute the appropriate local luminance compression for each pixel in the input image.

Algorithm 7 lists the algorithm used for local tone mapping, applied in parallel to each pixel $p(x, y)$ contained in the input image. This algorithm illustrates the general procedure for tone mapping; the details may vary from implementation to implementation (there were a number of different implementations of this kernel, as described in Chapter 6).

Algorithm 7 Algorithm for local tone mapping.

Input: TEX_{HDR}, L_M, L_S, L_{SAT}, ϵ
Output: TEX_{LDR}

1: $s_{curr} = 1$
2: $activity = 0$
3: $i = 0$
4: **while** ($activity < \epsilon$ AND $i < 7$) **do**
5: $\quad s_{next} \approx 1.6 s_{curr}$
6: $\quad V_{curr} = BOX(L_{SAT}, s_{curr}, (x, y))$
7: $\quad V_{next} = BOX(L_{SAT}, s_{next}, (x, y))$
8: $\quad activity = getActivity(V_{curr}, V_{next})$
9: $\quad s_{curr} = s_{next}$
10: $\quad i = i + 1$
11: **end while**
12:
13: $L_{ldr}(x, y) = L_S(x, y)/(V_{curr} + 1)$
14: $TEX_{ldr}(x, y) = compressRGB(TEX_{hdr}, L_{ldr}, L_M, (x, y))$

Starting from a scale of $s_{curr} = 1$, and increasing scale size with the relation $s_{next} \approx 1.6 s_{curr}$, Algorithm 7 loops until $activity$ exceeds a predefined threshold ϵ, or total of 8 scales have been traversed, whichever comes first. Lines 6 and 7 use the SAT L_{SAT} to

compute the average scaled luminance within square regions centered at (x,y), with edge lengths of s_{curr} and s_{next}, saving the results in V_{curr} and V_{next}, respectively. This operation can be interpreted as constructing a new layer in the box-filter pyramid on the left in Figure 5-18 in each iteration. Once V_{curr} and V_{next} have been computed, a measure of difference, or *activity*, between them is computed. Line 8 computes this difference using the function $getActivity$, which evaluates the center-surround function given in Equation 4.5 (Section 4.1.2) and stores the result in the variable $activity$. Once $activity$ exceeds the threshold ϵ or all 8 scales have been traversed, the most recently computed value of V_{curr}, i.e the largest identified quadratic region of approximate isoluminance surrounding the current pixel (Figure 5-18), is used as the basis of local luminance compression (Equation 4.7 in Section 4.1.2). Finally, the LDR output luminance L_{LDR} is used by the function $compressRGB$ to compress in the input HDR red, green and blue channels into the displayable range, according to the Equation given below [RWP+06, p.229]:

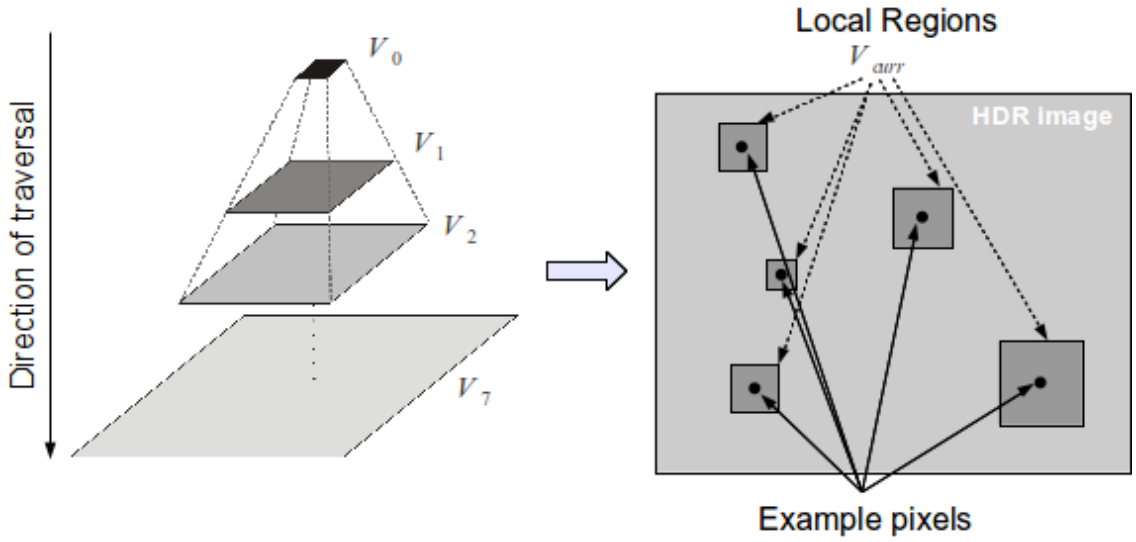


Figure 5-18: Conceptual illustration of the dodging-and-burning procedure in Algorithm 7. For each pixel in the input image, a box filter pyramid is iteratively constructed (left), starting from the top (smallest scale) and iterating to the bottom (largest scale). In each iteration, a metric of difference, or activity, between the newly constructed layer V_{next} and its immediate predecessor V_{curr} is evaluated. When this value exceeds a given threshold, V_{curr} is identified as the largest region of approximate isoluminance, or Local Region (right), surrounding the target pixel. Once identified, the Local Regions are used for local luminance compression according to Equation 4.7.

$$\begin{bmatrix} R_{LDR} \\ G_{LDR} \\ B_{LDR} \end{bmatrix} = L_{LDR} \begin{bmatrix} (R_{HDR}/L_W)^{\gamma} \\ (G_{HDR}/L_W)^{\gamma} \\ (B_{HDR}/L_W)^{\gamma} \end{bmatrix} \tag{5.5}$$

where R_{HDR}, G_{HDR} and B_{HDR} are the red, green and blue components of a given pixel in the input HDR texture, L_W is that pixel's luminance, γ is a per-channel gamma correction parameter, and R_{LDR}, G_{LDR} and B_{LDR} are the resulting red, green and blue components of the resulting LDR pixel.

5.5 A shader tone mapping module

The main focus of this thesis is an investigation into tone mapping with CUDA. The motivation behind investigating local tone mapping with CUDA originates from the hypothesis that SAT generation - a crucial bottleneck in *Slomp and Oliveira*'s method - can be accomplished more efficiently using CUDA than with shaders, due to CUDA's exclusive access to high speed shared memory. A paper published by *Harris et al.* [HSO07] showed very promising results when implementing *Sengupta's* [SLO06] work-efficient parallel scan algorithm in CUDA.

In order to determine whether the CUDA tone mapping module presented in Section 5.4 does in fact deliver greater performances than are achievable with shaders, it must be compared to an equivalent, shader tone mapping implementation. Since, as far as could be determined, no such implementation exists, a shader tone mapping module, which uses Sengupta's work-efficient parallel scan algorithm for SAT generation (described in Section 5.2.4), was developed for this thesis.

To simplify the testing and comparison process, the shader tone mapping module was designed to be integrated into interactive HDR applications in same way as the CUDA tone mapping module shown in Figure 5-15. A host OpenGL application renders its HDR scene to texture, which is passed to the tone mapping module. The module tone maps the HDR texture, rendering the output to another texture, which is then read back by the host application and displayed on the monitor.

Like the CUDA module, the shader tone mapping module consists of a collection of parallel shader programs, together with CPU-based sequential code for invoking and manipulating them. Figure 5-19 shows the internal shader module structure. It is clear that the shader module design in Figure 5-19 is very similar to that of the CUDA module, shown in Figure 5-15. Indeed, each shader program or submodule performs the same operation as the equivalently named kernel in the CUDA module. For a detailed description of what each shader program does, refer to Section 5.4.

The main difference in structure between the shader and CUDA modules is that the shader module uses no reduction program. Rather than implementing a shader-based reduction technique to derive the scene key from the luminance map, a simple yet efficient method was employed: *mipmapping*. Mipmapping is a simple, hardware supported technique, typically used for anti-aliasing, by which a series of successive resamplings are generated for a given texture, where each resampling is half the resolution of its predecessor. Each texel in a given mipmap layer is the average of a number of its predecessor's texels. Therefore, the average texel value of a given texture can be computed by generating its mipmap and sampling the top-most, 1x1-dimensioned layer.

Unlike the CUDA luminance extraction kernel, which produces an array of single, floating point values in global GPU memory, the shader luminance computation program produces a texture with multiple channels; the computed luminance value of the each input texel is stored in the red output texture channel, while the green channel is used for its logarithm. Therefore, the scene key can be computed simply by generating a mipmap of the luminance map and reading the green channel of the top mipmap layer. This only gives correct results for square textures; more sophisticated reduction algorithms are required for rectangular textures. Nonetheless, for the purpose of performance comparison against

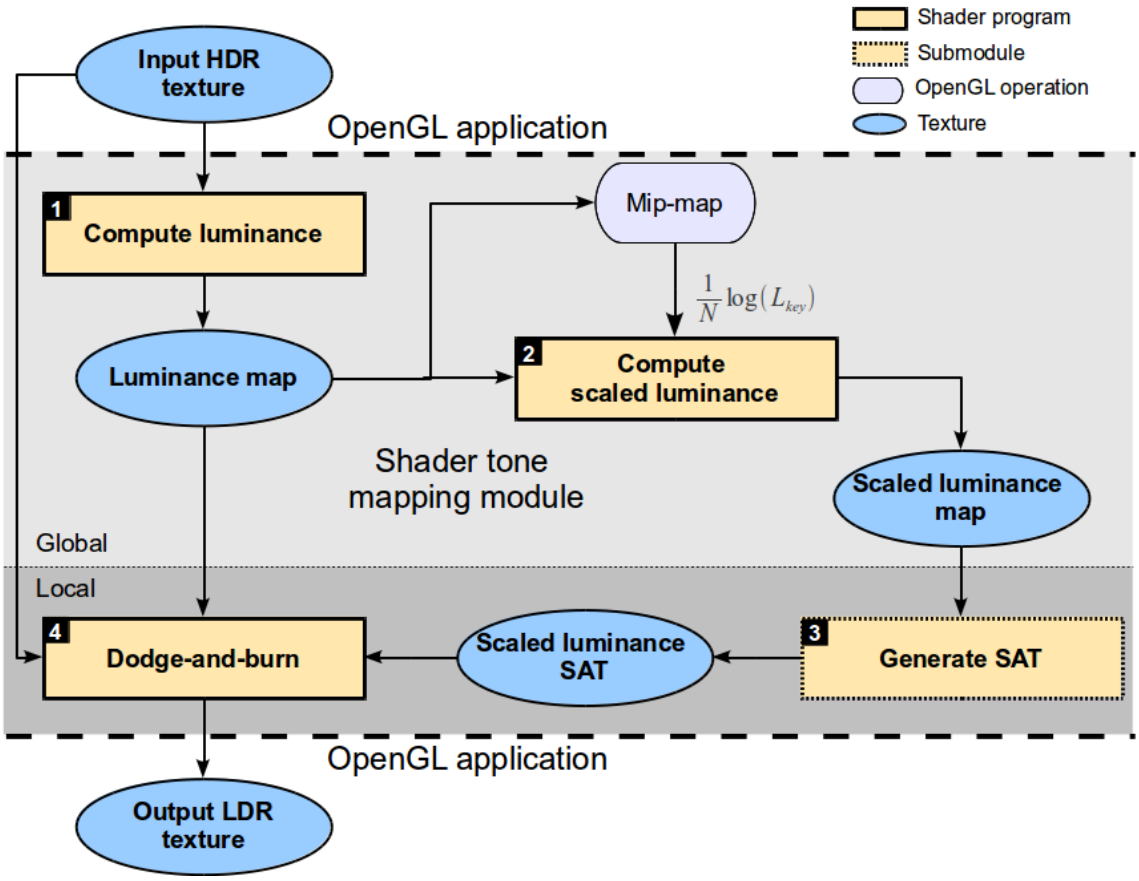


Figure 5-19: The design of the shader tone mapping module. Blue ovals represent textures, solid yellow rectangles represent shader programs that process these textures, and the "Mipmap" element represents a hardware-based mipmap operation. The dashed rectangle, denoted "Generate SAT" represents a submodule, which is itself a collection of sequential code and shaders. Shaders in the light, "global" zone apply the same operation to all input texels, and can be used for both global and local tone mapping operators. Shaders in the darker zone, on the other hand, are required specifically for local tone mapping. The dashed line represents the border between host OpenGL application and the tone mapping module.

the CUDA module, scene key computation by mipmapping is sufficient.

Another notable difference between the shader and CUDA modules is the implementation of the SAT generation step. Both use flavours of *Sengupta*'s algorithm, presented in Section 5.2.4, to generate row and column scans from their input textures. Because shaders work solely in texture memory, the shader SAT generation step, unlike its CUDA equivalent, does not need to transpose its results during computation. This means that the shader SAT generation step performs less work than its CUDA counterpart. The work that it does perform, however, is less efficient than that done during CUDA SAT generation. Furthermore, the shader SAT generation method is subject to a significantly larger memory footprint, resulting from the additional textures that must be allocated to store intermediate results (see Section 5.2.4).

Further details on the implementation of the individual steps in Figure 5-19 are given in Chapter 6.

5.6 Summary

In this chapter, the approach taken for meeting the goals set forth in Chapter 3 has been discussed.

Section 5.1 provided an overview of *Slomp and Oliviera*'s [SO08] approximation of Reinhard's local tone mapping operator, which uses high speed SAT-based box filters, rather than Gaussian convolution, for Local Region identification. *Slomp and Oliveira*'s method was analysed in search of optimization opportunities, revealing the SAT generation procedure as a potential performance bottleneck.

After introducing the parallel *scan* operation, the fundamental operation used for SAT generations, Section 5.2 showed that the SAT generation method used by *Slomp and Oliveira* is not *work-efficient*. Accordingly, a more recent parallel scan method, proposed by *Sengupta et al.* [SLO06], was introduced and it was shown that this method is work-efficient. It was then revealed that further performance gains, as well as a reduced memory footprint, can be achieved by taking advantage of certain GPU hardware features made available to the programmer through CUDA.

Motivated by the promising results of high speed CUDA-based scan computation on large datasets, as reported by *Harris et al.* [HSO07], and early reports of poor CUDA/OpenGL interoperability [LU08], a small study was undertaken in Section 5.3 to investigate the suitability of using CUDA for post-processing OpenGL applications. The experiments in Section 5.3 showed that the current generation of CUDA - Release 3.2 - is sufficiently mature for post-processing OpenGL applications at interactive frame rates.

To test the hypothesis that *Slomp and Oliveira*'s tone mapping method can be improved using modern, work-efficient algorithms implemented in CUDA, a CUDA module was developed for tone mapping interactive OpenGL applications. Section 5.4 presented the design of this module. Following a discussion of its intended use with OpenGL applications, an overview of the internal module design was given. Each kernel used in the module was then described in detail.

The tone mapping module was developed in CUDA, on the presumption that using CUDA for tone mapping would result in performance gains unattainable by shaders. Since the CUDA module uses a different algorithm for SAT generation to that of *Slomp and Oliveira*, comparing the CUDA module directly to a shader implementation of *Slomp and Oliveira*'s original method would be an unspecific comparison; it would be impossible to determine whether performance differences result from the different SAT-generation algorithms, or the different programming environments. Therefore, Section 5.5 presents a shader tone mapping module, which uses the same SAT-generation algorithm as the that of the CUDA module.

6 Implementation

This chapter presents the implementational details of the CUDA and shader tone mapping modules introduced in Chapter 5.

Section 6.1 describes in detail how each kernel in the CUDA module was implemented. Similarly, Section 6.2 covers the implementation of each shader program in the shader module. Finally, Section 6.3 introduces a custom prototyping platform for developing and testing high performance tone mapping operators.

6.1 Tone mapping with CUDA

Section 5.4 discussed the design of the CUDA module, introducing and discussing the five CUDA kernels used for local tone mapping. The task performed by each process was described on an abstract, algorithmic level. Even when implementing simple and efficient algorithms in CUDA, however, care must be taken to make efficient use of hardware. In particular, suboptimal memory access patterns can cause a CUDA kernel to deliver a fraction of expected performance. As an example, the NVIDIA technical report *Optimizing Parallel Reduction in CUDA* by Mark Harris [Har08] shows how loop unrolling and optimization of memory access patterns can increase the throughput of a parallel reduction operation (discussed in Section 5.2.4) by a factor of 30. Consequently, much of the work in this research is in optimization of implemetation-specific details within the five CUDA kernels.

This section discusses some of the technical aspects of the implementations of the CUDA kernels in Figure 5-16.

6.1.1 Luminance extraction

The luminance extraction kernel is one of the simplest kernels in the tone mapping module, making it a suitable initial example for introducing the basic CUDA image processing concepts used by all kernels discussed in this thesis. Listing 6.1 lists the complete code for Kernel 1.

```
1 __global__ void computeLuminance(float* lumMap, int lumMapPitch)
2 {
3     // map current thread to a texel in the input texture
4     int tx = threadIdx.x; // thread x position within block
5     int ty = threadIdx.y; // thread y position within block
6     int bx = blockIdx.x;  // block x position within grid
7     int by = blockIdx.y;  // block y position within grid
8     int bw = blockDim.x;  // block width
9     int bh = blockDim.y;  // block height
```

```
10      int x = bx*bw + tx;    // map thread x to texture x
11      int y = by*bh + ty;    // map thread y to texture y
12
13      // get current pixel value
14      float4 res = tex2D(inTex, x, y);
15
16      // compute luminance
17      float lum = RGB2LUM(res.x, res.y, res.z);
18
19      // compute linear output index
20      // note: we are using pitch-linear memory
21      int i = y*lumMapPitch+x;
22
23      // save logarithm of luminance in the luminance map
24      lumMap[i] = log(lum+1.0);
25  }
```

Listing 6.1: CUDA code for luminance computation kernel.

Kernel 1 is implemented as a global CUDA C function taking two arguments: `lumMap`, a pointer to the target luminance map (L_M in Section 5.4.1), and a parameter `lumMapPitch` required for correctly indexing this array (explained further below).

Lines 4 to 11 map each thread in the CUDA thread matrix (see Section 2.3.3) to a texel in the input texture.

Once the thread-to-texel correspondance has been determined, the correct texel is sampled from the HDR input texture on Line 14. The symbol `inTex` is a globally accessable handle to the input HDR texture residing in GPU texture memory. The CUDA API function call `tex2D(inTex, x, y)` returns a 4 element vector of 32 bit single precision floats, representing the red, green, blue and alpha channels of the texel at position (x,y) in the input texture. After the correct input texel has been retrieved and stored in `res`, Line 17 uses the globally defined macro `RGB2LUM(r,g,b)` to compute the texel's luminance according to Equation 2.1.

The final step in Kernel 1 is to store the logarithm of the computed texel luminance in the luminance map `lumMap`; this is done on Lines 21 to 24. The most interesting part of this operation is Line 21, which computes the location in the luminance map where the result of the current thread should be stored. The array index computation on Line 21 is necessary because the array `lumMap`, which is a 2D map of 32 bit floating point luminance values, is allocated as a single dimensional array. In general, allocating single dimensional arrays to represent 2D memory regions is common practice in CUDA.

In order to access element (x, y) of a 2D image stored in a single dimensional array, the correct array index i is computed as follows:

$$i = imgw * y + x \tag{6.1}$$

where $imgw$ is the width of the image.

At first inspection, it appears that one would need to allocate `lumMap` with dimensions wxh, where w and h are the dimensions of `inTex`, and use Equation 6.1 with $imgw = w$

for coordinate to index mapping throughout the program. Although this would produce correct results, significant performance penalties are likely to occur. As is often the case with CUDA, additional considerations must be made.

As mentioned in Section 2.3.3, for maximum kernel throughput, accesses to global memory must be *coalesced*, i.e the address of each memory access must be aligned with some transaction size parameter λ. When allocating a new block of memory, the CUDA memory allocation function, `cudaMalloc()`, ensures that the starting address of the newly allocated memory is correctly aligned. If, however, $imgw$ in Equation 6.1 is not a multiple of λ, then all 2D memory accesses to elements in rows where $y \bmod \lambda \neq 0$ will be uncoalesced.

The solution to this problem is to pad each row of the 2D memory map with a number of additional elements, denoted $padding$, such that $(w + padding) \bmod \lambda = 0$. CUDA provides a special function, `cudaMallocPitch()`, which allocates padded memory called *pitch linear* memory. The total width of each row, including padding, is called the memory *pitch*. Figure 6-1 illustrates this concept.

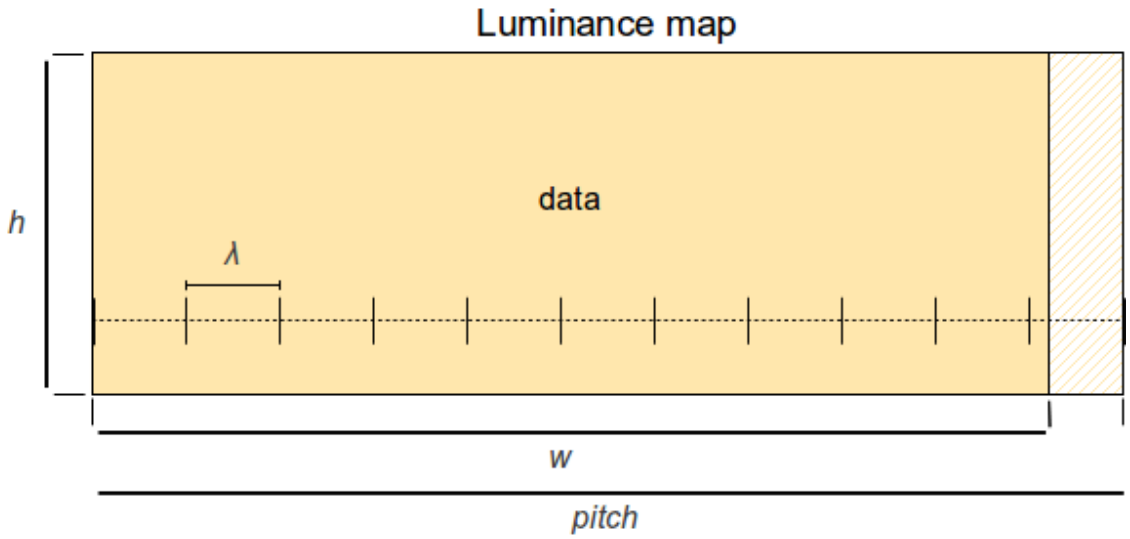


Figure 6-1: The luminance map is stored using pitch linear memory. Additional padding is added to each row such that $pitch \bmod \lambda = 0$, ensuring that all address accesses are aligned with λ and are thus coalesced.

The output array `lumMap` has been allocated as pitch linear memory, where the kernel parameter `lumMapPitch` is the pitch. Line 21 maps the coordinates (x, y) to the appropriate output index in `lumMap`, using `lumMapPitch` as image width. Therefore, all writes to the output luminance map are coalesced.

6.1.2 Reduction

The whitepaper *Optimizing Parallel Reduction in CUDA* by Mark Harris [Har08] describes in detail how a reduction method can best be performed using CUDA. The code used for the reduction kernel in the CUDA tone mapping module is from this whitepaper, which is supplied with the CUDA SDK. This section gives an overview of this method,

outlining the main differences between Harris' CUDA reduction and common shader-based reduction methods. For more detail on the reduction kernel implementation, including how shared memory bank-conflicts are avoided, and thread divergence is minimized, see [Har08].

Although the reduction algorithm described in Section 5.2.4 is well suited for implementation with shaders, when implementing this method in CUDA, a number of fundamental modifications are required. Unlike shader-based reduction methods, which require allocation of additional textures for storage of partial results, CUDA can reduce arrays in-place. In-place reduction requires multiple passes; each pass breaks the input array into segments, copies the segments into blocks of shared memory, reduces each segment within shared memory to a single sum, and writes the resulting sum value back into the original input array. Figure 6-2 shows this procedure. After $\lceil log_2(n) \rceil$ such passes, where n is the size of the input array, the first element of the input array contains the sum of its original contents.

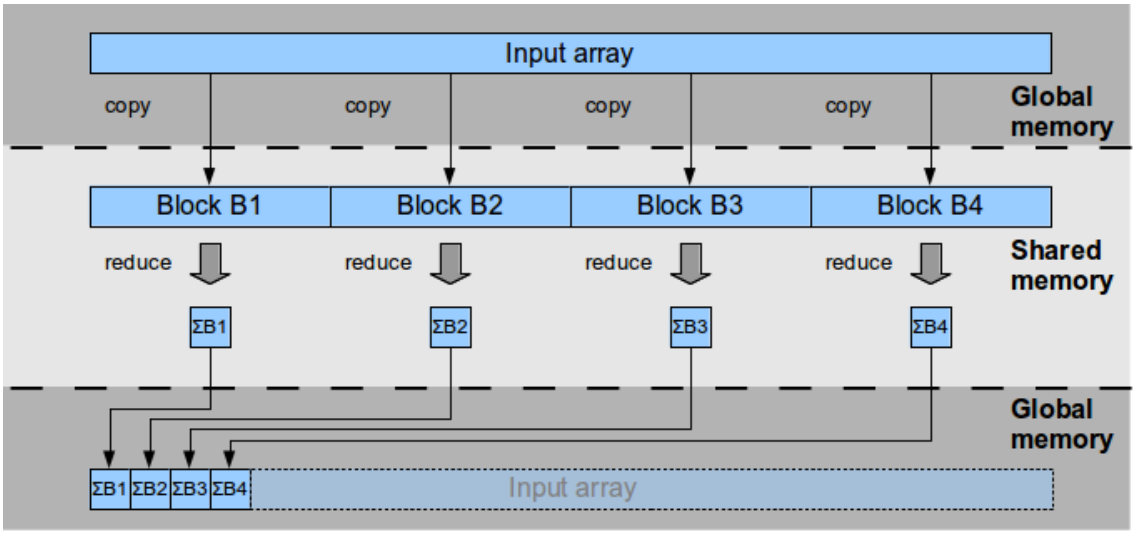


Figure 6-2: A single pass of in-place reduction using CUDA. The input array, stored in global memory, is divided into segments, which are copied as "blocks" into shared memory. In this example, the input array is small enough to be divided into 4 blocks; generally many more would be required. These blocks are then reduced efficiently within shared memory, resulting in a single sum per block. Finally, the block sums are written back into the input array in global memory. After repeating this process $\lceil log_2(n) \rceil$ times, the first element of the input array contains the complete sum of the original input array.

Reducing blocks within shared memory can be accomplished considerably faster than would be possible with shaders, which are forced to perform similar operations solely using texture memory and registers. However, in order to benefit from the theoretical performance gains promised by shared memory access, great care must be taken to organize memory access patterns such that shared memory bank conflicts, as well as thread divergence (see Section 2.3.3), are avoided. In particular, the reduction algorithm used by *Sengupta et al.* [SLO06], discussed in Section 5.2.4 and illustrated in Figure 5-5, involves memory access patterns that lead to a high level of thread divergence. This arises from the fact that each element in a block is processed by a dedicated thread. Each reduction step reduces the size of the input array by half. Therefore, for each reduction step $i + 1$,

half of the threads used in step i have no work to do, and consequently remain idle. If each step reduces the input array by pairwise addition between direct neighbours, and consecutive threads are used to process consecutive array elements, every second thread in reduce step $i + 1$ will be idle. Because all threads in each half-warp (or warp, depending on the GPU architecture) execute in lock-step, and each warp must be re-executed for each thread divergent thread, this situation will incur significant performance penalties. To minimize thread divergence, *Harris* [Har08] introduced an alternative, sequential addressing scheme. Harris' method is shown in Figure 6-3 for an example array containing 8 elements. By adding each element to a neighbour at a distance of $n/2$ elements to its right, thread divergence within each warp is kept to a minimum.

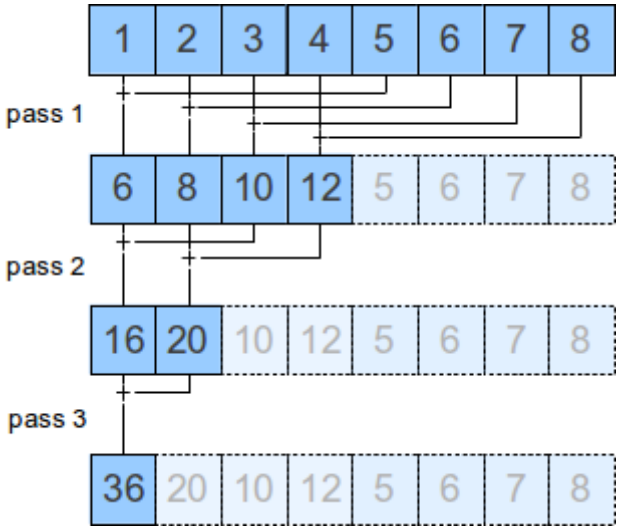


Figure 6-3: Reduction in shared memory, using sequential addressing for minimal thread divergence.

In addition to the fundamental modifications to traditional, shader-based reduction algorithms, Harris also employed a number of low-level CUDA optimization techniques such loop-unrolling to achieve a practical throughput near that of the theoretical maximum for his hardware. For more information, see [Har08].

6.1.3 Scaled luminance computation

The scaled luminance computation kernel applies the same principles as the luminance computation kernel (Section 6.1.1). Taking the luminance map from Section 6.1.1 and its global log-average, computed using the reduction described in Section 6.1.2, the scaled luminance computation kernel computes the scaled luminance (Equation 5.4) of each pixel in the luminance map, and stores the results in a scaled luminance map. Like the luminance map, the scaled luminance map is an array allocated as pitch-linear memory, containing at least wxh elements, where w and h are the dimensions of the input HDR texture. Listing 6.2 shows the most important code used by this kernel.

```
1 __global__ void scaleLuminance(float* scaledLumMap,
2                                 float* lumMap,
3                                 float lavg,
4                                 float key,
5                                 size_t pitch)
```

```
6  {
7     // compute thread <-> pixel corresponance
8     //see computeLuminance(...)
9     ...
10
11     // save scaled luminance
12     scaledLumMap[i] = exp(lumMap[i])*(key/lavg);
13 }
```

Listing 6.2: Scaled luminance computation kernel.

The first few lines of the kernel `scaleLuminance` in Listing 6.2 map each CUDA thread to an element in the input luminance map, using Equation 6.1, with w set the input parameter `pitch`. The 1D index in the luminance and scaled luminance maps is stored in the variable `i`. Given `i`, Line 12 uses Equation 4.2 to compute the scaled luminance for each value in the luminance map, storing the results in the output scaled luminance map.

6.1.4 SAT generation

Section 5.4.4 explained the approach used for constructing Summed-Area Tables using CUDA. First, scans for each row in the input image are computed in parallel using *Harris et al.*'s [HSO07] work-efficient scan. Second, in order to respect CUDA memory coalescing requirements, the result is transposed. Finally, Harris' scans are applied to each row of the transposed results simultaneously.

Up to this point, Harris' scan algorithm has been briefly described as an in-place, CUDA-based modification of *Sengupta et al.*'s work-efficient algorithm (described in Section 5.2.4). The modifications necessary for updating an algorithm designed for shaders to run efficiently in CUDA can be complex. Indeed, a closer look into Harris' paper reveals significant low-level optimization work required to reach the promised performance. Fortunately, Harris, together with the other authors of [HSO07], have created a CUDA library, called *CUDPP*, to encapsulate the scan functionality described in a number of their papers [SHZ+07] [SHG08] [SHG09] [TW08]. CUDPP was used in the CUDA tone mapping module for parallel scan computation.

The SAT generation step is implemented as submodule, which, like the overall CUDA tone mapping module shown in Figure 5-16, is made of up a collection of CUDA kernels as well as C++ code for manipulating them. The functionality for SAT generation, including CUDA kernels for computing an image transpose and parallel scan operations, is interfaced through a C++ class. Figure 6-4 uses UML to illustrate the important parts of this interface. The interface for all SAT operations is specified by an abstract class named `SAT`. The class `SATHarris` derives `SAT`, implementing the `compute()` method in CUDA. Note that the `satHarris` class is contained in a namespace `libCUDA`, so that implementations based on other technologies, such as shaders, can easily be added to the system. Indeed, a class implementing *Hensley et al.*'s [HSC+05] non-work efficient SAT generation method used by *Slomp and Oliveira* [SO08] will be required for adequately evaluating the results in Chapter 7.

Given an input scaled luminance map of size wxh, passed to the `SATHarris` object via its `setInput(...)` method, the `init` method allocates space for two additional memory

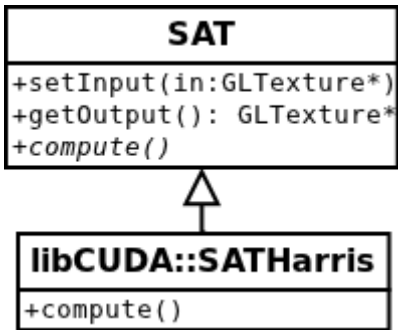


Figure 6-4: The SAT construction class UML.

maps, S_1 and S_2, of dimensions $(w + p_1)$ x (h) and $(h + p_2)$ x (w), respectively, where p_1 and p_2 are padding parameters necessary to meet memory coalescing requirements for pitch-linear memory (Section 6.1.1). The first map, S_1 is necessary for storing the row-scans produced from the input scaled luminance map, without overwriting the original input. Although CUDPP computes scans in-place, in the sense that an input array is divided into segments, which are loaded into shared memory where in-place partial result computation is performed, CUDPP writes the final scan results to a second, output array in global memory. Therefore, a second map, S_2, with transposed dimensions to S_1, is required for ping-ponging the partial results generated during the three set scan-transpose-scan SAT generation process.

The `setInput()` and `init()` methods are designed to be called once during program initialization. Once the program has been set up and has commenced simulation, the `compute()` method called from the main loop. The essential parts of this function are shown in Listing 6.3

```
1  // set the in/out buffer indices
2  _in = 0;
3  _out = 1;
4
5  // transposed output dimensions
6  _ow = _h;
7  _oh = _w;
8
9  // first row scan: _input -> _sat[_in]
10 cudppMultiScan(_scanPlan1, _sat[_in], _input, _w, _h);
11
12 // transpose _sat[_in] -> _sat[_out]
13 SAT_transpose_fast(_sat[_out], _sat[_in], _outPitch, _w, _h);
14 std::swap<int>(_out, _in); // swap buffer indices
15
16 // second row scan: _sat[_in] -> _sat[_out]
17 cudppMultiScan(_scanPlan2, _sat[_out], _sat[_in], _ow, _oh);
18 ...
```

Listing 6.3: SAT generation using CUDPP.

Lines 2 to 7 are used to track the input and output datasets and dimensions throughout the computation. Row scans are computed on Lines 10 and 17 using the CUDPP function `cudppMultiScan(...)`, which implement Harris' work-efficient algorithm described in

[HSO07]. The parameters `_scanPlan1` and `_scanPlan2`, of type `CUDPPHandle`, point to so-called *CUDPP Plan* structures, which provide important details on how the scan is to be performed (such as data type and scan operator). The remaining parameters are the input and output dataset, as well as the dimensions of the output dataset.

The function `SAT_transpose_fast(...)`, called on Line 13, interfaces a CUDA kernel for computing a high speed matrix transpose using shared memory. The implementation of this kernel is specified in the NVIDIA whitepaper *Optimizing Matrix Transpose in CUDA* by Greg Ruetsch and Paulius Micikevicius [RM10], which is supplied with the CUDA 3.2 SDK toolkit.

6.1.5 Dodging-and-burning

The final, dodging-and-burning kernel is the most complex kernel developed for the CUDA tone mapping module. Despite the apparent simplicity of the generic dodging-and-burning algorithm presented in Section 5.4.5, implementing this algorithm efficiently in CUDA presents a number of challenges.

The dodging-and-burning kernel inputs are: the original LDR texture, the luminance map produced by the *Compute luminance* kernel, the scaled luminance map computed by the *Compute scaled luminance* kernel and the scaled luminance SAT computed by the *Generate SAT* kernel. Since a large volume of memory must be processed by the dodging-and-burning kernel, to avoid a bandwidth bottleneck, care must be taken when designing memory access patterns. Furthermore, because the luminance compression factor varies from pixel to pixel in local tone mapping, a number of branches are necessary in the kernel, which are liable to cause thread divergence. It should also be noted that a significant amount of arithmetic must be performed to repeatedly evaluate the center-surround function given by Equation 4.5 in Section 4.1.2. Each thread block has access to limited resources; as kernel complexity increases, the number of registers available to each GPU processor decreases, reducing kernel *occupancy* [NVI11, p. 86]. Eventually, when insufficient registers are available, internal kernel variables must be allocated in *local memory*, which has the same latency as global memory (local memory is in fact a region in global memory).

Motivated by the complexity of the doding-and-burning computation, a number of implemenations of the dodging-and-burning kernel have been considered. This section begins by presenting a "naive" implementation, which most closely represents the algorithm presented in Section 5.4.5. After examining the sub-optimal performance of this kernel, a number of optimizations are presented in the remaining subsections.

6.1.5.1 A naive implementation

Listing 6.4 shows a first, naive dodging-and-burning kernel implementation.

```
1 __global__ void dodgeAndBurn_Naive(
2     uchar4* ldr_RGB_out,      // output LDR image
3     float* lumMap,            // luminance map
4     float* sLumMap,           // scaled luminance map
5     size_t mapPitch,          // pitch (width+padding) of input
```

```
    float lavg,                 // luminance log average
    float* sat,                 // scaled luminance SAT
    size_t satPitch,            // SAT pitch
    float alpha,                // exposure parameter
    float phi,                  // sharpnes parameter
    float epsilon,              // activity threshold
    float gamma,                // gamma adjustment
    int imgw                    // output image width
)
{
  // compute thread to pixel correspondance
  // see computeLuminance()
  // current thread coordinate (x,y)
  int outIndex = y*imgw + x;
  int mapIndex = y*mapPitch + x;

  // retrieve luminance and scaled luminance
  float ls = sLumMap[i];

  // box filter scales, not including first scale of 0
  float scales[7] = {3.0,5.0,7.0,11.0,17.0,25.0,39.0};

  int s = 0;  // initial scale 0
  float scaleCenter = 0;
  float scaleSurround = scales[0];
  float V_center = ls;
  float V_surround = getAverage(sat, satPitch, y, x,
                                scaleSurround);

  // compute center-surround function
  float w = CENTER_SURROUND(...); // details hidden in macro

  // begin traversing box filter pyramid
  while (abs(w) < epsilon && s < 7){
    scaleCenter = scales[s];
    scaleSurround = scales[s+1];
    V_center = surround;
    V_surround = getAverage(sat, satPitch, y, x, scaleSurround);

    w = CENTER_SURROUND(...); // details hidden in macro
    s++;
  }

  // now do local tone mapping
  float ldr = (ls/(1+V_center))*255;

  // use ldr to compress input RGB (details hidden in macro)
  float4 res = tex2D(inTex, x, y);
  float lum = exp(lumMap[i]);
  ldr_RGB_out[outIndex] = COMPRESS_RGB(res, lum, ldr, gamma);
}
```

Listing 6.4: A naive dodging-and-burning CUDA kernel.

Lines 2 to 13 in Listing 6.4 are the kernel input arguments, including all input datasets, the parameters α and ϕ as defined in Equation 4.5, ϵ as defined in Equation 4.6 and γ as given in Equation 5.5.

As usual, the first few lines of the kernel are dedicated to mapping each CUDA thread to a 2D coordinate in the input image. This code is identical to Lines 4 to 11 in Listing 6.1; it has been emitted here for the sake of brevity. The 2D coordinate of the current thread is then mapped to indices in the input and output image arrays by Lines 19 and 20, respectively. Although all input and output images are of the same resolution, two seperate mappings are required. This is because both input maps - the luminance and scaled luminance maps - have been allocated as pitch-linear memory (Section 6.1.1), while output is written directly to a Pixel Buffer Object (PBO) (see Section 5.3.2), which, unlike the input maps, contains no horizontal padding.

Once the correct input and output data indices for the current thread have been determined, Line 23 reads the scaled luminance for the current pixel from the luminance map `sLumMap` and stores it in the local variable ls. Because the thread-to-index mapping has been computed such that consecutive threads within each warp read horizontally consecutive input elements, and `sLumMap` is allocated as pitch-linear memory, we can rest assured that this read will be coalesced for all threads.

The scaled luminance for the current pixel, ls, is used as the bottom scale of the 8-scale box filter pyramid required for automatic dodging-and-burning. The remaining 7 are defined on Line 26. Each element i in the scale array `scales` represents the edge length of layer i in the box filter pyramid. The values used here are the same as used in the investigation by *Linnemann et al.* [LWR+09], where box filter based tone mapping with these scales was shown to give desirable results.

Lines 28 to 33 initialise and compute the scaled luminance values for the first two scales of the box filter pyramid, and Line 36 evaluates the *activity* between them by evaluating the center-surround function defined in Equation 4.5, storing the result in the variable w. To avoid clutter, the center-surround arithmetic has been hidden inside a macro. Of particular interest is the function `getAverage` on Line 32. This function takes the scaled luminance SAT, a 2D position (x, y) and an edge length l, and returns the average scaled luminance within a square region centered at (x, y) with edge length s. Note that the transpose of the current thread's coordinates is passed to `getAverage`. This is necessary because, as described in Section 6.1.4, the input SAT is stored in transposed format.

The procedure of constructing additional pyramid layers and using them to search for significant changes in average scaled luminance is iteratively repeated on Lines 39 to 47 using a `while` loop. Prior to commencing each loop, the most recently computed center-surround result, `w`, is compared to the input threshold, `epsilon`. Once the threshold has been exceeded, or all 8 scales of the box filter pyramid have been evaluated, the pyramid construction and evaluation procedure is aborted. At this point, `V_center` contains the average scaled luminance of the largest quadratic region of approximate isoluminance surrounding the current pixel. Therefore, `V_center` is used on Line 50, where the output LDR luminance is computed by applying Equation 4.7.

Finally, Lines 53 to 55 use the newly computed LDR luminance value to compress the red, green and blue channels of the HDR input pixel using Equation 5.5. To accomplish this, Line 53 samples the input HDR texture at the coordinates (x, y) to which the current thread has been assigned, storing the resulting red, green, blue and alpha channel values in the `float4` vector `res`. The input HDR luminance, required for RGB compression, is retrieved from the luminance map on Line 54. Since the luminance map contains the logarithm of the input HDR luminances, the original luminance value can be attained by applying the exponential function. Line 55 uses the macro `COMPRESS_RGB` to compress the input HDR pixel using Equation 5.5 and saves the result in `ldr_RGB_out`. Writing to `ldr_RGB_out`, which points to a PBO allocated by the host OpenGL application, is the final action taken by the CUDA tone mapping module.

6.1.5.2 Naive kernel performance

Figure 6-5 lists the execution times of the naive kernel implementation shown in Listing 6.4, for a number of input resolutions. All execution times were measured on the system detailed in Appendix A.

Resolution	Kernel execution time (ms)
512x512	5.4
1024x1024	28.7
2742x2048	139.4

Figure 6-5: Average kernel execution times for the naive dodging-and-burning kernel.

It can be seen that, if an application's sole task were to execute this kernel on a 1024x1024 input image, the maximum achievable frame rate would be $1/28.7 \approx 35$ fps. For larger images, the minimum frame rate of 30 fps for interactive applications is not reachable. This is unacceptable, considering that *Slomp and Oliveira* [SO08] reported frame rates of up to 102 fps for a 1024x1024 scene using non-work-efficient SATs.

A common practice when optimizing a CUDA kernel is to determine whether it is *memory bound*, i.e its main bottleneck is caused by accessing memory, or *instruction bound*, i.e most of the kernel execution time is spent performing arithmetic [Mic10]. A *balanced* kernel contains memory access and arithmetic in equal measure, where the arithmetic is performed while global memory is being fetched, hiding memory latency.

Whether a kernel is memory or instruction bound can be determined by measuring its execution times when all arithmetic-based instructions, or memory-based instructions have been removed. Figure 6-6 shows the execution times of these modified kernels. It can clearly be seen that the naive dodging-and-burning kernel is memory bound.

The input HDR image, as well as the luminance and scaled luminance maps, are each sampled once per thread. The thread-to-coordinate mapping computed in the beginning of the kernel ensures that these reads are all coalesced, so little can be done to increase their efficiency.

Each thread performs one write to the output PBO. This can also not incur a significant performance penalty, considering the execution times reported in Section 5.3 for simple

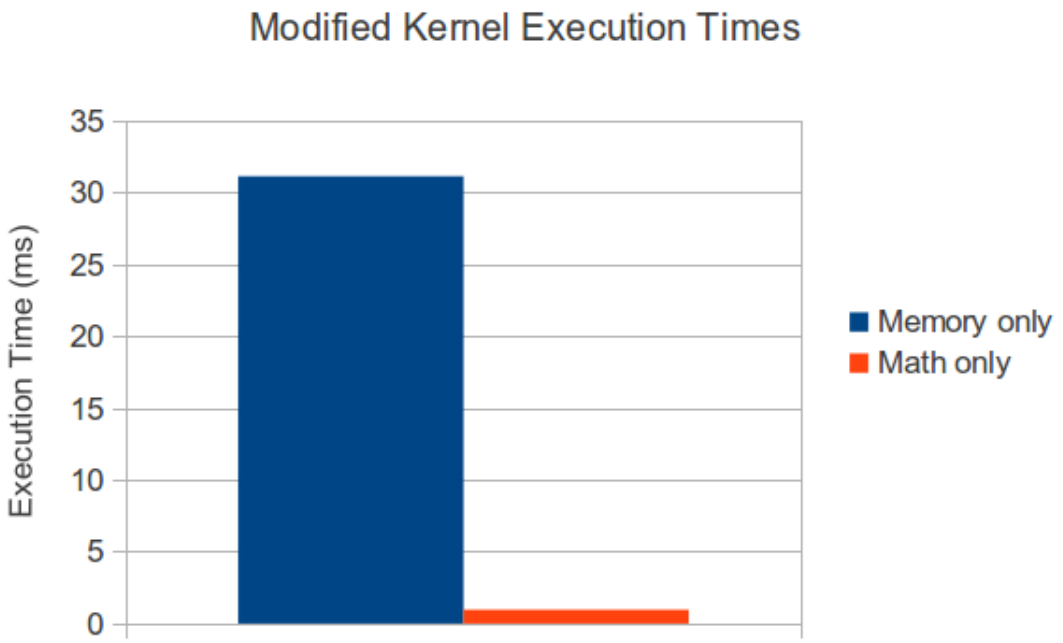


Figure 6-6: Execution times for modified dodging-and-burning kernels operating on a 1024x1024 input image. The blue bar on the left shows the execution time of the naive kernel, modified such that all complex arithmetic instructions have been removed. Similarly, the red bar on the right shows the execution time of kernel modified such that only complex arithmetic is executed. Comparing the two shows that the naive dodging-and-burning kernel implementation in Listing 6.4 is heavily memory bound.

kernels that write to an output PBO. This leaves the remaining memory accesses, i.e accesses to the scaled luminance SAT, as the only possible memory bottleneck.

Indeed, a closer look at the SAT access patterns reveals that they are far from optimal for global memory. All reads from the input SAT are done by the `getAverage()` (Lines 32 and 43 in Listing 6.4) method. The average of any region within a given image can be accomplished by reading the region's 4 corner points from the image's SAT. Examining the code in Listing 6.4 reveals that `getAverage()` may be invoked up to 8 times per thread. Hence, each thread may invoke up to 28 SAT reads. If each SAT element is a 32-bit floating point value, and the application resolution is 1024x1024, up to 112 MB of global memory must be read per frame for SAT lookups. Therefore, in order maintain a minimum interactive frame rate of 30 fps, global GPU memory must be streamed at up to 3.36 GB/s just to support SAT reads!

6.1.5.3 Using texture memory

Figure (a) visualizes the SAT access patterns for the first few box filter computations for two threads belonging to the same warp, labelled A and B. It can be seen that, despite the level of scattering surrounding each single thread, there remains a high degree of coalescence for threads within a warp. Additionally, it is apparent that there is a high degree of *spatial locality* in the reads, i.e if a certain element is read, it is likely that another element nearby in the 2D space will also be read. A similar situation is depicted

in Figure (b), this time for diagonally neighbouring threads. In this situation, a number of elements are read by both threads. Using the naive kernel in Listing 6.4, these elements are read seperately by each thread. In general, when processing a frame, there is a very high amount of redudant SAT accesses; it would make sense to, once an element has been transferred from global memory, keep it for some time in high-speed, on-chip memory to service subsequent requests.

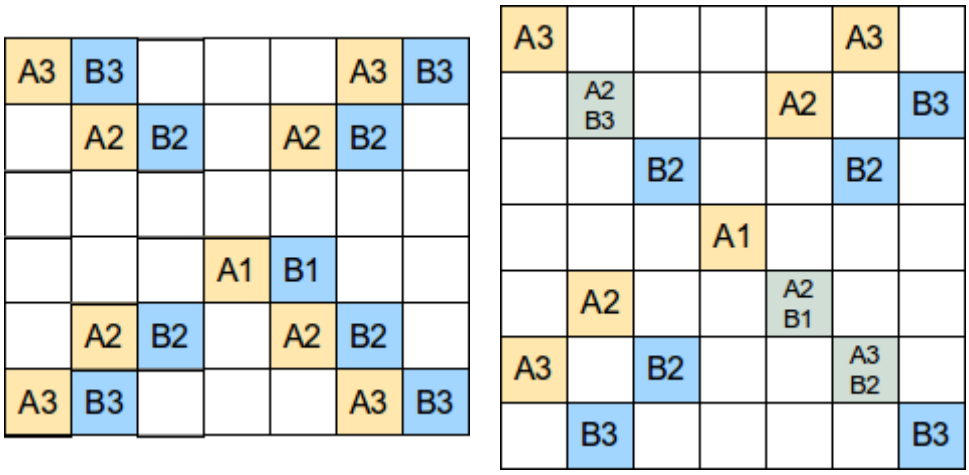


Figure 6-7: SAT access patterns for the first few dodging-and-burning iterations (Lines 39 to 47 in Listing 6.4) of two neighouring threads, denoted as A and B. On the left (a), threads A and B are horizontal neighbours, and in (b) they are diagonal neighbours. The grid of white squares represents the SAT; blue elements are read by thread A, orange elements are read by B and turqious elements are read by both. Each coloured element is annotated with the name of the thread that read it, juxtaposed with the iteration in which it was read. For example, if an element is read by thread A in its 3rd iteration, it is labeled $A3$.

Fortunately, *texture memory*, which has been available to programmers since CUDA Release 2.2, is *spatially cached*, which is precisely what is required to reduce global memory traffic in the dodging-and-burning kernel. Spatial caching means that, while texture memory resides off chip in the global memory space, texture reads are cached in high speed, on-chip memory. This cache is optimized for 2D spatial locality, so threads within a warp accessing memory that is spatially close together will benefit greatly.

Global memory can be read as texture memory by *binding* a texture pointer to a defined region in global memory. Once the texture pointer has been bound, the global memory can be accessed by sampling the texture using CUDA methods such as `tex2D(...)`. In addition to spatial caching, when sampling memory in this manner, common texture filtering and clamping operations can automatically be performed. In particular, for correct SAT evaluation, *clamp-to-edge* functionality is required. This was previously implemented manually in the `getAverage` function in Listing 6.4, and was the cause of performance degradation via resulting from warp divergence.

Binding a texture handle to the global memory containing the scale-luminance SAT, and updating `getAverage` in Listing 6.4 to access SAT elements through texture sampling, rather than reading them directly, leads to significant performance improvements. Figure

6-8 shows the average kernel execution times for a number of input resolutions, when using texture memory for SAT access.

Resolution	Kernel execution time (ms)
512x512	1.6
1024x1024	6.7
2742x2048	32.2

Figure 6-8: Average kernel execution times for a dodging-and-burning kernel that uses texture memory for SAT access.

The kernel execution times listed in Figure 6-8 for the texture memory dodging-and-burning kernel show a significant improvement over the naive kernel execution times presented in Figure 6-5. This owes to the high degree of spatial locality in the dodging-and-burning algorithm. By caching texture lookups, global memory traffic was significantly reduced. Figure 6-9 shows the number of cache hits and misses for a number of different HDR scenes. For the scenes measured, a hit ratio of at least 92% was achieved, leading to significantly reduced global memory traffic.

Scene	Resolution	Cache misses	Cache hits	Hit ratio
snow	1024x1024	462 344	5 807 480	92.6%
vine sunset	720x480	146 416	2 229 032	93.8%
Iwate	3720x1396	2 555 640	35 796 192	93.3%

Figure 6-9: Cache hit ratio. Measurements for a single GPU Texture Processing Unit (TPC) were made using the NVIDIA Visual Profiler [Vis10], then extrapolated for the entire kernel. For more information on the scenes used, see Appendix A

6.1.5.4 Loop unrolling

Examining the code in Listing 6.4 reveals the potential for further optimization. In particular, while the loop on Lines 39 to 47 is the most straightforward implementation of the dodging-and-burning kernel algorithm given in Section 5.4.5, it is not the most optimal.

A common optimization technique used for GPU programs, as well as traditional CPU programs, is a technique known as *loop unrolling*. Loop unrolling involves manually repeating all, or a portion of, the code in the body of a loop. This provides advantages such as elimination of unnecessary branches, which allows better optimization through *Instruction Level Parallelism* (ILP) [RF93]. This is particularly effective for simple `for`-loops that iterate for a constant number of iterations (loop unrolling for simple situations such as this is often performed by the compiler).

In the case of dodging-and-burning, however, unrolling the loop in Listing 6.4 for the sake of reducing condition evaluation would have little effect; the number of loops performed is directly influenced by the result of each loop. Nonetheless, there is a very good reason to apply this optimization. Lines 40 and 41 retrieve the next two scales for the box filter pyramid from the array `scales`, which was declared on Line 26. The elements read from

`scales` depend on the variable `s`, which is determined by the current loop index. Because these addresses are not known at compile time, CUDA allocates the array `scales` in *local memory* [NVI11, p. 92], which degrades performance. Indeed, compiling the naive kernel using the compiler flag `--ptxas-options=-v`, which instructs the compiler to print kernel memory allocation details during compilation, produces the following results:

```
ptxas info: Used 17 registers, 28+0 bytes lmem, 48+16 bytes smem,
            48 bytes cmem[1]
```

The output `28+0 bytes lmem` indicates that 28 bytes of local memory are being used the kernel. The terms `smem` and `cmem` represent *shared memory* and *constant memory*, respectively.

By unrolling the loop on Lines 39 to 47, all access offsets within `scales` can be specified at compile time. Listing 6.5 shows this optimization.

```
1  // same as naive kernel
2  ...
3
4  // loop in naive kernel unrolled as series of conditions
5  if (fabs(w) < epsilon){
6    V_center = V_surround;
7    V_surround = getAverageTex(y, x, scales[2]);
8    CENTER_SURROUND(scales[1], ...);
9  }
10 // one for each box filter...
11 ...
12
13 // final scale
14 if (fabs(w) < epsilon){
15   V_center = V_surround;
16   V_surround = getAverageTex(y, x, scales[7]);
17   CENTER_SURROUND(scales[6], ...);
18 }
19
20 // same a naive kernel
21 ...
```

Listing 6.5: Optimization by loop unrolling

It can be seen in Listing 6.5 that the array `scales` is no longer indirectly indexed; all access indices are now known at compile time. Compiling the kernel with the optimizations given in Listing 6.5 with the compiler option `--ptxas-options=-v` produces the following output:

```
ptxas info: Used 19 registers, 44+16 bytes smem, 60 bytes cmem[1]
```

The absence of the term `lmem` confirms that no local memory is used by this kernel. Figure 6-10 shows the execution times for a kernel with the box filter loop unrolled, using texture memory for SAT access.

Resolution	Kernel execution time (ms)
512x512	1.2
1024x1024	4.7
2742x2048	22.5

Figure 6-10: Average kernel execution times for a dodging-and-burning kernel that uses texture memory for SAT access and loop unrolling.

6.1.5.5 Using shared memory

Solving problems efficiently with shared memory
One of the major advantages of using CUDA, rather than shaders, for solving GPGPU problems is its access to shared memory, which can be accessed with a latency of approximately 2 orders of a magnitude less than that of global memory. A common approach to solving memory-bound problems is to load segments of the input data from global memory into shared memory, quickly solve partial problems within shared memory, and combine the partial solutions in global memory. In some cases, this process is repeated for a number of passes to produce the final result. The reduction method discussed in Section 6.1.2 is an example of such a process.

In the reduction kernel, the task of mapping elements from global memory to shared memory was simple: each thread in the CUDA processing grid was responsible for loading one element from global memory into shared memory. The threads within each block would then cooperate to reduce the segments stored in shared memory. The dodging-and-burning kernel, however, is not so simple.

As previously discussed, the main performance bottleneck is caused by accesses to the input SAT during the automatic dodging-and-burning procedure (Lines 26 to 47 in Listing 6.4). Because the dodging-and-burning procedure evaluates a box filter pyramid at each pixel, there is a high degree of locality in the memory accesses patterns of each thread. Therefore, if each block in the CUDA processing grid loads a sufficiently large segment of the input image into its shared memory upon invocation, all dodging-and-burning SAT accesses can be done within shared memory. Unlike the reduction kernel in Section 6.1.2, however, loading one element per thread into shared memory is insufficient to compute correct partial solutions in each block.

Apron requirements
Consider the situation depicted in Figure 6-11. The pale orange grid represents the shared memory allocated in one CUDA thread block of dimension bwxbh. The shared memory has been allocated such that each element corresponds to one element in the input image residing in global memory, and each element has been loaded from global memory by a dedicated thread. Each thread in the thread block is then responsible for computing the Local Region for one element in shared memory using the dodging-and-burning procedure described in Section 5.4.5.

The dodging-and-burning procedure involves evaluating a series of box filters of increasing scales surrounding each input pixel. The size of the maximum scale evaluated is denoted S_{max}. Figure 6-11 (a) shows the situation in which an element close to the center of the shared memory array (marked red) is being processed. The blue rectangle rep-

resents the largest possible scale in the box filter pyramid, S_{max}, centered at the target pixel. Because SAT-based box filters are computed by evaluating the corner points of the target region, valid data must be available in shared memory within a range of $S_{max}/2$ elements in all directions of the target pixel. Therefore, the dodging-and-burning for the pixel highlighted in Figure 6-11 (a) will be correct, because valid data can be found in shared memory for the largest possible box filter scale used. This is not true, however, for pixels closer to the edge of the shared memory array, such as the target pixel shown in Figure 6-11 (b). In this case, the correct computation of several layers of the box filter pyramid is dependent on data outside of the shared memory block.

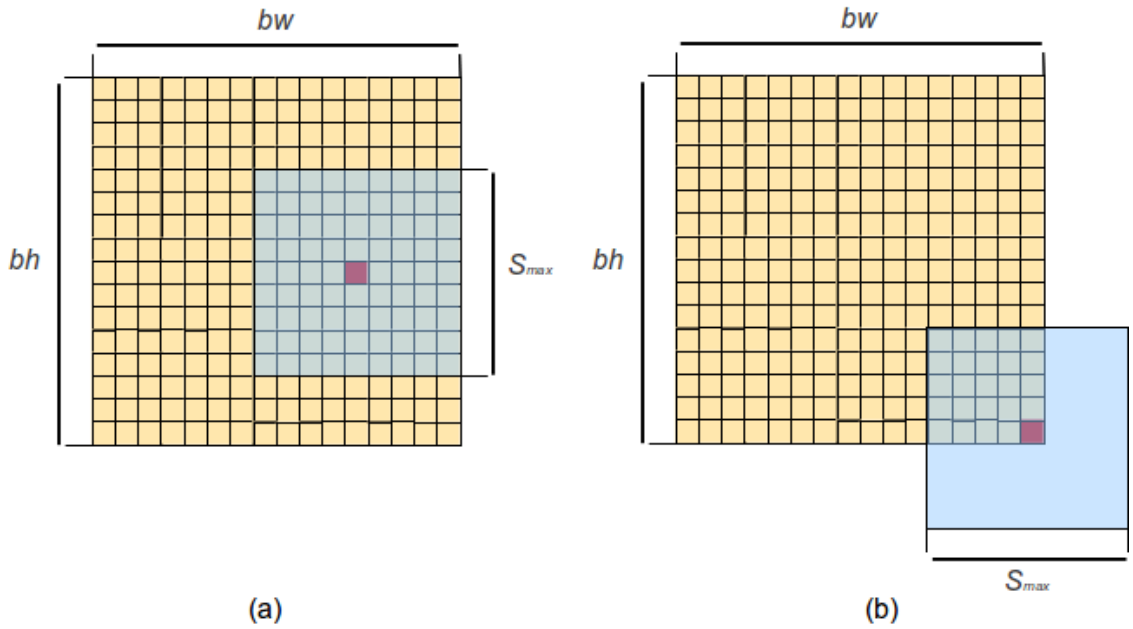


Figure 6-11: Dodging-and-burning in shared memory. In the situation depicted here, each thread in a CUDA thread block has loaded one input image element from global memory to shared memory (the pale orange grid). In order to identify the Local Region for any given target pixel (marked in red), valid data must exist in shared memory within the range indicated by the blue rectangle, which represents the largest possible scale in the box filter pyramid. While pixels near the center of the shared memory block can be processed correctly (a), pixels along the edges lack access to necessary data for correct box filtering (b).

In order to ensure correct results for elements at the edge of each block, each thread block must fill its shared memory with more elements than there are threads in the block. The additional elements loaded into shared memory for each block are denoted as that block's *apron*. Figure 6-12 shows how an apron is used to ensure correct results for threads at the edge of each block. The yellow elements represent entries in shared memory to which a thread is dedicated, and for which a Local Region will be identified. As in Figure 6-11, this region is of dimensions bwxbh, where bw and bh are the width and height of the CUDA thread block. The orange region surrounding these elements is in the block *apron* [Pod07]. For correct results, an apron with a minimum width of $S_{max}/2$ is required.

Avoiding thread divergence

It is highly inconvenient for a nxn thread block to load a chunk of mxm elements from

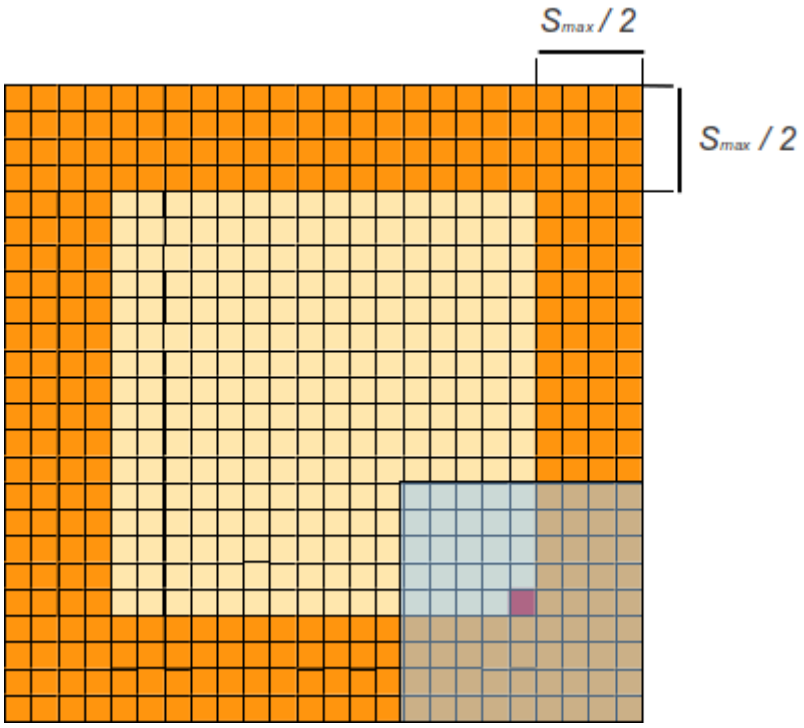


Figure 6-12: By loading an apron of additional data into shared memory (dark elements) dodging-and-burning of block output elements (light elements) can be accomplished successfully. For correct results, the apron must be a minimum of $S_{max}/2$ elements wide, where S_{max} is the width of the largest box filter evaluated.

global memory into its shared memory, when m is not a multiple of n. This is because each thread in the nxn thread grid must load a number of elements: the element that it will process, as well as a number of elements in the surrounding apron. CUDA threads use the Single Instruction Multiple Data (SIMD) computation model; for optimal results, each thread should perform exactly the same operation at the same time, with the only difference between threads being the data that they process. In the context of loading memory from global memory to shared memory using CUDA threads, this is achieved when the shared memory array dimensions are multiples of thread block dimensions.

Figure 6-13 shows the situation where the apron is the same width as the block itself, i.e n elements wide. The elements loaded by a single example thread are highlighted. By configuring each thread to read a 3x3 grid of elements centered at its location within the thread block, where each element is spaced at a distance of n from its horizontal and vertical neighbours, the entire shared memory array can be loaded with no thread divergence. For aprons with larger, multiple widths of n, the 3x3 thread read-grid can be expanded without issue. If, however, the apron width were not a multiple of the block width, thread divergence would occur. After loading most of the required elements using the 3x3 grid configuration shown in Figure 6-13, the majority of active threads would need to block while a small number of threads load the remaining elements of the outer, previously unreached apron fringe.

Closely examining the kernel code in Listing 6.4 reveals that the largest box filter actually computed during the dodging-and-burning procedure has an edge length of 25. There-

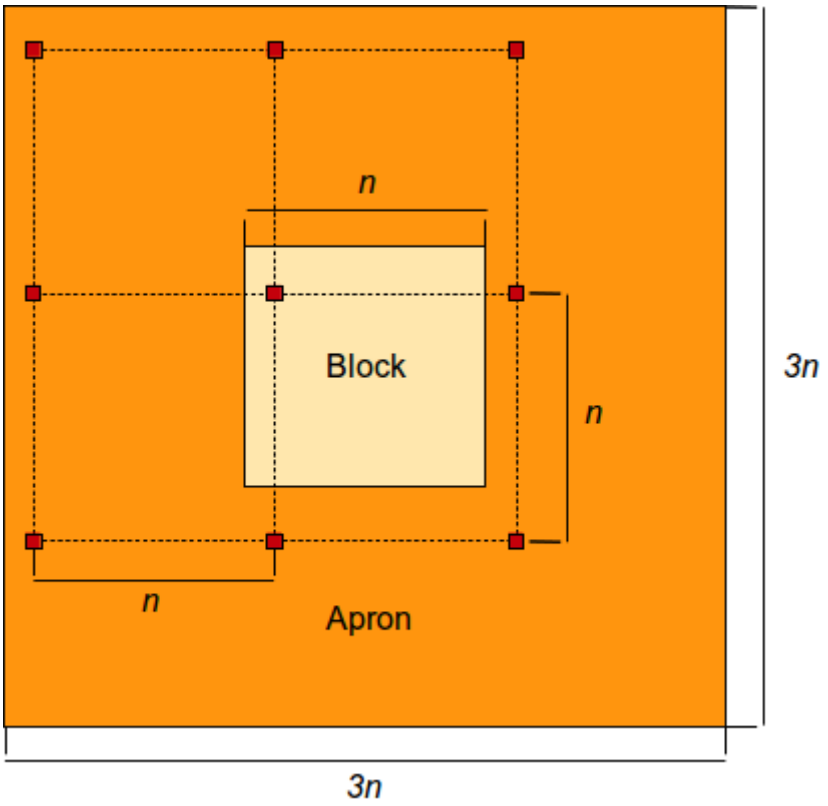


Figure 6-13: To avoid thread divergance, the apron width must be a multiple of the block width, n. In this case, shared memory is filled by assigning each thread to load a regular grid with element spacing n from global memory. An example 3x3 grid for a single thread is shown above.

fore, the minumum apron size required for each block is $\lfloor 25/2 \rfloor = 12$. It would seem, therefore, that the optimal kernel execution configuration would be with a block size of 12x12 threads, which allocate a 36x36 array of shared memory each. This, however, is not the case in practice. An additional property of CUDA kernel execution must be taken into account: that of multiprocessor *occupancy*.

Maximizing occupancy

Occupancy is defined as the ratio of the number of active warps per multiprocessor to the maximum possible number of warps per multiprocessor. Each multiprocessor on the GPU has access to a limited amount of registers and shared memory. Therefore, as the per-thread demand on these resources increases, the number of threads that can be scheduled per multiprocessor decreases.

There are a number of advantages to maximizing the number of threads per multiprocessor. As well as increasing the degree of parallelism, maximizing occupancy can help cover global memory access latency; when a thread blocks on a global memory request, it will be more likely that the blocked thread can be replaced by new thread from the ready queue. As a rule of thumb, it is desirable to attain an occupancy as close to 100% as possible.

The expected occupancy for a kernel can be computed using the *NVIDIA Occupancy Calculator*. The NVIDIA Occupancy Calculator is a spreadsheet that, given the per-thread

register and shared memory usage, computes the expected multiprocessor occupancy for a given GPU. For a given kernel, per-thread resource requirements are identified by passing the argument `--ptxas-options=-v` to the CUDA compiler, producing output demonstrated in Section 6.1.5.4 during kernel compilation.

Using the NVIDIA Occupancy Calculator, it was found that executing the dodging-and-burning kernel with a block size of 12x12 threads, where each block uses 36x36 elements of shared memory, resulted in an occupancy of 21% on the development system. Although increasing the block size led to an increased occupancy, it must also be considered that larger blocks result in increased global memory traffic due to the greater amount of data required for the aprons of each block. After some experimentation, an optimum was found for a block size 16x16 elements, where each block uses 42x42 elements of shared memory. This resulted in an occupancy of 33% on the test system.

It should be noted that the GPU used for this research, the NVIDIA GeForce 8800 GTX (see Appendix A.1), was one of the earliest GPUs with a unified architecture that supports CUDA. As such, this card is currently dated compared to the current state of the art in GPU hardware. For example, the same dodging-and-burning kernel that runs on our system with an occupancy of 33% would run on a modern GPU, such as any of the current NVIDIA Quadro cards, with an occupancy of 83%.

Global memory traffic
For maximal occupancy and minimal thread divergnace, a block size of 16x16 has been chosen, where each block allocates a total of 48x48 elements in shared memory. This means that for each 48x48 set of elements read from global memory, a total of 16x16 final results will be computed. Therefore, each block must read approximately 9 times more memory from the input texture than it ultimately outputs.

Because the texture cache is high speed on-chip memory, shared memory will only bring significant benefits if the total number of cache misses for a given frame can be reduced through its use. A simple experiment was conceived, which constructed a SAT from a 1024x1024 instance of the "snow" HDR scene (see Appendix A) and loaded each SAT element from texture memory to shared memory. Analysing the experimental kernel showed that this, best case texture memory access pattern resulted in a total of 32768 texture cache misses for the entire 1024x1024 input image. This number is so low partly due to the 2D spatial locality feature of the texture cache and partly because memory accesses were designed such that cache misses, which result in global memory fetches, are coalesced. Coalesced reads hide many cache miss events by servicing several horizontally neighbouring read requests at once. Loosely assuming that reading the aprons for each block will result in a similar texture cache hit/miss ratio as the 16x16 data within each apron, a rough figure of $294912 \approx 300000$ can be estimated as the expected number of texture cache misses when using shared memory for dodging-and-burning. Comparing this value to the texture cache misses reported in Figure 6-9, it can be concluded that performance gains may not be as drastic as previously expected. Nonetheless, to test the actual performance attainable by using shared memory for doding-and-burning, a kernel that uses shared memory has been implemented.

A shared memory based dodging-and-burning kernel
Listing 6.6 shows the important parts of an implementation of the dodging-and-burning kernel using shared memory.

```
1  __global__ void dodgeAndBurn_SharedMemory(
2      // same arguments as naive kernel
3  )
4  {
5    // shared memory cache for this block
6    // initialised with 48+1 rows to reduce bank conflicts.
7    __shared__ float cache[48][48+1];
8
9    // get block position in grid
10   int bx = blockDim.x*blockIdx.x;
11   int by = blockDim.y*blockIdx.y;
12
13   // identify in and output coordinates
14   int satx = by + threadIdx.x;
15   int saty = bx + threadIdx.y;
16   int outx = bx + threadIdx.x;
17   int outy = by + threadIdx.y;
18
19   // to reduce indexing clutter
20   int tx = threadIdx.x;
21   int ty = threadIdx.y;
22   int bw = blockDim.x;
23   int bh = blockDim.y;
24
25   // fill cache
26   CACHE(ty+bw, tx+bw) = SATTEX (satx, saty);
27   CACHE(ty+bw, tx) = SATTEX(satx-bw, saty);
28   CACHE(ty+bw, tx+2*bh) = SATTEX(satx+bw, saty);
29
30   CACHE(ty, tx) = SATTEX( satx-bw, saty-bw );
31   CACHE(ty, tx+bh) = SATTEX( satx, saty-bw );
32   CACHE(ty, tx+2*bh) = SATTEX( satx+bw, saty-bw );
33
34   CACHE(ty+2*bw, tx) = SATTEX(satx-bw, saty+bw);
35   CACHE(ty+2*bw, tx+bh) = SATTEX(satx, saty+bw);
36   CACHE(ty+2*bw, tx+2*bh) = SATTEX(satx+bw, saty+bw);
37
38   // barrier: all threads must reach here before we continue
39   // ensures cache is correctly loaded
40   __syncthreads();
41
42   // proceed as in other kernels, this time accessing CACHE
43   // rather than SATTEX for box filter computation
44   ...
45 }
```

Listing 6.6: Shared memory based dodging-and-burning kernel.

The first important line in Listing 6.6 is Line 7, which specifies the shared memory array used by the current block. As previously discussed, each block requires 48x48 shared memory elements. Because 48 is a multiple of 16, which is the shared memory bank

width (Section 2.3.3), accessing N columns simulatenously in the same warp will result in an N-way bank conflict. This can easily be resolved by padding each row with a single element.

Lines 10 and 11 identify the position of the current block within the CUDA processing grid. The current thread is then mapped, in Lines 14 to 17, to an input SAT element at position `(satx, saty)` and to a pixel in the output PBO located at `(outx, outy)`.

The appropriate SAT elements required for dodging and burning within the current block are then loaded into shared memory on Lines 26 to 36. Indexing the shared memory array `cache` is simplified by the macro `CACHE()`, whose definition is given in Listing 6.7.

```
1 #define CACHE(x,y) (cache[y][x])
```

Listing 6.7: A macro for simplifying access to shared memory.

Finally, the call to the CUDA function `__syncthreads()` on Line 40 creates a barrier at this point in the program; no thread in the block can proceed beyond this point until all threads in the block have called `__syncthreads`. This ensures that the block's shared memory is correctly filled before proceeding with dodging-and-burning.

The remainder of the kernel is similar to the texture memory kernel with unrolled loops, except that SAT elements are read from shared memory, rather than texture memory.

Unlike in previous kernel implementations, the input and output coordinates in Listing 6.6 are different. This is because the kernel is invoked such that the dimensions of the CUDA processing grid correspond to those of the input and output images, and each thread in the processing grid can easily be mapped to a pixel in the input and output textures. The input SAT is transposed, and thus the manner by which each thread is mapped to an element in the input SAT must differ from the way it is mapped to its input and output textures. Reading transposed SAT elements in the naive (Listing 6.4) and texture memory (Listing 6.5) kernels was straightforward: the x and y coordinates of global or texture memory accesses requests were simply transposed. When using shared memory, each thread must have to access, within its shared memory, to the SAT entry corresponding to the input HDR pixel it is processing. In order to achieve this, the SAT is transposed as it is read into each block's shared memory. Figure 6-14 illustrates this process.

Within each identified block in the SAT, elements are read on a row-by-row basis. This maximizes memory access efficiency; in the event of a texture cache miss, the ensuing global memory reads will always be coalesced. Each row read from a block in the SAT is written as a column in shared memory (horizontal red lines in Figure 6-14). Because shared memory does not have the same coalescing requirements, writing columns, rather than rows, will incur no performance penalty.

Shared memory performance

The average execution times measured for dodging-and-burning a test scene at a number of resolutions are listed in Figure 6-15. As can be seen in the Figure, the shared memory kernel implementation delivers the best performance of all optimizations presented here.

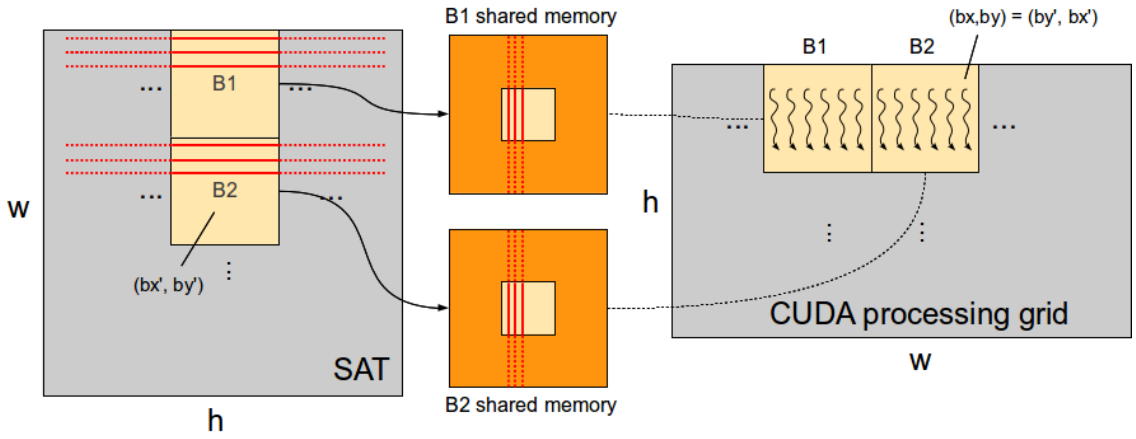


Figure 6-14: The CUDA processing grid (right) is invoked with the same layout as the input HDR texture, where each thread in the grid corresponds to one input texel. The processing grid is divided into blocks. Each block loads a block of memory from the scaled luminance SAT (left) into its shared memory (center). Each row loaded from the SAT (red, horizontal lines) is stored as a column in shared memory. For each block, sufficiently many rows of sufficient width are loaded to fill the apron. The portion of each loaded SAT row belonging to the apron is marked by a dashed red line. This procedure, together with an appropriate block coordinate mapping ($(bx, by) = (by', bx')$) results in an entirely transposed copy of the input SAT (plus aprons) residing in the collective shared memory of the processing grid.

Resolution	Kernel execution time (ms)
512x512	0.9
1024x1024	3.9
2742x2048	19.7

Figure 6-15: Average kernel execution times for dodging-and-burning with shared memory.

6.1.5.6 Summary

The dodging-and-burning kernel was the most complex kernel written for the CUDA tone mapping module. Although the algorithm performed by this kernel (given in Section 5.4.5) is relatively simple, additional considerations were necessary for its efficient implementation in CUDA. A number of implementations were presented, using global, texture, and shared memory for SAT access. Each optimization introduced an additonal amount of complexity to the dodging-and-burning kernel, in exchange for improved performance. Kernels using texture and shared memory, with unrolled loops, both gave good results. Figure 6-16 compares the execution times of each of the dodging-and-burning kernel implementations presented.

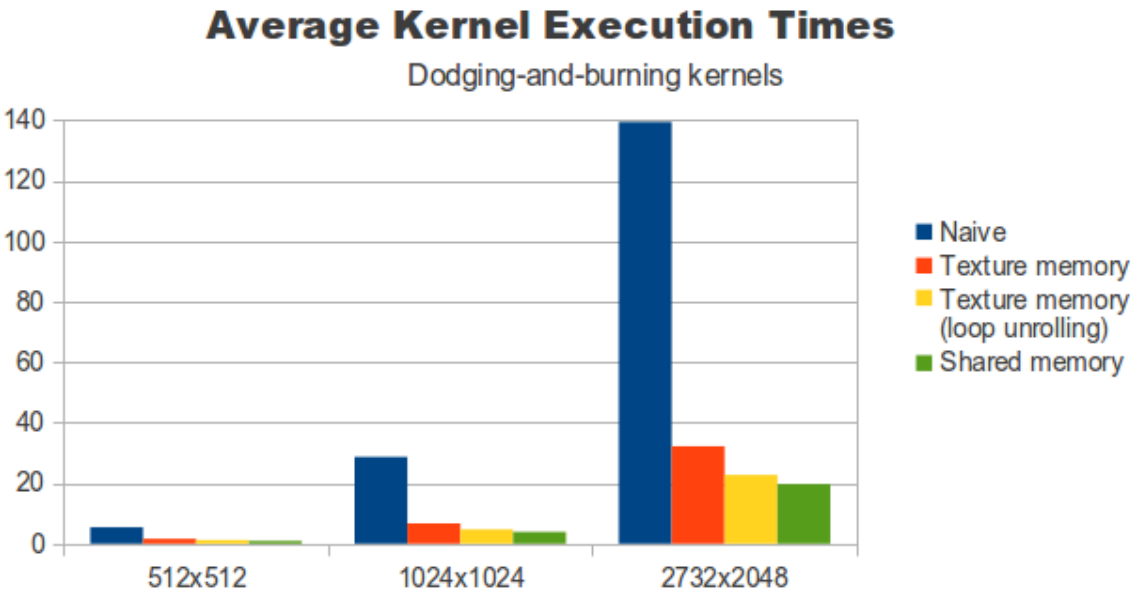


Figure 6-16: A comparison of the average execution times of the dodging-and-burning kernels discussed in Section 6.1.5.

6.2 Tone mapping with shaders

The shader tone mapping module, introduced in Chapter 5, was implemented as a collection of OpenGL GLSL shader programs, together with C++ code for organizing and invoking them. This section describes the shader programs used for tone mapping, which were introduced in Section 5.5.

6.2.1 A bypass vertex shader

Most of the tone mapping work in the module, with the exception of SAT generation described in Section 5.4.4, is done performed solely by fragment shaders. Therefore, unless otherwise specified, the shader programs presented in the succeeding subsections are fragment programs. These programs receive their interpolated position and texture coordinates from a simple, *bypass* vertex shader, given in Listing 6.8.

```
1 void main()
2 {
3         gl_Position = ftransform();
4         gl_TexCoord[0] = gl_MultiTexCoord0;
5 }
```

Listing 6.8: GLSL bypass vertex shader used to pass interpolated position and texture coordinate data to the fragment shaders presented throughout the remainder of this section.

The vertex shader shown in Listing 6.8 consists of two important lines. Line 3 computes the position of each input vertex in screen coordinates using the built-in GLSL function `ftransform()`, which simply transforms the input vertex's position from world coordinates to screen coordinates. The result is written to the 4-vector `gl_Position`, which is a

parameter that is passed to the fragment shader. Similarly, vertex texture coordinates are retrieved from the predefined GLSL vector `gl_MultiTexCoord0` on Line 4, and passed to the fragment shader via `gl_TexCoord[0]`.

6.2.2 Luminance extraction

The luminance extraction fragment shader is one of the simplest in the tone mapping module. Its complete source is listed in Listing 6.9.

```
1  // hdr input texture
2  uniform sampler2D hdrTex;
3
4  void main(void)
5  {
6    // get color from the texture
7    vec4 currTexel = texture2D(hdrTex, gl_TexCoord[0].st);
8    float lum = dot(vec3(0.2125,0.7154,0.0721), currTexel.rgb);
9
10   // save luminance and log-luminance
11   // in output channels
12   gl_FragColor.r = log(lum + 1.0f);
13   gl_FragColor.y = lum;
14 }
```

Listing 6.9: GLSL fragment shader for computing luminance.

Line 2 declares a handle to the only shader input, the input HDR texture `hdrTex`, which is passed to the shader from the C++ code from which it was invoked. The red, green, blue and alpha channels of the current texel are then sampled on Line 7 using the GLSL texture sampling function `texture2D`, storing the results in the 4 element vector `currTexel`. The first 3 components of this vector are then used on Line 8, which computes the luminance of the current input texel by appying equation 2.1.

Once the current pixel luminance has been computed, Lines 12 to 13 save the results in the output fragment color vector, `gl_FragColor`.

Unlike the luminance extraction CUDA kernel presented in Section 6.1.1, which writes its results to a single-channel array of float in global GPU memory, the fragment shader in Listing 6.9 writes its results to 4-channel HDR output texture. Therefore, the logarithm of the computed luminance, which is required for scene key computation, is saved into the red channel of the output texture, while the original luminance is saved into the green output channel. This avoids, at the expense of a larger memory footprint, the necessity of evaluating an exponential function each time luminance is required in subsequent shaders, as was the case in the CUDA module. Note that the value 1.0 on Line 12, which corresponds to δ in Equation 4.1, is added to ensure that the result of the log-function is defined when `lum` is 0.

6.2.3 Scaled luminance computation

The important fragment shader code for computing scaled luminance is shown in Listing 6.10. For brevity, the `main()` function and input variable declarations are left out.

```
1  float l = texture2D(lumTex, gl_TexCoord[0].st).g;
2  float lMean = tex2Dlod(lumTex, float4 (0.5, 0.5, 0.0, 100)).r;
3
4  float ls = (alpha / exp(lMean)) * l;
5  gl_FragColor.rg = vec2(ls, l);
```

Listing 6.10: GLSL fragment shader for computing scaled luminance.

The first two lines in Listing 6.10 sample the input texture, `lumTex`, which contains the results of the luminance extraction fragment shader discussed in Section 6.2.2. The luminance of the current texel is sampled on Line 1 using the same sampling procedure as described in Section 6.2.2 and stores the result in the local variable `l`. Line 2, on the other hand, is more interesting.

As noted in Section 5.5, the shader tone mapping module contains no reduction operation. Instead, a mipmap is generated from the output of the luminance extraction shader prior to invoking the scaled luminance shader. This process is illustrated in Listing 6.11, which shows how the module C++ code calls the OpenGL procedure `glGenerateMipmapEXT` to generate a mipmap between shader calls.

```
1  runLuminanceShader (inHDRTex, lumTex);
2
3  // generate a mipmap from lumTex via OpenGL
4  glBindTexture(GL_TEXTURE_2D, lumTex)
5  glGenerateMipmapEXT(GL_TEXTURE_2D).
6  glBindTexture(GL_TEXTURE_2D, 0)
7
8  runScaledLuminanceShader(lumTex, scaleLumTex)
```

Listing 6.11: Generating a mipmap between shader calls.

Knowing that the input texture `lumTex` has previously been mipmapped, Line 2 in Listing 6.10 uses the GLSL function `tex2Dlod` to sample the center of the top element of `lumTex`'s mipmap, storing the red channel in the variable `lMean`. Because the red channel of each texel in `lumTex` contains the logarithm of its luminance and the top, 1x1 level of a mipmap is a single texel containing the average all texels in the bottom, wxh level, `lMean` will contain:

$$\frac{1}{N}\sum_{x,y} log(\delta + L_w(x,y))$$

where $L_w(x,y)$ is the luminance of texel (x,y) in the input HDR texture, and N is its total number of texels (see Equation 2.1).

The scene key can be retrieved from `lMean` by applying the exponential function. Line 4 does this, using the scene key to compute the scaled lumininance for the current texel

by computing Equation 4.2 and saving the result to `ls`. The parameter `alpha`, which corresponds to α in Equation 4.2, is given as a shader input parameter.

Finally, Line 5 makes use of the multiple output channels by saving the computed scaled luminance to the red output channel, and copying the input luminance to the output green channel.

6.2.4 SAT generation

The SAT generation submodule constructs SATs using *Sengupta et al.*'s [SLO06] work-efficient parallel scan algorithm, presented in detail in Section 5.2.4. The submodule is implemented using the OpenGL GLSL shader language and C++. Like its equivalent submodule in the CUDA tone mapping module, all SAT generation functionality, including all necessary shader programs and associated C++ code for managing them, is interfaced through a C++ class. The most important parts of this class, `SATSengupta`, are depicted using UML in Figure 6-17. Note the use of the namespace `libGL`, indicating that its implementation uses the OpenGL GLSL shading language. This allows for easy integration of additional implementations using other technologies, such as CUDA or the NVIDIA CG shading language.

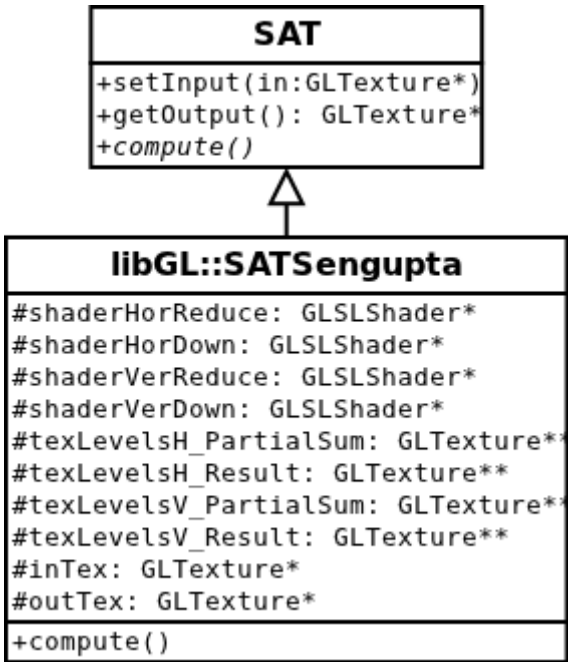


Figure 6-17: The shader SAT generation submodule interface.

A number of protected attributes are visible in Figure 6-17. Indeed, to implement Sengupta's parallel scan algorithm, two GLSL shader programs are required: one for the reduction phase and one for the down-sweep phase. As discussed in Section 5.2.4, each phase requires an additional set of textures to store intermediate results. Therefore, to compute scans for each row in an input texture, two shader programs (`_shaderHorReduce` and `_shaderHorDown`) and two arrays of intermediate textures (`_texLevelsH_PartialSum` and `_texLevelsH_Result`) are required. In order to generate a SAT, column scans must be computed from the results of the row scans, requiring two additional shader objects (`_shaderVerReduce` and `_shaderVerDown`), as

well as two additional arrays of intermediate textures (`_texLevelsV_PartialSum` and `_texLevelsV_Result`).

The manner with which the class `SATSengupta` uses its shader programs and intermediate texture objects to generate a SAT from an input texture is illustrated in Figure 6-18, for a simple input 4x4 input texture with all texels set to 1.

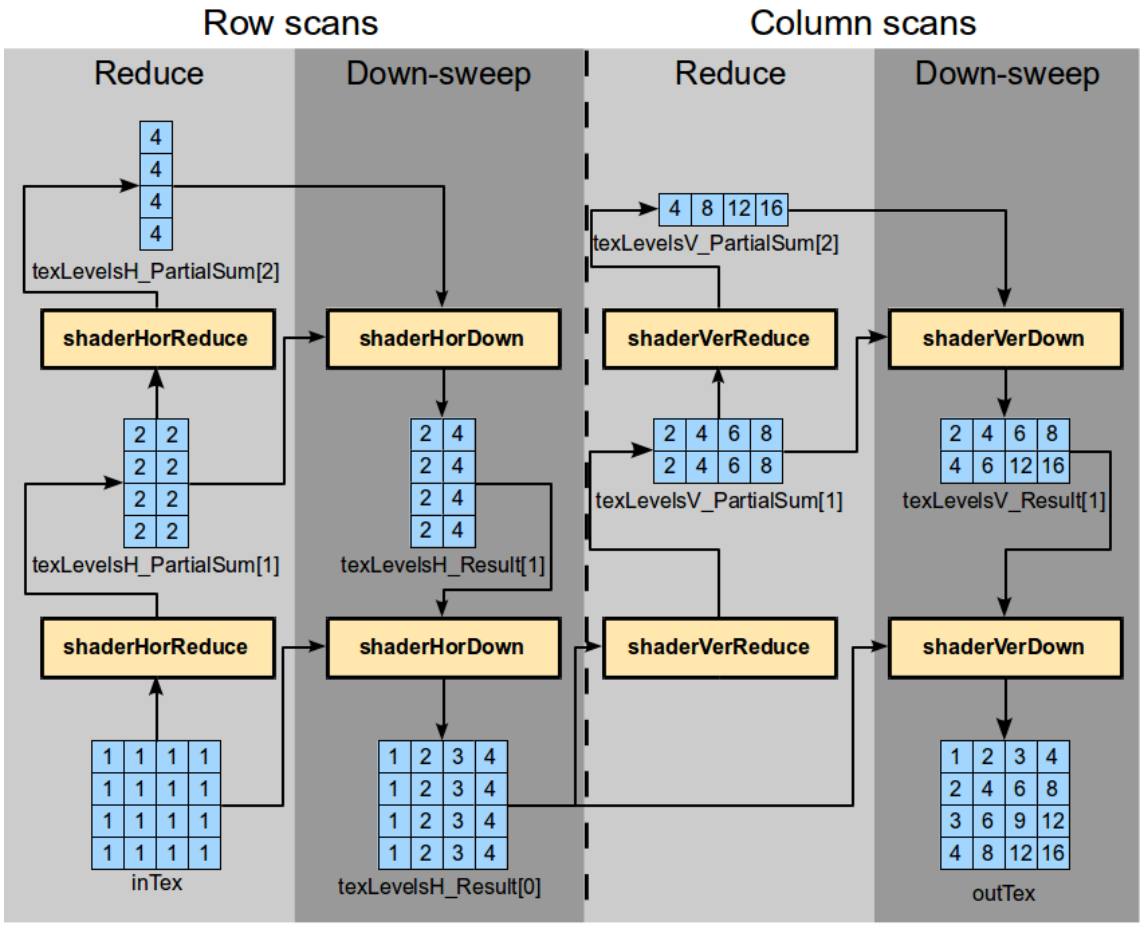


Figure 6-18: Generating a SAT from an example 4x4 input texture.

The diagram in Figure 6-18 is vertically partitioned into two parts: the left half computes scans for each row of the input texture and the right half computes column scans on the result. Within each partition, computation used in the reduce phase of the algorithm is shown with a light grey background, while down-sweep computations are marked with a dark grey background. Each shader pass is represented by a yellow rectangle, labelled to indicate which shader program in `SATSengupta` is being executed. Similarly, the contents of `SATSengupta`'s intermediate textures is shown between each shader invocation.

In Figure 6-18, a total of 8 passes are required to correctly compute the SAT for 4x4 input texture. In general, for a wxh input texture, where w and h are the texture's width and height, respectively, each phase in the left partition requires $log_2(w)$ passes to complete, and each phase in the right partition requires $log_2(w)$ passes. Therefore, in general, a total of $2\,(log_2(w) + log_2(h))$ passes are required to generate a correct SAT from a wxh input texture.

It is clear from Figures 6-17 and 6-18 that four seperate shaders are used for SAT gener-

ation. This section discusses the two shaders used for computing row scans (on the left in Figure 6-18); the other two are simply small modifications of these, which work with columns rather than rows.

6.2.4.1 Reduction

The reduction phase used for row scan computation uses both vertex and fragment shaders. The vertex shader computes the texture coordinates required for pairwise texel addition and the fragment shader samples its input texture at these coordinates, interpolated for each fragment position, and applies pairwise texel addition. During this process, it is important to account for the fact that, for each pass, the output texture is half the width of the input texture.

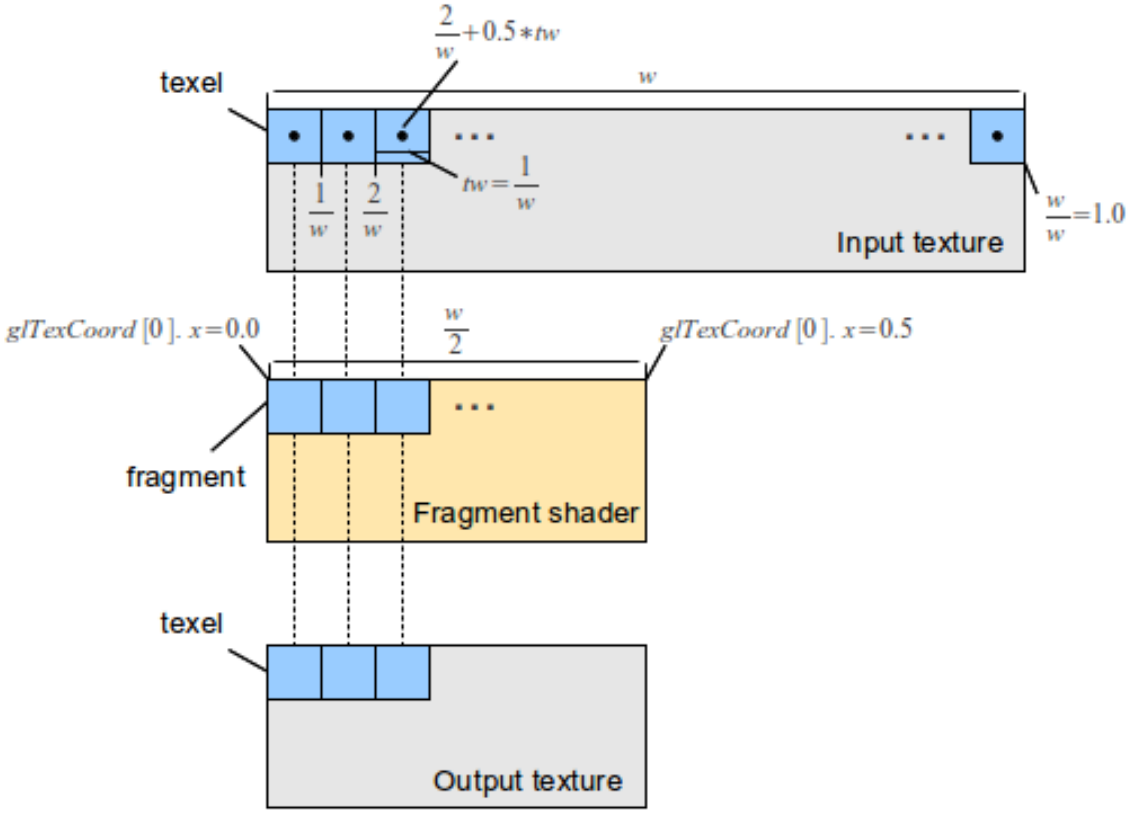


Figure 6-19: A single pass of the reduction algorithm. The output texture is half the width of the input texture. In this situation, the fragment shader is invoked with the same dimensions as the output texture. Since the texture coordinates for each fragment (gl_TexCoord[0]) are interpolated from the screen aligned quad used to invoke the shaders, which has the same dimensions as the input texture, the x texture coordinates in the fragment shader will only extend to 0.5. The position and width of each texel in the input texture is given in normalized coordinates. Correspondences between fragments and input texels are marked with dashed lines.

Each pass i invokes its reduction shaders by rendering a screen aligned wxh quad, where w and h are the dimensions of the input partial result `_texLevelsH_PartialSum[i-1]`. Rather than rendering to screen, the output of the fragment shader is written to the next

partial result texture, `_texLevelsH_PartialSum[i]`. This situation is depicted in Figure 6-19, where the input texture corresponds to `_texLevelsH_PartialSum[i-1]` and the output texture `_texLevelsH_PartialSum[i]`.

Because the output texture in Figure 6-19 is half the width of the input texture, only fragments with texture coordinates $x \leq 0.5$ are processed by the fragment shader. Therefore, if each fragment samples the input texture at it's interpolated texture coordinate, `glTexCoord[0].xy`, which represents its position within the screen aligned quad normalized against the quad's vertex texture coordinates, only the first half of the input texture will be sampled.

The problem shown in Figure 6-19 can be accounted for in the vertex shader. Listing 6.12 shows how this is done.

```
1  vec2 uv = gl_MultiTexCoord0.xy;
2  float tw = 1.0/imgw;      // texel width
3
4  gl_TexCoord[0].xy = vec2(2*uv.x-0.5*tw, uv.y);
5  gl_TexCoord[0].wz = vec2(2*uv.x+0.5*tw, uv.y);
```

Listing 6.12: Reduction phase vertex shader code for computing row scans.

Line 1 in Listing 6.12 copies the current vertex coordinates into a local vector, `uv`, to make the remaining code more readable. The width of each input texel is then computed on Line 2. Because GLSL shaders by default work with normalized coordinates, the texel width is attained by dividing 1 - the width of a texel in absolute coordinates - by the width of the input texture, `imgw`, which is passed to the vertex shader as an input parameter.

The solution to the problem in Figure 6-19 is found on Lines 4 and 5. By doubling the texture x coordinates at the canvas vertices in the vertex shader, as done Line 4, the interpolated texture coordinate read by each fragment in the fragment shader will extend to 1.0. However, this also has the effect of doubling the texture coordinates at each fragment. This situation is shown in Figure 6-20. It can be seen in Figure 6-20 that sampling each texel in the input texture at `gl_TexCoord[0].xy` for each fragment in the fragment shader reads only half of the total input texels. Therefore, Line 5 saves, into the `z` and `w` components of 4 element output vector `gl_TexCoord[0]`, the coordinates of a theoretical vertex at a distance of one texel to the right of the current vertex. When interpolated in the fragment shader, the `gl_TexCoord[0].zw` vector addresses each texel that is missed by `gl_TexCoord[0].xy`.

Once the correct texture coordinates coordinates have been computed in the vetex shader, the fragment shader is left with the straightforward task of sampling its input texture at the texture coordinates assigned to each fragment, and performing pairwise additions on these samples. This operation is shown in Listing 6.13.

```
1  vec4 uv = gl_TexCoord[0];
2  float curr = texture2D(tSrc, uv.xy).r;
3  float next = texture2D(tSrc, uv.wz).r;
4
5  gl_FragColor.r = curr+next;
```

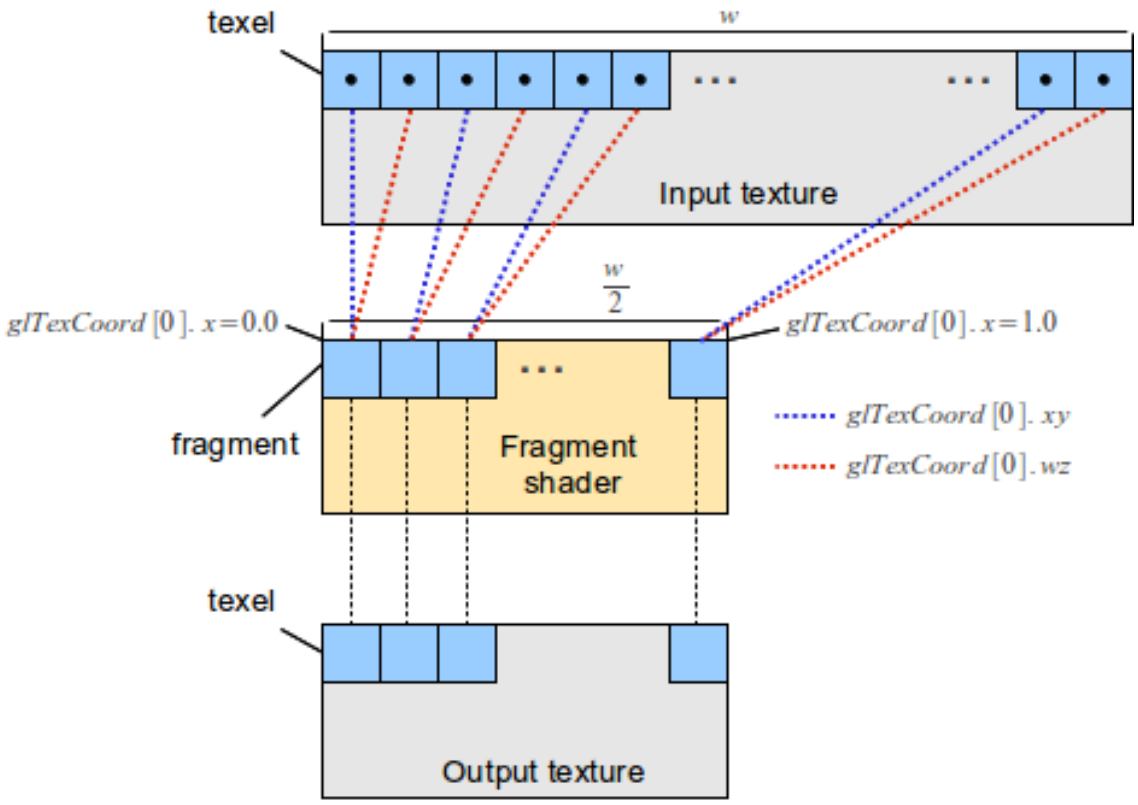


Figure 6-20: By adjusting texture coordinates in the vertex shader, each fragment can sample the correct input texels. Each even texel is addressed with the vector gl_TexCoord[0].xy, and each odd one with gl_TexCoord[0].wz.

Listing 6.13: Reduction phase fragment shader code for computing row scans. The symbol tSrc is a handle to the input texture.

Note that all results in Listing 6.13 are read from and written to the red channel of the in- and ouput textures. This is because only one channel is required to generate and represent a scaled luminance SAT. In fact, to reduce unnecessary memory traffic incurred from sampling four channel textures for each texel read in SAT generation, all textures in the `SATSengupta` class have been defined as single-channel, 32-bit floating point `GL_LUMINANCE` textures. Using this texture format for SAT generation resulted in significant performance gains over implementations using traditional, `RGBA` format textures, such as that of *Linnemann et al.* [LWR+09] (see Chapter 7).

6.2.4.2 Down-sweep

The down-sweep phase iterates through the partial sum `_texLevelsH_PartialSum` textures from smallest to largest, generating partial results, stored in the texture array `_texLevelsH_Result`, during each iteration. The width relationship between the input and output textures for each shader pass is exactly of the opposite of that of the reduction phase: during each pass, the down-sweep fragment shader writes to a texture twice the width of its input. When working with normalized texture coordinates, this leads

to significantly simplified fragment-to-texel mapping compared to that of the reduce phase.

When a shader's input texture is narrower than its output, texture coordinates in the fragment shader are not clipped. Therefore, each fragment can correctly address its corresponding texel in the input texture. Furthermore, because normalized coordinates are relative to the texture dimensions, the mapping of texels from input to output by means of coordinate (or index) doubling required by the down-sweep algorithm, shown in Algorithm 6 in Section 5.2.4, is attained implicitly through the use of normalized coordinates. Figure 6-21 shows how the x-coordinates map from a narrow texture to a wider one when using normalized coordinates.

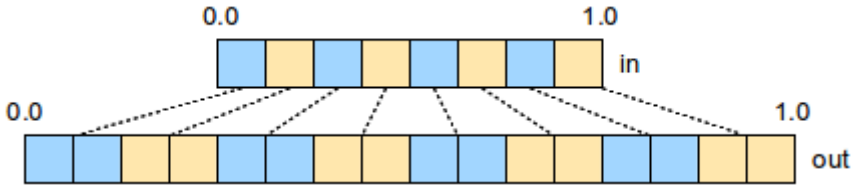


Figure 6-21: Mapping of elements between textures of different sizes when using normalized coordinates. The input texture (top) is half the width of the output (bottom), both textures have a height of 1. Each texel position in the input texture maps to the border between to output texels. Furthermore, the width of a single texel in the input corresponds to the width of two texels in the output. This mapping is ideal for the down-sweep alogrithm.

Because texture coordinates require no adjustment, the down-sweep uses the simple bypass vertex shader described in Section 6.2.1 for passing texture coordinate and position data to the fragment shader. The fragment shader then computes a down-sweep pass by applying the down-sweep algorithm.

The implementation of the down-sweep algorithm is a trivial translation of Algorithm 6 in Section 5.2.4 into GLSL shader code. For brevity, this code is emitted here. To see the complete down-sweep fragment shader, refer to Appendix B.

6.2.5 Dodging-and-burning

Unlike with CUDA, when working with shaders, it is often the case that the most straightforward implementation of an efficient algorithm is itself highly efficient, requiring no further optimization. Fortunately, this applies to the implementation the dodging-and-burning kernel. Therefore, a direct GLSL implementation of the general dodging-and-burning algorithm, given in Section 5.4.5, gives the desired results. The only code optimization used in the shader implementation that slightly deviates from the algorithm specified in Section 5.4.5 is that of loop unrolling, which is described in Section 6.1.5.4. Therefore, the code used for the dodging-and-burning shader is not presented here. To see the full dodging-and-burning shader code, see Appendix B.

6.3 TMStudio: An OpenGL test platform

To support the research in this thesis, a test platform was built. The test platform, called *TMStudio*, is a cross-platform, GUI-based OpenGL application written in C++ and Qt, which is designed to allow easy development, debugging and evaluation of tone mapping operators. All tone mapping modules developed in this thesis were created and tested using TMStudio.

Section 6.3.1 gives an overview of the functionality of TMStudio. The internal design and implementational details of TMStudio are described in Section 6.3.2.1. Finally, Section 6.3.3 concludes by outlining possible future work.

6.3.1 Using TMStudio

TMStudio provides the basic functionality required to test the speed and visual output of any tone mapping operator. To test a new operator, a developer must first add their implementation to the class structure described in Section 6.3.2.1 and recompile TMStudio. Once running, the first step is to load any HDR image in OpenEXR format from file. The selected OpenEXR file is loaded and displayed on a screen aligned quad within an OpenGL render widget of the same dimensions as the loaded image. The developer can then select their implementation from the list of all of available tone mapping operators in the dockable "tone mapping operator" widget. Using the same widget, the developer can directly modify tone mapping parameters and view the effects on the HDR scene in real time.

To evaluate tone mapping performance, the frame rate of the scene rendering, which includes the time required for tone mapping, is shown on the bottom left of the TMStudio main window frame. For performance measurements that are less affected by instantaneous speed fluxuations, the average frame rate recorded during the first 30 seconds of execution is shown, when ready, in the "information" widget.

For further, offline evaluation of the visual output of any operator, the LDR scene displayed in the render widget can be saved to file in a number of standard image formats, such as PNG, JPEG or BMP.

Figure 6-22 shows an example of TMStudio in action.

6.3.1.1 Widgets

Interaction with the currently selected tone mapping operator is achieved via a collection of dockable widgets in the TMStudio environment. This subsections gives a brief description of each widget currently implemented.

Tone Mapping Operator
This widget is used to select the current tone mapping operator and interact with it. For the currently selected operator, the relevant parameters can be updated via sliders and input boxes. The effects on the scene are updated in real time.

The *Tone Mapping Operator* widget can be seen on the top left in Figure 6-22. In the

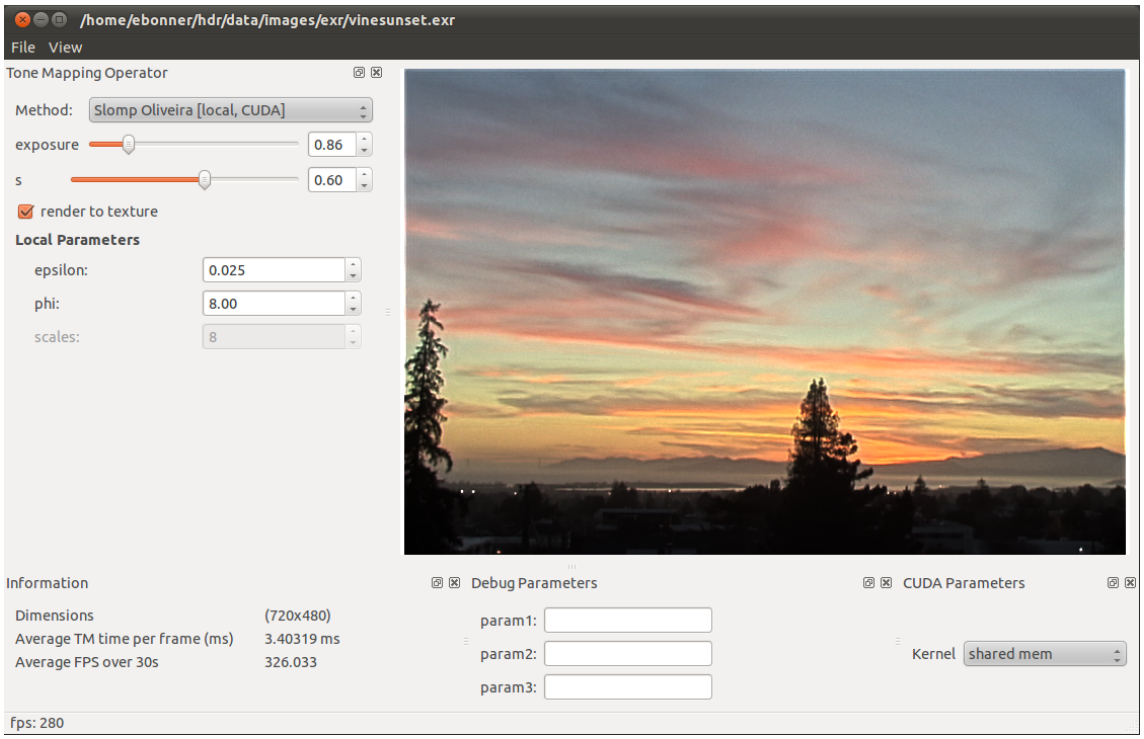


Figure 6-22: TMStudio in action. The user has chosen the "vinesunset" HDR scene for tone mapping (see Appendix A) and is selecting an appropriate tone mapping operator from the list of available operators in the "tone mapping operator" widget on the left. The current frame rate is shown on the bottom left of the window, and the average frame rate for the first 30 seconds of tone mapping with the current operator is shown in the "information" widget.

figure, a CUDA implementation of *Slomp and Oliveira*'s method has been selected. The only parameter that doesn't correspond directly to one of the four parameters required by *Slomp and Oliveira*'s operator is *render to texture*. When this flag is set, the output of the tone mapping operator is rendered to an offscreen texture, before it is displayed in the render widget. This is used to for accurate frame rate measurements of scenes larger than the screen resolution; otherwise, when using shader implementations, invisible fragments will be clipped before reaching the fragment shader, leading to incorrect performance measurements.

Information

The *Information* panel displays information on the current simulation. At present, the resolution of the input HDR scene, the time (in ms) required to tone map each frame and the average frame rate over the first 30 seconds of execution are shown.

Debug Parameters

Often, when debugging an operator, it is useful to pass it some temporary, debugging parameters not intended to be used in the final version. Updating the GUI to support additional temporary parameters, which will be removed again later, can be a tedious process. Therefore, the *Debug Parameters* widget allows the user to send up to three debug parameters directly to the currently executing operator.

CUDA Parameters

A number of implementations of the CUDA tone mapping module described in Section

6.1 were proposed. Each of these implementations can be tested by selecting the appropriate kernel in the *CUDA Parameters* widget.

6.3.2 Implementational details

6.3.2.1 A class structure for OpenGL tone mapping

Figure 6-23 shows a simplified version of the class structure used by TMStudio for testing tone mapping operators. The module has been designed to support more tone mapping approaches than the methods examined in this thesis, implemented using any number of technologies. This is particularly useful for debugging; a software implementation of any given method can easily be added for verifying its visual quality (both qualitativley and quantatively), before optimizing performance using GPU techniques. Furthermore, the performance of different implementations of any particular method can be directly compared - a procedure frequently undertaken during this research. Because the code for implementing a specific method is generally well encapsulated with one or two classes, it is easy to extract an operator, once tested and optimized, from TMStudio and adapt it to any desired target application.

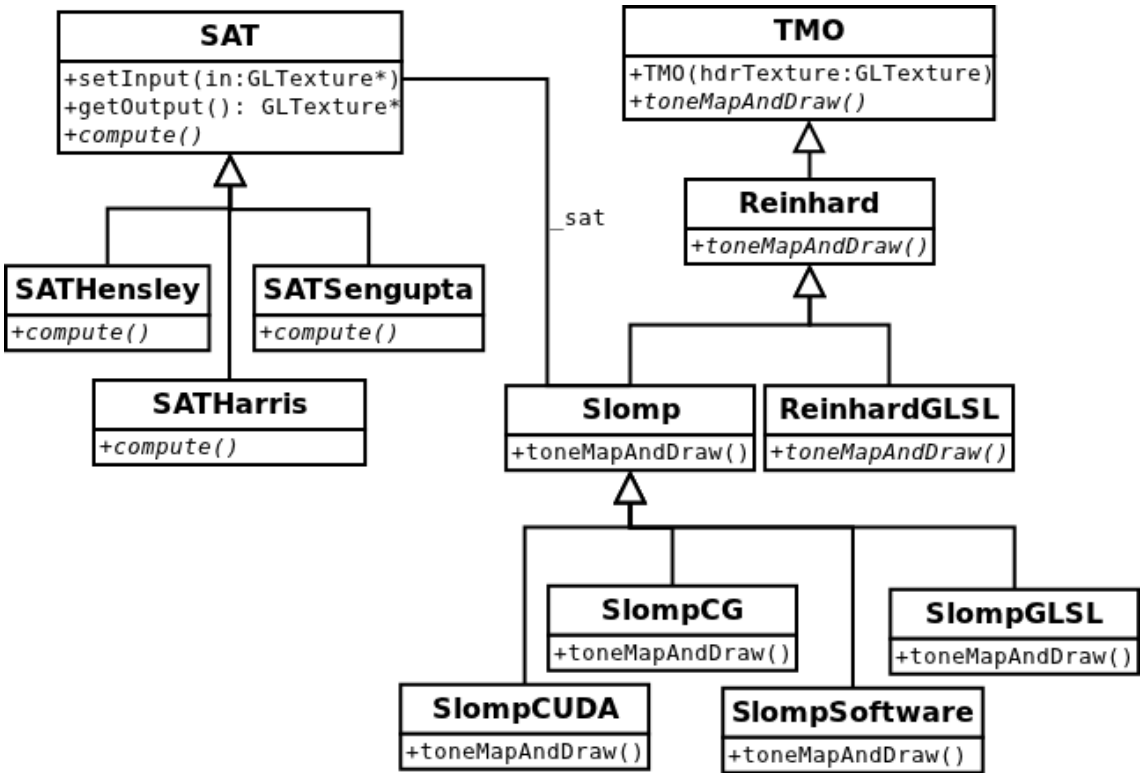


Figure 6-23: A class structure for developing and testing tone mapping operators.

It can be seen in Figure 6-23 that all tone mapping methods are children of the abstract class `TMO`, short for "Tone Mapping Operator". Any new tone mapping operator added to the module must derive this class. Because the details of each implementation of a given method can vary greatly, each method added to the class structure should be abstract. Concrete implementations of a method inherit from its definition class and implement the virtual function *toneMapAndDraw*, which tone maps the input HDR texture and draws

the results to the OpenGL canvas. By calling this function in each iteration of the main application loop, the real time performance of the operator can be evaluated.

At present, the only the tone mapping methods of *Slomp and Oliveira* and *Reinhard* have been implemented. Because the former method is an optimization of the latter, the abstract class `Slomp`, representing *Slomp and Oliveira*'s method, is derived from the class `Reinhard`, rather than `TMO`. Similarly, if one were to implement the methods of *Goodnight et al.* [GWW+03] or *Krawczyk et al.* [KMS05] discussed in Chapter 4, they would be derived from the class `Reinhard`. On the other hand, if *Ashikhmin et al.*'s operator [Ash02], *Pattanaik et al.*'s operator [PY02], or any other operator unrelated to Reinhard's were to be added, they would be derived directly from `TMO`.

The key to *Slomp and Oliveira*'s performance is parallel SAT generation. It was hypothesized (in Chapter 5) that using a more efficient algorithm for SAT generation would lead to significant performance gains in the overall tone mapping procedure. To verify this hypothesis, the SAT generation algorithms of *Hensley et al.* [HSC+05], *Sengupta et al.* [SLO06] and *Harris et al.* [HSO07] have been implemented (on the left in Figure 6-23). Each algorithm derives the abstract SAT interface class `SAT` and implements the virtual, implementation-specific function *compute()*.

6.3.2.2 Technologies used

To implement TMStudio, a number of technologies were used. These are:

- CMake.
- OpenGL.
- The Nokia Qt framework.
- The Cg shader language.
- The GLSL shader language.
- The CUDA C programming language.
- The OpenEXR API.

All of the technologies and libraries listed above support cross-platform development. Therefore, TMStudio has been deployed and tested on the following platforms:

- Windows XP, 32-bit.
- Windows XP, 64-bit.
- Windows 7, 32-bit.
- Ubuntu Linux 11.04, 32-bit.
- Ubuntu Linux 10.10, 64-bit.

The TMStudio source code is supplied with CD accompanying this thesis.

6.3.3 Outlook

TMStudio was developed as a cross-platform prototyping tool for developing and evaluating new tone mapping operators. Since performance was a primary focus of the research in this thesis, the tone mapping occurs in an interactive environment where the user can tweak parameters and view the results in real-time. This is a distinguishing factor from similar existing software, such as the widely used *PFSTools* [RGR+06] HDR toolset for Linux, in which a number of tone mapping operators have been implemented on the CPU. Therefore, with some further work on error handling and the user interface, in addition to serving as a prototyping platform for developers, TMStudio has potential as a user-friendly, interactive tone mapping tool, suitable for use with HDR photography.

In terms of prototyping, a number of addtional features would be useful. A list of additional features, that would be implemented given the time, is given below.

An interactive OpenGL environment
At present, the OpenGL scene rendered in the render widget is very simple: a screen-aligned quad is rendered with an attached HDR texture. To better test tone mapping operators designed for interactive environment, an interactive environment should be used. The user should able to navigate through a simple, virtual environment containing HDR light sources. This would allow the developer to test their operator on dynamic HDR input, and develop features such as *temporal adaption* [DD00] for temporal exposure adjustment.

Additional performance metrics
The current implementation of TMStudio evaluates performance entirely using measures of rendering frame rate. For more comprehensive evaluation, additional metrics are necessary. For instance, it would be useful to add a method `getMemUsage()` to the tone mapping class interface `TMO` in Figure 6-23, which returns the amount of GPU memory allocated by each method. Furthermore, implementing visual metrics, such as the *Visual Difference Predictor* [MMS04], would be useful for comparison of operator visual output.

A wider range of supported file formats
Presently, only HDR images in OpenEXR format can be loaded using TMStudio. To view other HDR images, such as those in the common *.hdr* format, one must first externally convert them to OpenEXR format prior to loading them in TMStudio. In future, support for the most common HDR file formats should be added.

6.4 Summary

Section 6.1 described how each kernel in the CUDA tone mapping module was implemented. The two most complex were those for SAT generation and dodging-and-burning. SAT generation was implemented as a submodule, encapsulated in a C++ class, using the CUDPP CUDA C library for scan computation. Four dodging-and-burning implementations were proposed; starting from a naive implementation of the dodging-and-burning algorithm given in Chapter 5, each kernel implementation was a progressive optimization of its predecessor. Ultimately, two implementations gave promising results: one using texture caching reducing global memory traffic caused by SAT access, and a second that

attempts to reduce memory traffic by loading blocks of the SAT into shared memory.

Section 6.2 detailed the implementation of the shader tone mapping module. The most challenging part of this module was the SAT generation submodule. Like the CUDA implementation, shader SAT generation is encapsulated in a C++ class. To compute row- and column-scans, an existing, work-efficient parallel scan algorithm has been implemented. Since shaders are unable to perform in-place computations, a collection of additional textures must be allocated to store partial results, leading to a larger memory footprint than that CUDA implementation. In general, it was found that implementation of the tone mapping steps outlined in Chapter 5 was significantly simpler in shaders than in CUDA.

Finally, in Section 6.3, a custom prototyping platform designed for implementing and testing high performance tone mapping operators was introduced. An overview of how to use this platform, called *TMStudio*, was given, followed by a discussion of its internal class design, which is intended to support easy integration of additional tone mapping techniques in future.

The performance of the implementations presented here will be examined in the following chapter.

7 Results

This chapter presents the visual output and performance of the tone mapping techniques described in Chapters 5 and 6.

Section 7.1 presentes the visual results of local tone mapping on HDR night driving scenes generated by VND. After comparing the headlight beam pattern detail preserved by global and local operators, each of the four parameters that control local tone mapping are discussed and their influence on output demonstrated.

A detailed look at the CUDA module developed in Chapter 6 is given in Section 7.2. In addition to measuring the overall performance of the module, individual kernel execution times are reported, giving in an indication of the distribution of work across kernels. Finally, individual kernel occupancy is listed for all current GPU architectures.

After Section 7.3 compares CUDA and shader implementations of a simple, global tone mapping module, Section 7.4 provides a comprehensive comparison and evaluation of the tone mapping modules developed in this thesis.

Finally, Section 7.5 shows the effects of reducing the number of scales used in the local tone mapping to increase performance.

7.1 Tone mapping HDR night driving scenes

The long term goal of the research presented in this thesis is to develop a high speed, local tone mapping module that can be integrated into the Virtual Reality headlight prototyping platform VND. Although the focus of this thesis is the development of this module, and, due to time restrictions, its integration into VND has been left as future work, initial impressions on the effect on tone mapping night driving scenes have been attained by extracting HDR scenes from a modified, HDR version of VND. Once HDR scenes were extracted from VND, they were opened with TMStudio for an initial, informal assessment of the effects of tone mapping parameters on HDR scenes illuminated by virtual headlights.

7.1.1 Global vs local tone mapping

Since global tone mapping operators are significantly faster and simpler to implement than their local counterparts, they have been the method of choice for most interactive HDR applications. Consider, for example, the tone mapping system recently developed for a night driving simulator similar to VND called *Rivoli* [PB10]. Considering that high performance is a crucial requirement on the tone mapping module (Requirement 2 defined in Chapter 3), it is tempting to use a global operator for tone mapping VNDF. When

comparing the effects of global and local operators on night driving scenes in TMStudio, however, it is clear that important headlight distribution detail is lost in global tone mapping. Figure 7-1 shows the difference between tone mapping using global and local instances of *Slomp and Oliveira*'s method in VND.

The scenes in Figure 7-1 and 7-2 are each illuminated by different headlight prototypes and tone mapped using global and local operators. Clearly, significant beam pattern detail lost by the global operator is preserved by its local counterpart. Since the primary focus of VND is to examine, in detail, the manner in which the beam pattern of a headlight prototype illuminates its surroundings, it is necessary to use local tone mapping in VND (satisfying Requirement 1).

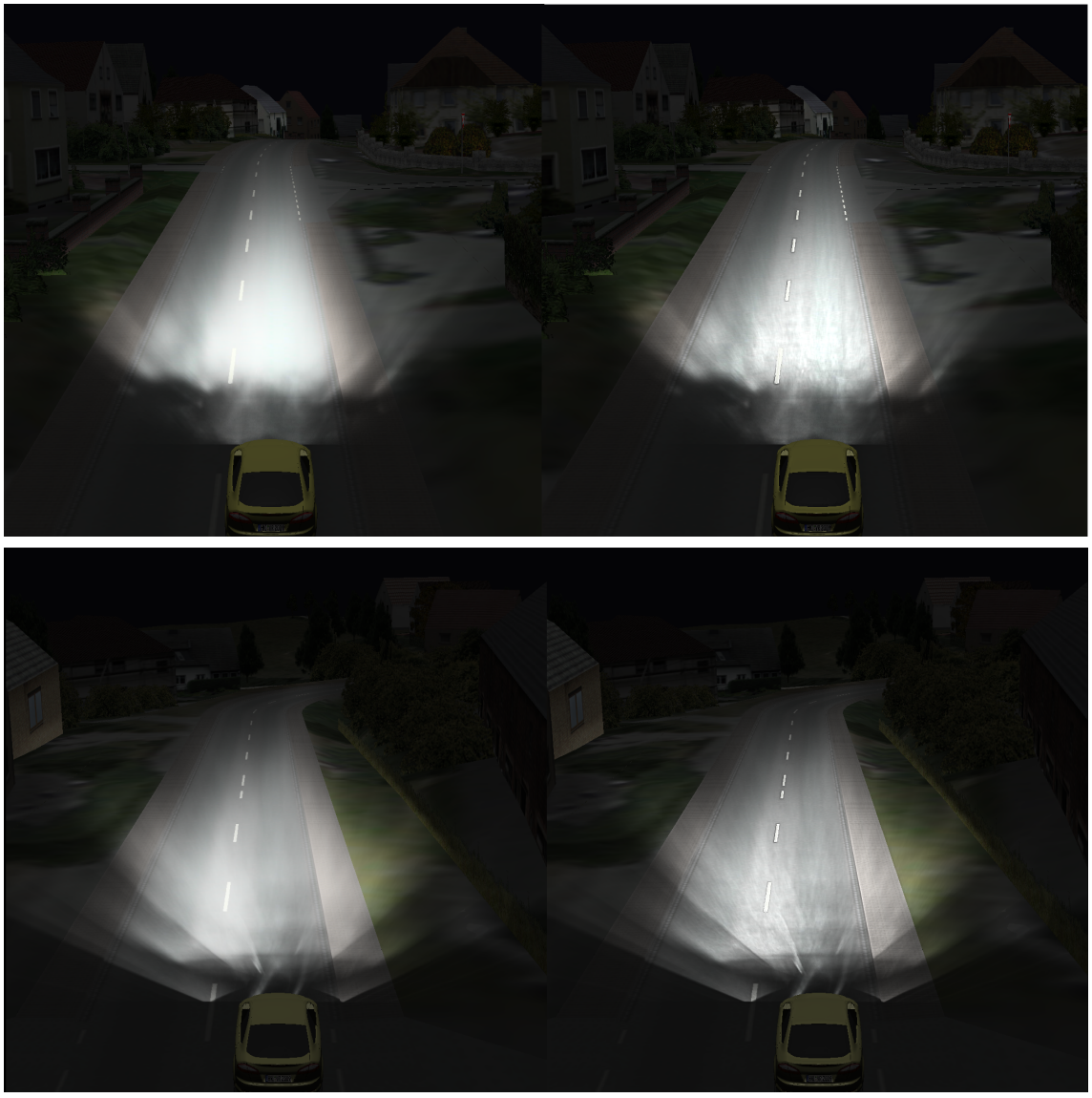


Figure 7-1: The effects of global (left) vs local (right) tone mapping on HDR night driving scenes.

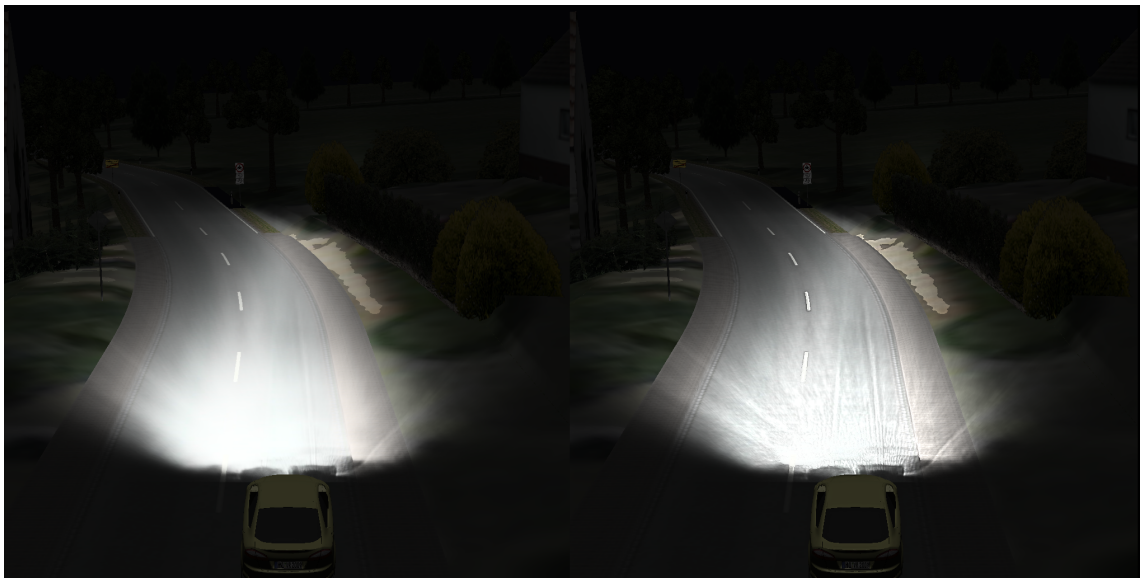


Figure 7-2: The effects of global (left) vs local (right) tone mapping on HDR night driving scenes.

7.1.2 Tone mapping parameters

In addition to its efficiency and high visual quality, one of the main motivations for selecting *Slomp and Oliveira*'s method as the basis for this investigation was its self-sufficiency: its quality is affected by only a small number of input parameters, and the default parameter values (given in [SO08] and [RSS$^+$02] and reiterated in Chapter 4) suffice for most scenes. Therefore, this operator is ideal for implementation as a self-contained post-processing module, meeting Requirement 3.

Four parameters affect tone mapping results: the parameter α used in scaled luminance computation (Equation 4.2), which is analogous to camera exposure in traditional photography; a gamma parameter, γ, for adjusting the color saturation of the final LDR scene (Equation 5.5); a sharpness parameter, ϕ, involved in evaluating the center-surround function between two adjacent levels of the box filter pyramid (defined in Equation 4.5); and the threshold ϵ, above which the aforementioned center-surround function must evaluate for identification of a Local Region (Section 4.1.2).

The effects of varying the exposure parameter α are shown in Figure 7-3. As described in Chapter 4, dark scenes require a higher value of α than lower ones. A default value of $\alpha = 0.18$ is recommended in [RSS$^+$02]. As can be seen in Figure 7-3, night driving scenes are comparatively dark, thus requiring higher values of α than the recommended default value. Experience experimenting with TMStudio has shown that a value between 0.72 and 1.0 gives best results. When integrating the tone mapping module into VND, an automatic α adjustment technique proposed by *Reinhard et al.* [RWP$^+$06] should be considered. Furthermore, since VND is a dynamic environment, and scene contrast can change drastically between frames as the headlight itself moves in and out of the view frustum, a method for smoothly adjusting α to its changing environment should be considered. This processes, which mimics the phenomenon of temporal adaption in

the human visual system, has been implemented in interactive systems by [DD00] and [GWW+03].

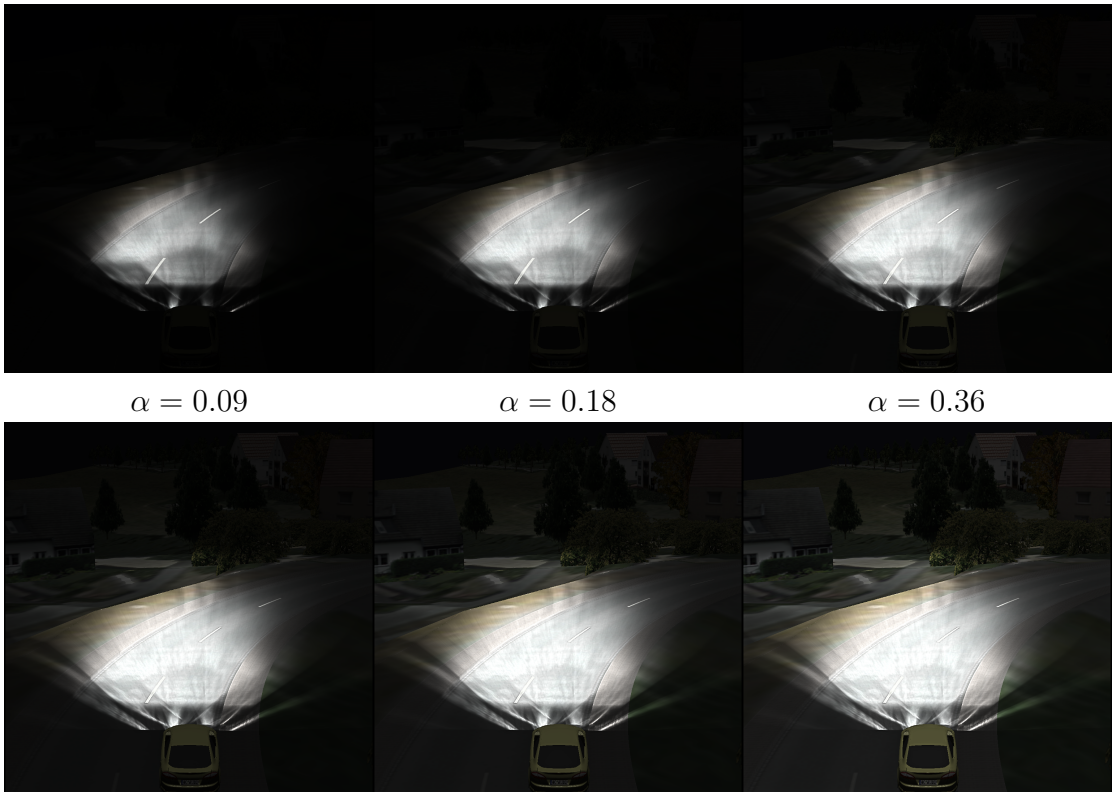
Figure 7-4 shows a VND scene tone mapped with varying values of γ. The γ parameter is simply one of color saturation; setting it to 0 results in an entirely monochrome scene, while a value of 1 reproduces the HDR input colors completely. Experimentation with TMStudio showed that setting $\gamma = 1$ gave appealing results for the entirely artificial scenery of VND, while values ranging between 0.6 and 0.8 gave more desirable output for many HDR photographs. In general, varying the parameter γ has no effect on performance and only a minor effect on scene appearance. Its value can be set to accommodate the personal taste of each user.

The parameters ϕ and ϵ both affect the local features of the operator. More specifically, since ϕ has a direct effect on the center-surround function defined in Equation 4.5 and ϵ is the threshold against which the result of this function is compared, both parameters affect the size of the Local Region identified for each image pixel.

The final step of the local tone mapping procedure is to compress the luminance of each pixel against the average luminance within its Local Region. Therefore, the smaller the Local Region, the fewer local surroundings are taken into account during tone mapping. In the limit case, when the Local Region converges to the target pixel itself, the surroundings of each pixel have no effect on the final result, and a global operator is reached. On the other hand, as the size of the Local Region increases, ever more of each pixel's surroundings are used for its compression. If the Local Region encompasses high contrast edges, where bright regions directly border with darker ones, such as the silhouette of a mountainside against a setting sun or a reflective lane marker on a dark road, these edges will be exaggerated. This exaggeration occurs because either side of a high contrast edge is comparatively bright or dark compared to the average of its Local Region, mapping it to a particularly dark or bright region of the luminance compression transfer function (see Figure 4-3).

The default values of ϕ and ϵ are specified in [SO08] to be 8 and 0.025, respectively. The top frame of Figure 7-5 shows a scene in VND tone mapped with these default local parameters, cropped to focus on the headlight distribution. A close inspection of the first two lane markers in the center of the road reveal a slight overenhancement of their edges. This effect is small, however, and causes no qualitative degradation to the scene. Reducing ϵ to 0 causes all evaluations of the center-surround function to identify one Local Region: that of the target pixel itself. Therefore, the result is a global operator, which is shown in the center pane in Figure 7-5. Finally, setting ϵ to 0.05, which is the default threshold value for Reinhard's local operator, leads to an overestimation of Local Region size. A close look at the first two lane markers in the bottom pane in Figure 7-5 reveales "halo artifacts" in their immediate surroundings.

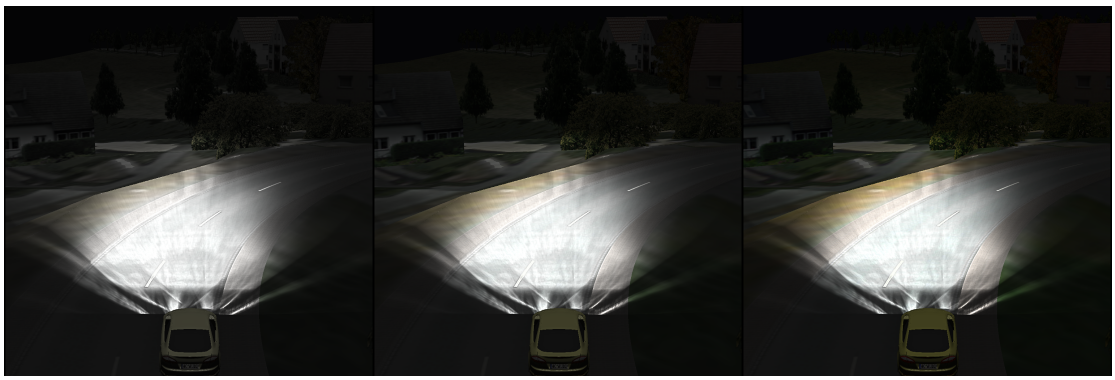
Varying the sharpness parameter ϕ has a similar, though less profound, effect on high contrast edges. In the top pane of Figure 7-6, ϕ has been reduced to 6 from its default value of 8, while leaving ϵ at its default of 0.025. This has the effect of softening the hard lane marker outlines, at the expense of a slight loss of detail in the headlight beam pattern. Increasing ϕ to 10, as shown in the bottom pane of Figure 7-6, once again leads to an overestimation of Local Regions, causing highly noticeable halo artifacts surrounding high contrast edges.



$\alpha = 0.09$ $\alpha = 0.18$ $\alpha = 0.36$

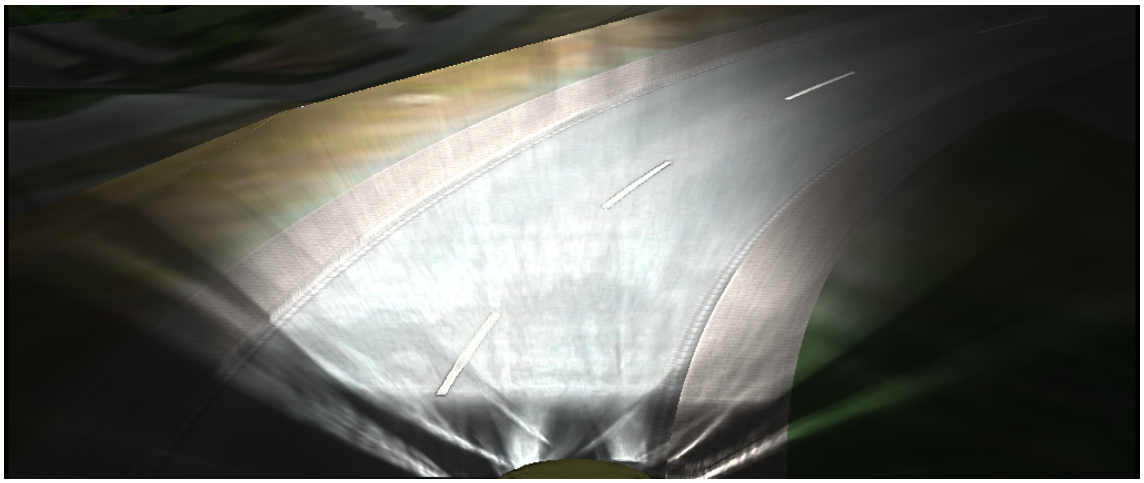
$\alpha = 0.72$ $\alpha = 1.0$ $\alpha = 1.44$

Figure 7-3: The effects of varying the exposure parameter α

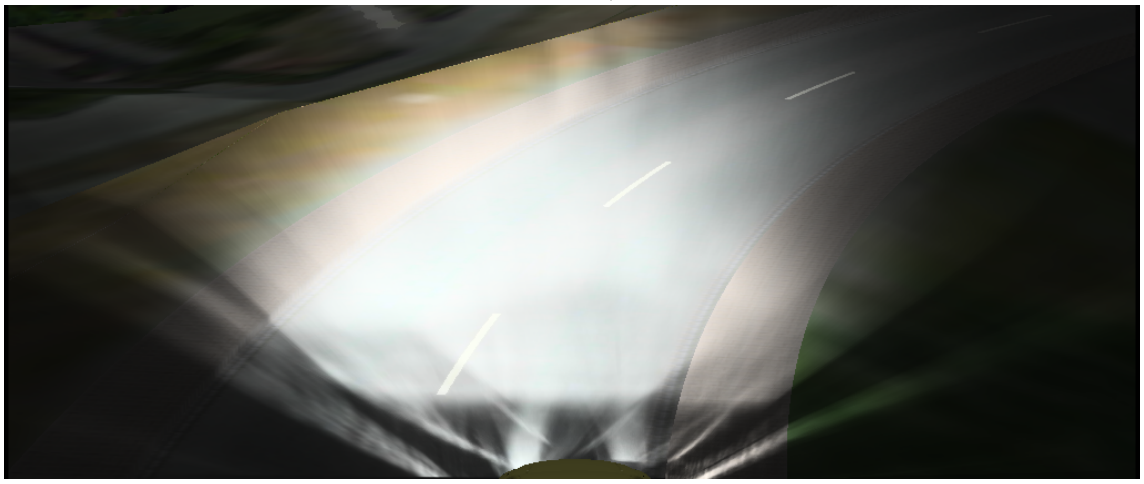


$\gamma = 0.30$ $\gamma = 0.60$ $\gamma = 1.00$

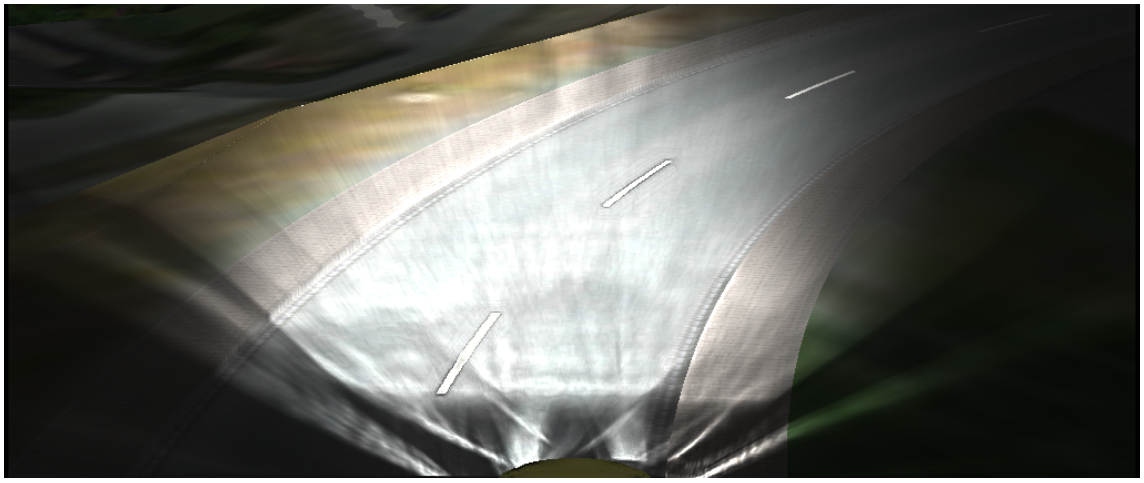
Figure 7-4: The effects of varying the colour saturation parameter γ



$\epsilon = 0.025$, $\phi = 8$

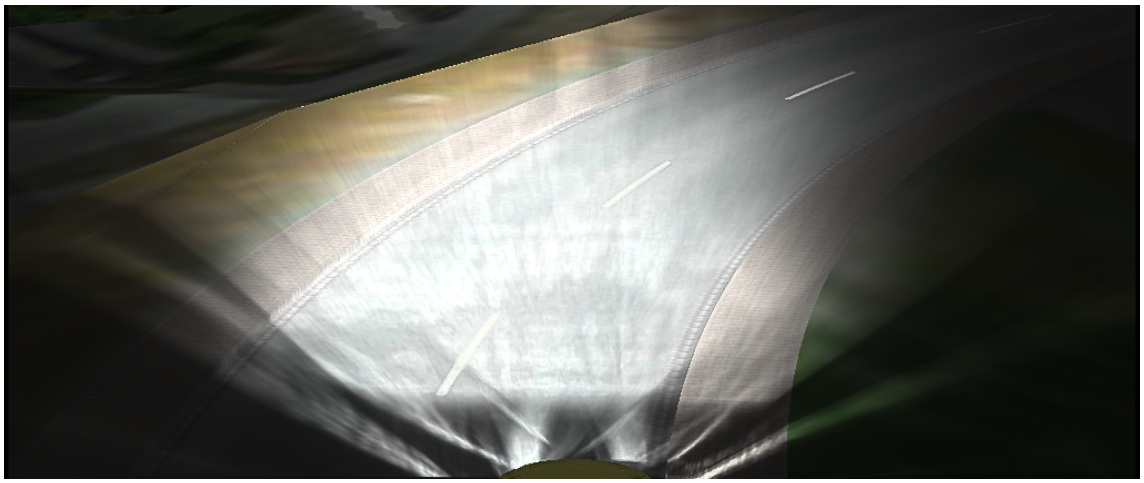


$\epsilon = 0.00$, $\phi = 8$

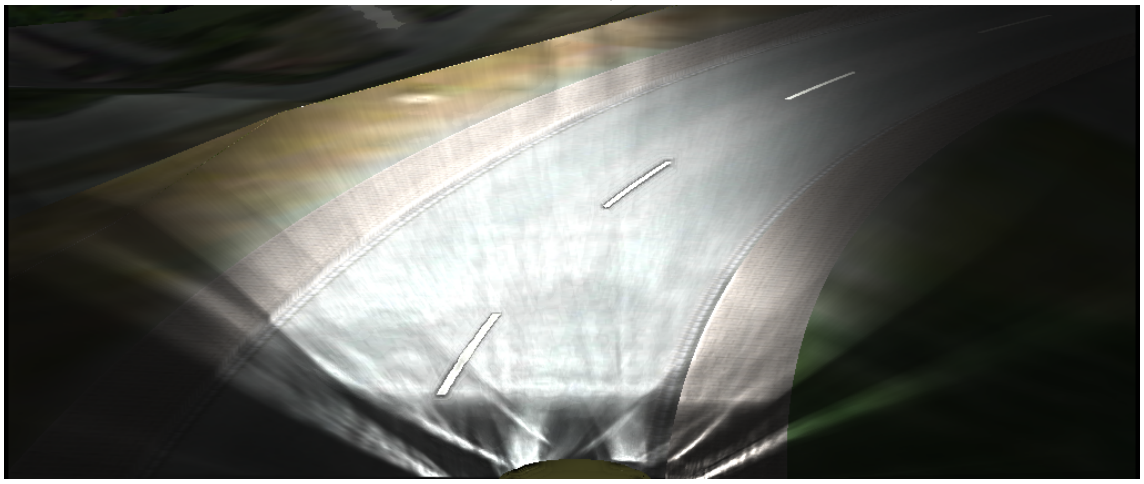


$\epsilon = 0.05$, $\phi = 8$

Figure 7-5: Varying the threshold ϵ, with ϕ fixed to 8.



$\epsilon = 0.025$, $\phi = 6$



$\epsilon = 0.025$, $\phi = 10$

Figure 7-6: Varying the sharpness parameter ϕ, with ϵ fixed to 0.025.

7.1.3 Summary

This section presented an informal examination of the visual results of tone mapping HDR night driving scenes with the tone mapping modules developed in Chapters 5 and 6. After comparing the visual results of these modules with the output of a standard global operator, and thereby confirming the choice of using a local operator (to meet Requirement 1), the effects of each of the four tone mapping parameters, with which the modules are controlled, were discussed.

A formal evaluation of the visual quality of tone mapping in night driving scenes, as well as a study into the impact of various tone mapping parameters on visual output, lies outside the scope of this thesis. Generally, all tone mapping modules developed in Chapters 5 and 6 are implementations of *Slomp and Oliveira*'s tone mapping method (see Section 5.1), and consequently produce equivalent visual results to this method. For an in-depth evaluation of the visual output of *Slomp and Oliveira*'s method, see [SO08], [LWR+09]. The remainder of this chapter focuses on the performance of the tone mapping implementations developed in this thesis.

7.2 The CUDA tone mapping module

Section 5.4 introduced the notion of a local tone mapping module implemented in CUDA. The details involved in implementing this module were then described in detail in Section 6.1. This section examines the performance of the CUDA tone mapping module as a whole, as well the individual kernels from which it is comprised.

7.2.1 Performance

Sections 6.1.5.1 to 6.1.5.5 presented a number of approaches for implementing and optimizing the most complex kernel in the CUDA tone mapping module: the fifth and final kernel (in order of execution) in the module, which uses the partial results produced by all other module kernels to compute the final tone mapping by means of dodging-and-burning. Each subsection concluded by listing the execution times of some specific optimization of this kernel for a number of input images.

Four dodging-and-burning kernel implementations were discussed in Chapter 6: a naive kernel, which is the most direct translation of the dodging-and-burning algorithm given in Section 5.4.5 into CUDA; a modification of the naive kernel using texture memory for SAT access; a modification of the texture kernel using unrolled loops; and an implementation that reduces global memory traffic by using shared memory for SAT access. The overall performance achieved by CUDA tone mapping modules using each of these kernels to tone map a set of input HDR scenes, ranging in resolution from 1024x1024 to 2048x2048, is shown in Figure 7-7.

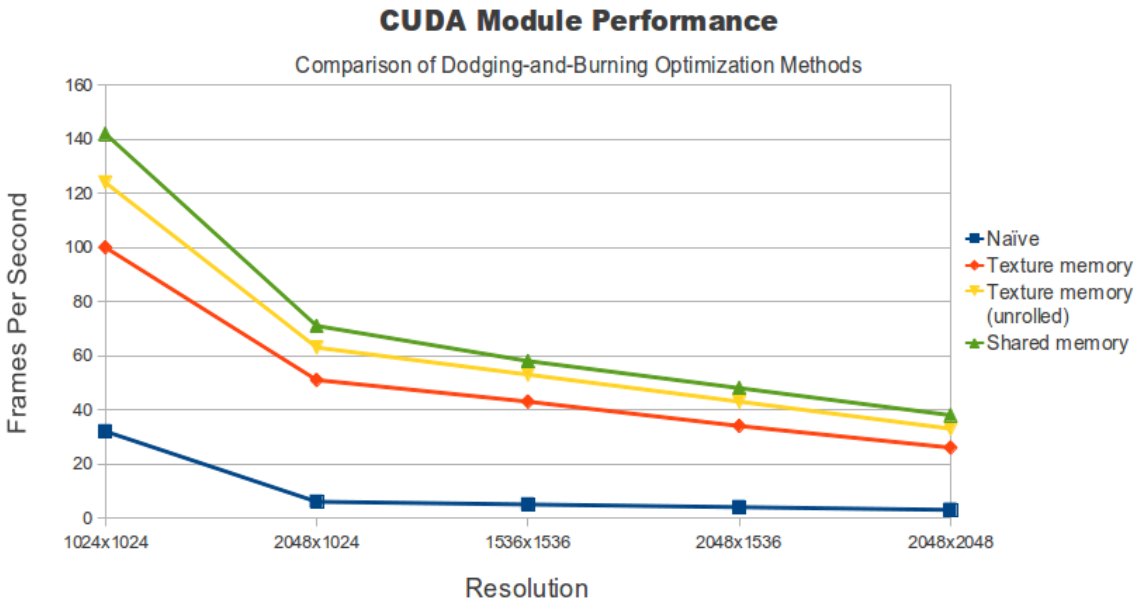


Figure 7-7: The performance of a number of implementations of the CUDA tone mapping module. Performance measurements were made for input frame resolutions ranging from 1024x1024 to 2048x2048 pixels.

The two most interesting implementations are those using the unrolled texture memory and shared memory dodging-and-burning kernels. The performance measurements in

Figure 7-7 show that both of these implementations maintain interactive frame rates, even at a resolution of 2048x2048, which exceeds the maximum PC system resolution called for by Requirement 2. Although the shared memory implementation consistently, though marginally, outperforms its unrolled texture memory counterpart, the unrolled texture memory implementation remains an attractive option. As was discovered in Chapter 6, implementing a dodging-and-burning kernel that uses texture memory for SAT access is substantially simpler than implementing one that loads segments of the input SAT into shared memory. Furthermore, the shared memory implementation is more intimatley dependent on the hardware properties of each GPU on which it is run; the amount of shared memory that can be used by each thread block depends on the amount of shared memory that each specific GPU has to offer. This also affects the number of threads that can be executed per block, as well as multiprocessor occupancy (see Section 6.1.5.5).

Nonetheless, because it provides the best performance, throughout the remainder of this thesis the term "'CUDA tone mapping module"' refers to a module using the shared memory dodging-and-burning kernel implementation, unless otherwise specified.

7.2.2 Kernel properties

The design of the CUDA tone mapping module was shown in Figure 5-16 in Section 5.4. A total of four kernels and one submodule are used for tone mapping: a luminance computation kernel, a reduction kernel, a scaled-luminace computation kernel, a SAT generation submodule (which uses a number of internal kernels) and a dodging-and-burning kernel. To tone map each frame, the CUDA module invokes each of these kernels or submodules in sequence, where each kernel/submodule computes partial results from the output of its predecessors.

7.2.2.1 Execution times

Figure 7-8 shows the average computation time of each kernel during 30 seconds of tone mapping a 2048x1536 HDR scene, using the fastest, shared-memory implementation.

Kernel/Submodule	GPU Time (ms)
misc. memory ops.	0.53
computeLuminance	0.73
reduce	0.10
scaleLuminance	0.50
SAT generation	2.17
dodgeAndBurn (SM)	9.81
Total	13.85

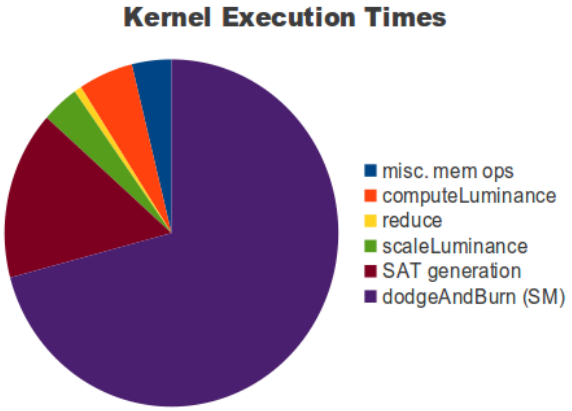


Figure 7-8: Execution times of each kernel/submodule in the CUDA tone mapping module.

The table on the left in Figure 7-8 lists the exact measured kernel/submodule execution times in milliseconds, and the pie chart on the right shows the proportion of total ex-

ecution time spent in each kernel. SAT generation is accomplished using a number of kernels - their aggregate time is given here. The entry "misc. memory ops." refers to miscelleneous memory operations, such as partial result dataset copies, performed between kernel invocations.

When considering the total of the computation times given in 7-8, one may mistakenly expect a final frame rate of $1000/13.85 \approx 72$ fps. In fact, Figure 7-7 shows that the actual frame rate achieved is just under 50 fps. This can be attributed to processing performed by the host application (in this case TMStudio), overhead of invoking the tone mapping module, and the fact that the times given in Figure 7-8 are GPU execution times; the additional time spent invoking each kernel and waiting for its result to return to the CPU are not accounted for here.

7.2.2.2 Occupancy

The concept of *occupancy* was introduced in Section 6.1.5.5. As stated in Section 6.1.5.5, occupancy is the ratio of active warps per multiprocessor to the maximum possible warps per multiprocessor. Generally, the closer a kernel's occupancy is to 1, the better its expected performance. Because the number of threads schedulable per multiprocessor depends on how much shared memory and how many registers are available to each multiprocessor, a kernel's occupancy is hardware dependent. CUDA-capable GPU devices are attributed with a property, called *compute capability*, that defines the basic hardware capability they provide to CUDA applications. A kernel's occupancy depends on the compute capability of the GPU on which it is run. The NVIDIA GeForce 8800GTX, used by the development system described Appendix A, has a compute capability of 1.0, whereas the NVIDIA Quadro FX 5800 GPUs used in the HD Visualization Center have compute capability 1.3.

Using the NVIDIA *CUDA Occupancy Calculator* (introduced in Section 6.1.5.5), the occupancy of a CUDA kernel can be determined for a target GPU of any given compute capability. Figure 7-9 lists the occupancies of the most important kernel developed for the CUDA tone mapping module for each of the compute capabilites currently supported by CUDA.

Kernel Occupancies					
Kernel/Submodule	**Compute Capability**				
	1.0	**1.1**	**1.2**	**1.3**	**2.0**
computeLuminance	100%	100%	100%	100%	100%
reduce	100%	100%	100%	100%	100%
scaleLuminance	100%	100%	100%	100%	100%
dodgeAndBurn (TM unrolled)	33%	33%	75%	75%	100%
dodgeAndBurn (SM)	33%	33%	25%	25%	83%

Figure 7-9: Tone mapping module kernel occupancies for the currently supported compute capabilities. TM stands for texture memory, SM for shared memory.

Since SAT generation is accomplished using the CUDPP CUDA library, and the transpose is performed by an efficient kernel provided in the CUDA 3.2 SDK, the SAT generation kernels have been omitted from Figure 7-9. The two most interesting kernels shown in

the figure are “dodgeAndBurn (TM unrolled)” and “dodgeAndBurn (SM)”, where TM stands for texture memory and SM for shared memory. It can be seen that, for compute capabilities greater that 1.0 (and particularly for 1.2 and 1.3), the texture memory kernel has a higher occupancy that its shared memory counterpart. This is in part due to the higher register load of the shared memory kernel (mainly due to the additional computation required for the transposed mapping from texture memory to shared memory shown in Figure 6-14) and in part due to its significantly higher shared memory demand.

7.3 Global tone mapping

This section details the performance of Reinhard’s simple and efficient global tone mapping operator, when implemented both in CUDA and with shaders.

Figures 5-16 and 5-19 in Chapter 5 show the respective internal designs of the CUDA and shader tone mapping modules. In both of these figures, there are a series of parallel GPU programs (kernels in the CUDA module and shaders in the shader module) that operate on a collection of datasets residing in GPU memory. Programs that operate globally, i.e apply the same computation to each input element, are gathered into a region in each figure marked with a light grey backdrop, while operations required for local tone mapping are grouped in the dark grey areas. In both cases, by replacing all operations in the dark grey, local regions with a single, simple kernel or shader that computes Reinhard’s global operator (Equation 4.3 in Chapter 4), a complete global tone mapping operator is implemented.

Figure 7-10 compares the performance of CUDA and OpenGL GLSL shader implementations of Reinhard’s global tone mapping operator, for a set of input images with resolutions ranging from 512x512 to 2048x2048. Performance was measured using the TM-Studio (Section 6.3) running on the development system detailed in Appendix A.

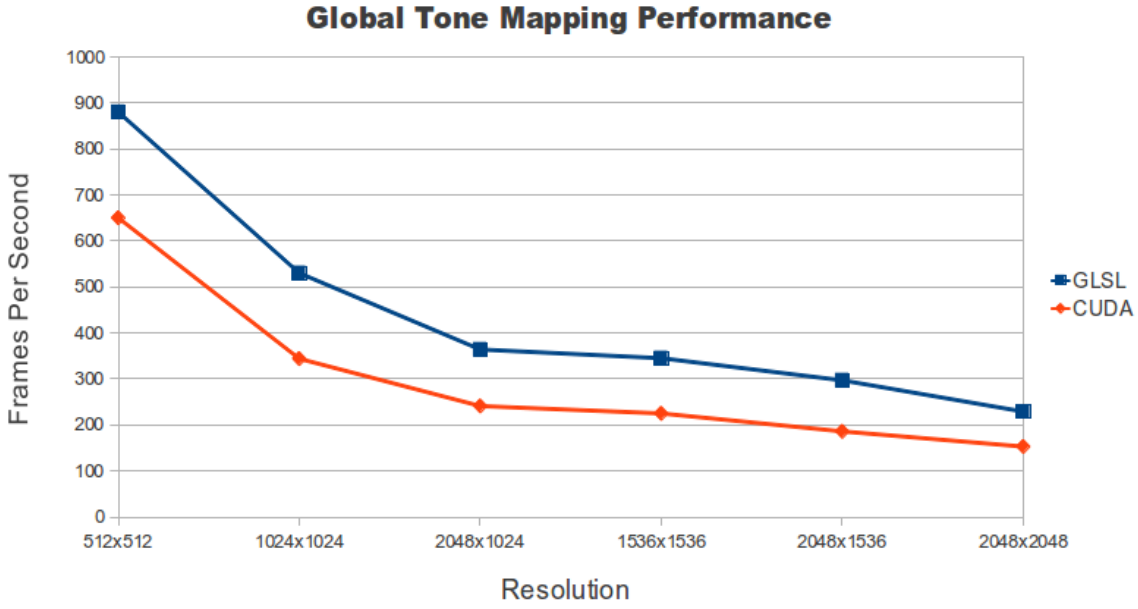


Figure 7-10: Global tone mapping: shaders vs CUDA.

It is apparent from Figure 7-10 that the GLSL implementation consistently outperforms

an equivalent CUDA implementation. This has a number of reasons.

Firstly, as mentioned in Section 5.5, the scene key computation in the GLSL implementation is accomplished by means of hardware assisted mipmapping, which means that no reduction shader is required. It has been previously noted that this only gives correct results for square input textures; the resolutions in Figure 7-10 that are non-square were tone mapping with an incorrect scene key. Although this gives the visual impression of false exposure, there is no effect on performance. Therefore, whether a scene is tone mapped with the correct scene key is irrelevant in the context of the performance evaluation shown in Figure 7-10 .

Secondly, both modules operate at frame rates well beyond those required to maintain interactivity of the host application. For example, when used by 1536x1536 HDR application, the GLSL global tone mapping module requires approximately 3 ms per frame for tone mapping. When the tone mapping modules perform their task so quickly, the overhead involved in calling them from the host application is no longer negligible. Since CUDA, unlike GLSL shaders, is unable to write its results directly to texture, its results are written to a Pixel Buffer Object (PBO), which is then "unpacked" into a texture (see Section 5.3 for more details). This additional unpacking step involves a complete copy of each frame within GPU memory, which accounts for the slight performance discrepancy apparent in Figure 7-10. Nonetheless, both modules effortlessley meet the performance requirements for standard PC systems specified by Requirement 2. Unfortunately, Reinhard's global operator loses significant detail during luminance compression (Section 7.1 demonstrates the effects on VND beam pattern simulation), and thus, both modules fail to meet Requirement 1.

Despite the loss of high contrast detail, the simplicity and performance of the global tone mapping modules make them ideal for rapidly visualizing and debugging HDR scenes.

7.4 Comparison of CUDA and shader implementations

This section evaluates, from a performance perspective, the CUDA and shader tone mapping modules by comparing them to each other and to a shader implementation of *Slomp and Oliveira*'s original method. After first measuring and comparing the efficiency of SAT generation using each method, the overall tone mapping performance, as measured in TMStudio, is reported. Motivated by these results, an additional, hybrid CUDA/GLSL module is then introduced, which takes advantage of the best features of both langauages. Finally, the tone mapping system is tested on the Heinz Nixdorf Institute HD Visualization Center (see Section A.2).

7.4.1 A shader implementation of *Slomp and Oliveira*'s original method

To determine if the CUDA and shader tone mapping modules conceived and implemented in Chapters 5 and 6 have successfully improved on the performance achievable by *Slomp and Oliveira*'s original operator, it is necessary to run an instance of their operator on the same platform as the two newly developed modules, testing all methods with the same input. In order to conduct such experiments, an implementation of *Slomp and Oliveira*'s original method is required.

Fortunately, a complete CG shader implementation of *Slomp and Oliveira*'s original method was developed by Matthias Linnemann as a part of his investigation into tone mapping with Augmented Reality [LWR+09], and made available to support the research in this thesis.

Linnemann's implementation, which was originally designed to work with the OpenGL scenegraph API *OpenSceneGraph* [BO04], was integrated into TMStudio. During integration, a number of modifications were made. Most of these were minor changes required to execute Linnemann's module using pure OpenGL, rather than OpenSceneGraph. One modification, however, was in fact an optimization and had a significant impact on performance.

The tone mapping method developed by Linnemann uses full-precision, 32-bit-per-channel HDR textures to store partial results, such as the luminance map and scaled luminance SAT. All of these textures are allocated with three color channels and one alpha channel, a texture format represented in OpenGL by the enumerant `RGBA32F`. Many of the shaders operating on the partial result textures use all of these channels to save their results. All operations on the scaled luminance SAT texture, including its generation and use for box filter computation, use only the red color channel in their computations. As a result, for each of the many SAT texture lookups performed during tone mapping, 12 bytes of precious texture cache memory are wasted, resulting in an excess of unnecessary global memory activity. By simply changing the scaled luminance SAT texture format from `RGBA32F` to the single channel `GL_INTENSITY32F` format, global memory traffic was reduced and performance increased. Section 7.4.3 reports exact performance gains achieved using this optimization.

7.4.2 SAT generation

The central motivation for using CUDA to implement the post-processing tone mapping module proposed in Chapter 3 was an expected speedup in SAT generation, because the parallel *scan* operation, which is the main operation used for SAT generation, can be computed faster using CUDA than with traditional shading languages (see Section 5.2).

A SAT is generated from an input image by computing scans on each row (row scans), the computing scans on each column of the result (column scans). When computing SATs in CUDA, extra care must be taken to use optimal memory access patterns. Because reading global memory in rows is more efficient than in columns, rather than computing column scans, after the first row scan the image is transposed and a second row scan is applied (see Section 5.4.4). Therefore, the operations used for generating SATs in CUDA fall into two categories: those involved in scan computation, and those used for transposition.

The CUDA runtime environment provides primitives, called *events*, which facilitate accurate measurement of CUDA kernel execution times. Events were used extensively in the `SATHarris` class presented in Section 6.1.4 to measure the time spent for scan and transpose computation during each SAT generation.

Figure 7-11 shows the times measured for generating SATs of various sizes, where the input datasets range in resolution from 1024x1024 to 5120x5120. The size of each dataset between these two extremes for which a measurement was made is explicitly labelled on the x-axis. The dashed blue line represents the time spent on scan computation and the

dashed red line displays the time spent on transpose computation. The total time required for each scan, which is the sum of scan and tranpose computation, is represented by the solid green line. All computations and measurements were performed on the development system described in Appendix A.

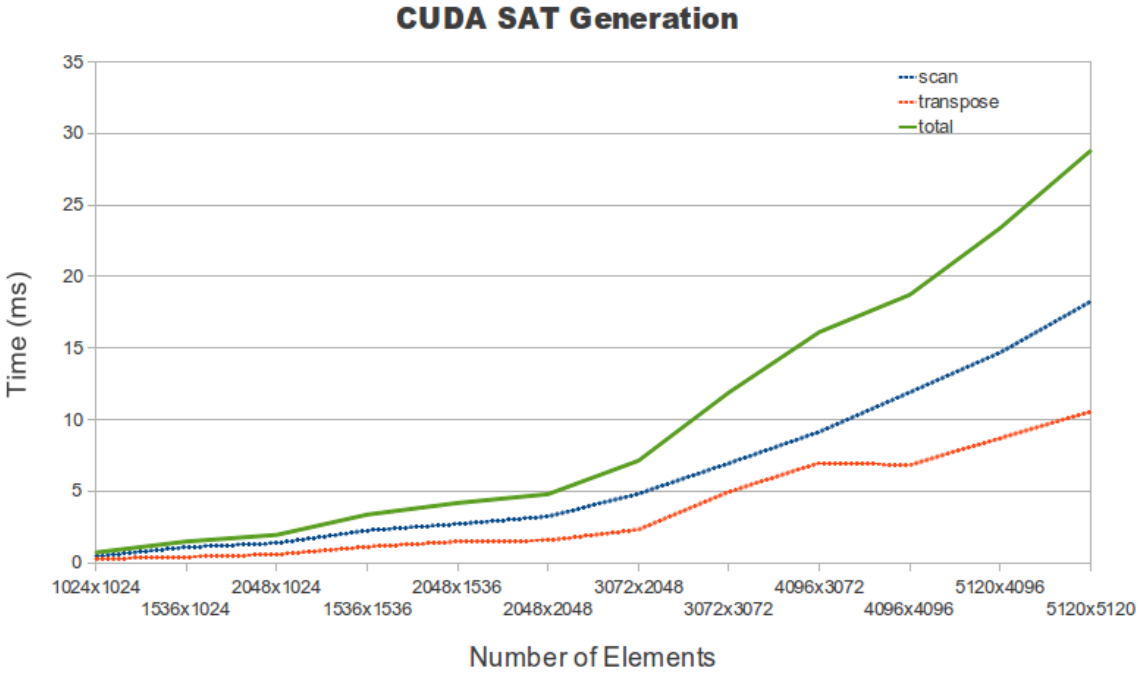


Figure 7-11: CUDA SAT generation performance. The dashed red line represents time spent on transpose computations, the dashed blue line represents scan computation and the solid green line represents total SAT generation time.

It should be noted that, because the execution times shown in Figure 7-11 were measured using CUDA events, the execution times reported are *GPU times*. GPU times show only the time spent computing on the GPU; the overhead involved in invoking each kernel and waiting for results to return (on the CPU side) are not accounted for here. Nonetheless, the execution times shown in 7-11 are encouraging. Theoretically, if SAT generation were the only operation involved in tone mapping, interactive frame rates would be possible for input resolutions of up to 5120x5120 on the development system.

Transpose computation in CUDA SAT generation is additional work required to satisfy CUDA memory access requirements; there is no equivalent operation involved in shader SAT generation procedures. Therefore, only the blue dashed line in Figure 7-11 represents *necessary* computation. If CUDA were capable of writing directly to texture memory, in the same manner as shaders do, the transpose step would no longer be necessary. In this case, the total SAT generation times would likely match the blue dashed line, rather than the solid green one. It remains to be seen if this capability will be introduced in future CUDA releases.

To determine whether the performance benefits caused by high-speed, shared memory based scan computation outweigh the cost of transpose computation, it is necessary to compare the CUDA SAT generation procedure with an equivalent shader implementation. Furthermore, considering that it was assumed that the CUDA implementation developed in this thesis would outperform the method used in *Slomp and Oliveira*'s tone mapping

implementation, it would be highly desirable to confirm this by directly comparing the two methods.

Fortunately, a CG implementation of the original SAT generation used by *Slomp and Oliveira*, provided by *Linnemann et al.* [LWR+09], as well as a GLSL implementation of SAT generation based on the parallel scan algorithm proposed by *Sengupta et al.* [SLO06], which was implemented as a part of the shader tone mapping module presented in Section 5.5, both exist in TMStudio. Accurately measuring their performance, however, is not an entirely straightforward task.

Unlike CUDA, shader programming languages provide no primitives for measuring shader execution times. Furthermore, as far as could be determined, no reliable external tools exist for this purpose. Therefore, in the absence of satisfactory means for measuring GPU execution times, shader computations can be measured from the CPU side, resulting in a measure of *CPU time*. Care must be taken when evaluating CPU time; since shader invocations are asynchronous, simply measuring the CPU time before and after a call to the shader SAT generation procedure is insufficient.

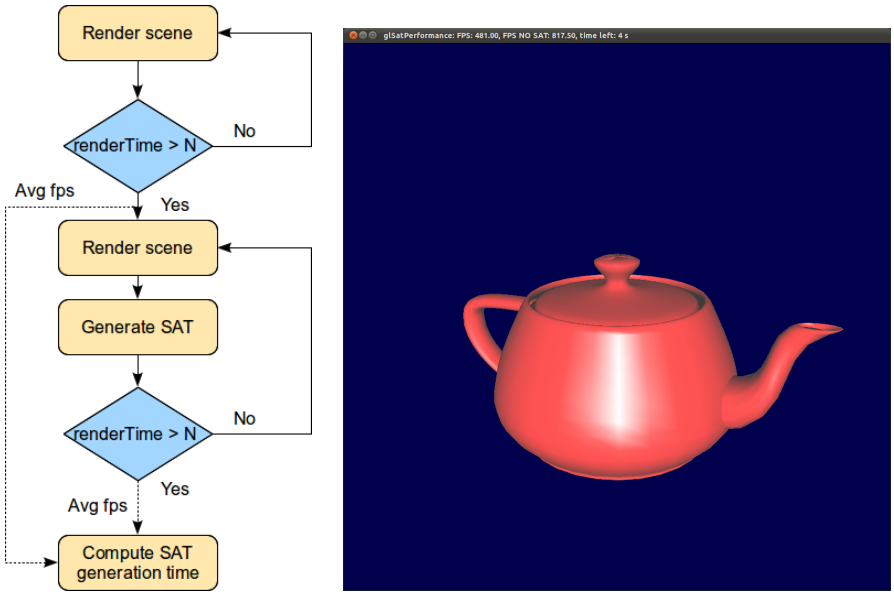
A utility program was developed for measuring shader SAT generation times. The utility program, called `glSatPerformance` and developed as a part of TMStudio, measures shader execution time as follows. First, a standard OpenGL scene is set up and displayed in a window on the desktop. In this case, the OpenGL scene is a rotating *Utah Teapot* (shown in Figure 7-12). After the first N seconds of execution are complete, the average frame rate during this execution is computed. The application then continues to update its scene for an additional N seconds, this time generating a SAT once per frame rendered. Once again, after N seconds, the average frame rate of this new render loop is computed. By comparing the average frame rates over the two N second intervals, the average SAT generation time can be determined. Figure 7-12 (a) shows this procedure as a flowchart, and Figure 7-12 (b) shows a screenshot of the scene during this process.

Using `glSatPerformance`, the CPU times of the SAT generation methods implemented in TMStudio, including the CUDA SAT generation submodule, were measured for a set of input resolutions ranging from 1024x1024 to 3072x3072. The results are shown in Figure 7-13.

The three SAT generation algorithms shown in Figure 7-13 are named after the author of the parallel scan algorithms that they incorporate. The original scan method used by *Slomp and Oliveira* is that of recursive doubling, proposed by *Hensley et al.* [HSC+05] (Section 5.2.3), the method used by the GLSL SAT generation module developed in this thesis is that of *Sengupta et al.* [SLO06] (Section 5.2.4) and the method used by the CUDA implementation is that of *Harris et al.* [HSO07] (Section 5.2.5).

It can be seen in Figure 7-13 that the CUDA implementation, despite its additional transpose computation, performs the best of the methods. The GLSL Sengupta implementation closely follows the performance of its CUDA counterpart, until input size extends beyond a resolution of 3072x2048, where the GPU is unable to provide the memory required for storing intermediate result textures. Considering the results shown in Figure 7-11, it is clear that the CUDA implementation can process inputs of substantially higher resolutions without encountering similar memory issues.

As expected, the implementations based on Harris' and Sengputa's work-efficient meth-



(a) (b)

Figure 7-12: The utility program developed for measuring CPU time of shader SAT generation procedures. On the left (a) is a flowchart that shows the execution flow of the utility program and on the right (b) is the OpenGL scene rendered by the program.

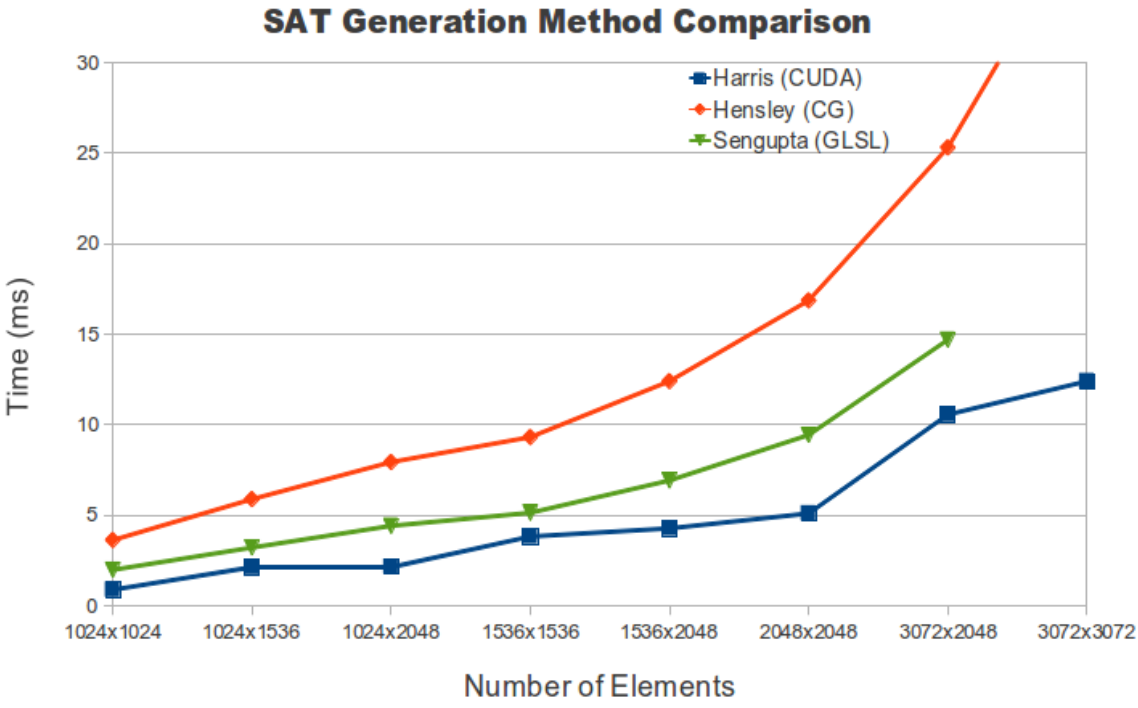


Figure 7-13: SAT generation times measured for three different approaches, each named after the author of the scan algorithm that they used. The language used to implement each method is specified in brackets after its name.

ods both significantly outperform the implementation using Hensley's method. This performance discrepancy increases with the input resolution. As discussed in Section 5.2.3, because Hensley's scan processes unnecessary elements, its computation is more rapidly serialized than that of the other two methods.

7.4.3 Local tone mapping performance

Using TMStudio, four tone mapping implementations were compared: the CUDA tone mapping module introduced in Section 5.4, the GLSL tone mapping module introduced in Section 5.5, an implementation of *Slomp and Oliveira*'s original method supplied by Linnemann [LWR+09], which uses 32-bit-per-channel RGBA format textures for SAT representation, and a modification of Linnemann's implementation, which uses single-channel 32 bit textures for SAT representation (see Section 7.4.1). The performance of these methods, measured on the development system detailed in Appendix A, is shown in Figure 7-14.

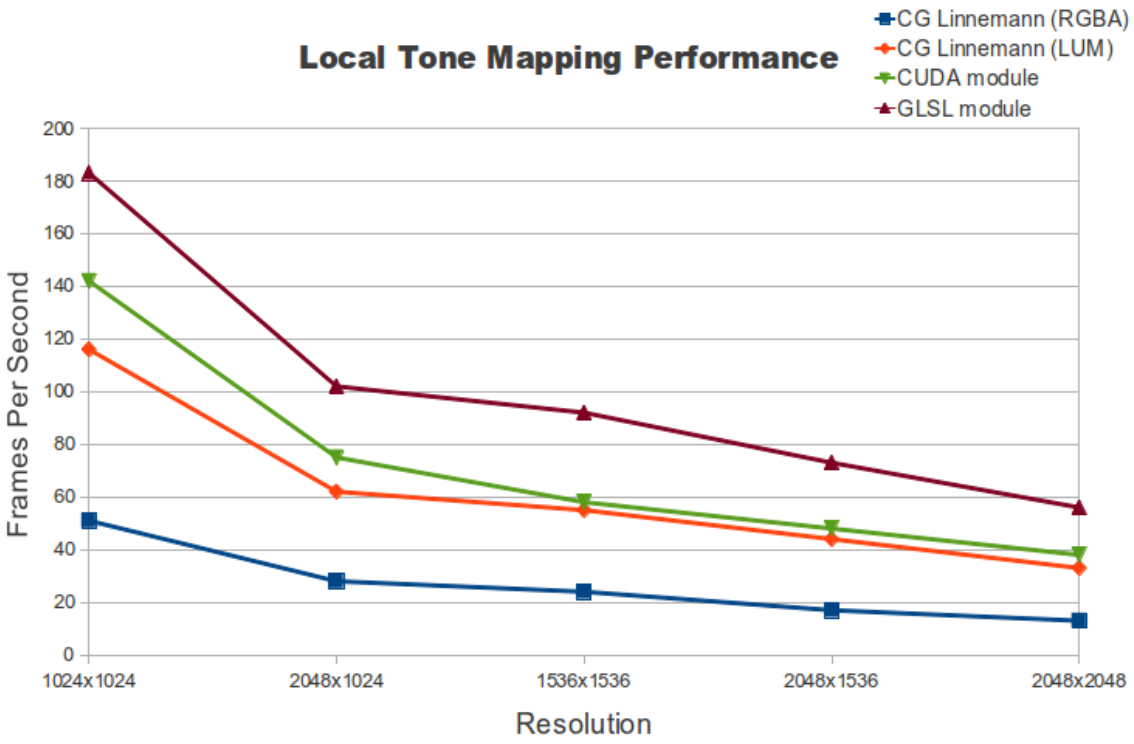


Figure 7-14: Comparison of local tone mapping implementations. Methods denoted "'CG Linnemann (...)"' are implementations of Slomp and Oliveira's original method using OpenGL texture formats specified within brackets. The CUDA and GLSL modules are the implementations developed in this thesis.

Figure 7-14 shows that the texture format modification made to Linnemann's implementation lead to a profound impact on performance. Unlike the original implementation, the modified version is now capable of maintaining interactive frame rates up to the maximum resolution for PC systems specified in Requirement 2.

The CUDA module consistently performs better than all implementations of *Slomp and Oliveira*'s method, but is unable to match its equivalent shader implementation, which significantly outperforms all other implementations. Although the CUDA module is subject

to an overhead involved in mapping between OpenGL and CUDA, it was shown in Section 5.3 that this overhead is too small to have a significant impact on performance. Additionally, Section 7.4.2 shows that SAT generation is accomplished faster in the CUDA module than in its GLSL equivalent. Therefore, it is likely that, despite the optimization efforts in Section 6.1.5, the tone mapping kernel used in the CUDA module is slower than its corresponding shader program in the shader module.

It appears that, despite CUDA's ability to generate SATs more efficiently, the small OpenGL/CUDA mapping overhead; the use of hardware mipmapping, rather than reduction; and a more efficient final, tone mapping shader program make the GLSL shader module a more efficient local tone mapping option.

7.4.4 A hybrid method

The results presented in Section 7.4.2 showed that optimal SAT generation performance is achieved using CUDA. At the same time, Section 7.4.3 discovered that, despite more efficient SAT generation, the entire tone mapping process, when implemented in CUDA, could not match the performance of an equivalent GLSL implementation. The most likely cause for this was identified as the final tone mapping kernel, which was shown in Section 7.2 to be responsible for 71% of the CUDA module processing time.

Motivated by the results of the previous sections, an additional, hybrid GLSL/CUDA tone mapping module was developed. The hybrid module uses the most efficient components of the CUDA and GLSL modules; the majority of the tone mapping process, including the final, box filter pyramid evaluation step, is accomplished using the GLSL shaders, while CUDA is used for SAT generation. Therefore, the hybrid module has an equivalent structure to shader tone mapping module shown Figure 5-19, where the only difference is that the "Generate SAT" submodule is the CUDA implementation used in the CUDA tone mapping module (Figure 5-16).

Figure 7-15 shows the performance of the hybrid method compared to that of the GLSL and CUDA modules. Indeed, for most input resolutions, the hybrid method gives the best results. As the input resolution increases, the frame rates of the hybrid and GLSL modules converge. This is caused by the additional copying necessary to accomodate the CUDA SAT generation submodule within the GLSL tone mapping pipeline: the GLSL pipeline operates solely on textures, while the CUDA SAT generation submodule neither reads nor writes from textures. Thus, when passing data to or retrieving the result from the SAT generation, a full copy of the entire frame from texture memory to global memory or from PBO to texture memory is necessary. Although these copy operations take place within high speed GPU memory, as resolutions increase, they begin to have a measurable effect on performance.

For resolutions up to and beyond the target of 1920x1200 for PC systems, as specified by Requirement 2, the hybrid module delivers excellent results. Compared to its closest competitor, the shader tone mapping module, the hybrid module provides both higher frame rates and, as a result of in-place scan computation enabled by CUDA, a significantly lower memory footprint. Furthermore, for all resolutions tested, the hybrid operator is capbable of matching or exceeding the *desired* frame rate specified by Requirement 2. For an input resolution of 1024x1024, the hybrid operator runs at more than double the

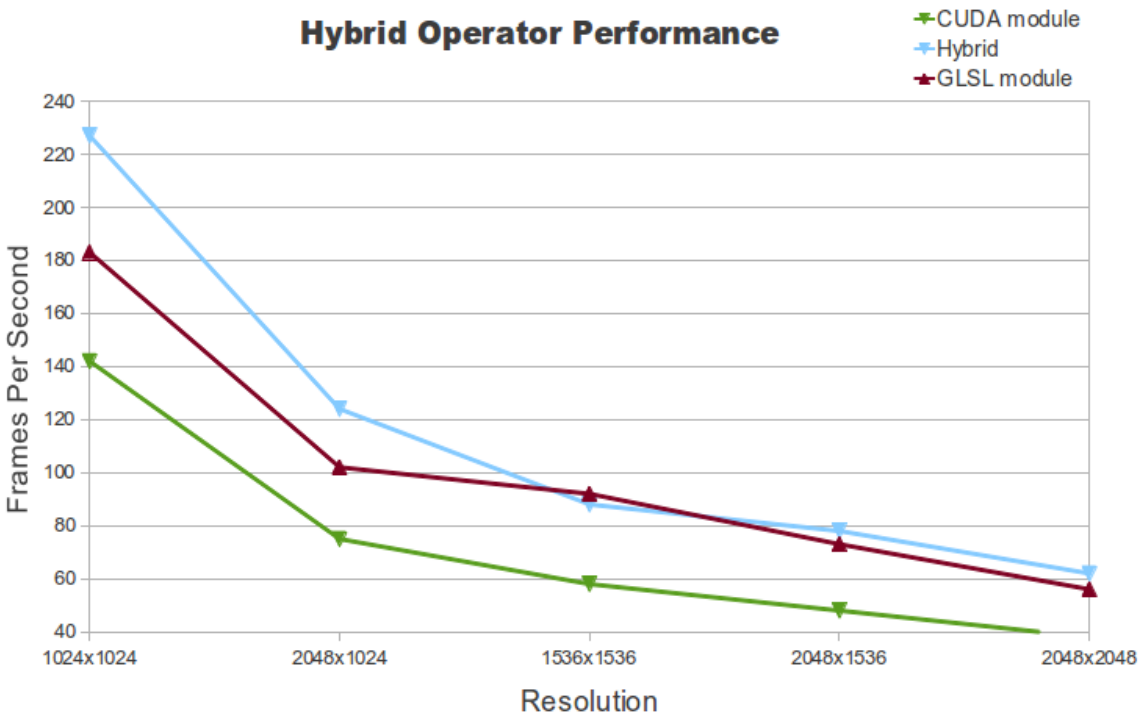


Figure 7-15: Hybrid method performance, compared to that of the CUDA and GLSL modules.

102 fps originally measured by *Slomp and Oliviera* at the same resolution and on the same GPU as our development system (see Section 4.2.3).

7.4.5 High Definition resolutions in the HD Visualization Center

Requirement 2.ii specifies that the tone mapping module should run at interactive frame rates on the HD Visualization Center (detailed in Appendix A), which supports a maximum resolution of 3840x2160. To determine if this requirement has been met, TMStudio was run on the HD Visualization Center to evaluate the performance of the tone mapping methods presented in this thesis. The results are shown in Figure 7-16, measured on a set of HDR scene ranging in resolution from 2048x1024 to 4096x2048.

Surprisingly, Figure 7-16 shows that all methods using CUDA perform at unacceptable frame rates, even for relatively low resolutions. This directly contradicts expectations: as described in Appendix A, the HD Visualization Center is driven by cluster of networked PCs, each equipped with a pair of NVIDIA Quadro FX 5800 GPUs. These GPUs are specifically designed to support CUDA. Considering that the Quadro FX 5800 offers 240 CUDA cores (vs the 128 offered by the GeForce 8800 GTX used in the development system detail in Appendix A) each with a clock rate of 650 MHz (vs the GeForce 8800 GTX's 575MHz per core), a frame buffer of 4GB (vs 756 MB) with memory clock frequency of 1632 MHz (vs 900 MHz) and a compute capability of 1.3 (vs 1.0), significantly better CUDA performance can be expected for equivalent resolutions on the HD Visualization Center than on the development system.

The graphics drivers installed on the HD Visualization Center are controlled by a large set of parameters. If the parameters are not configured optimally for the application being

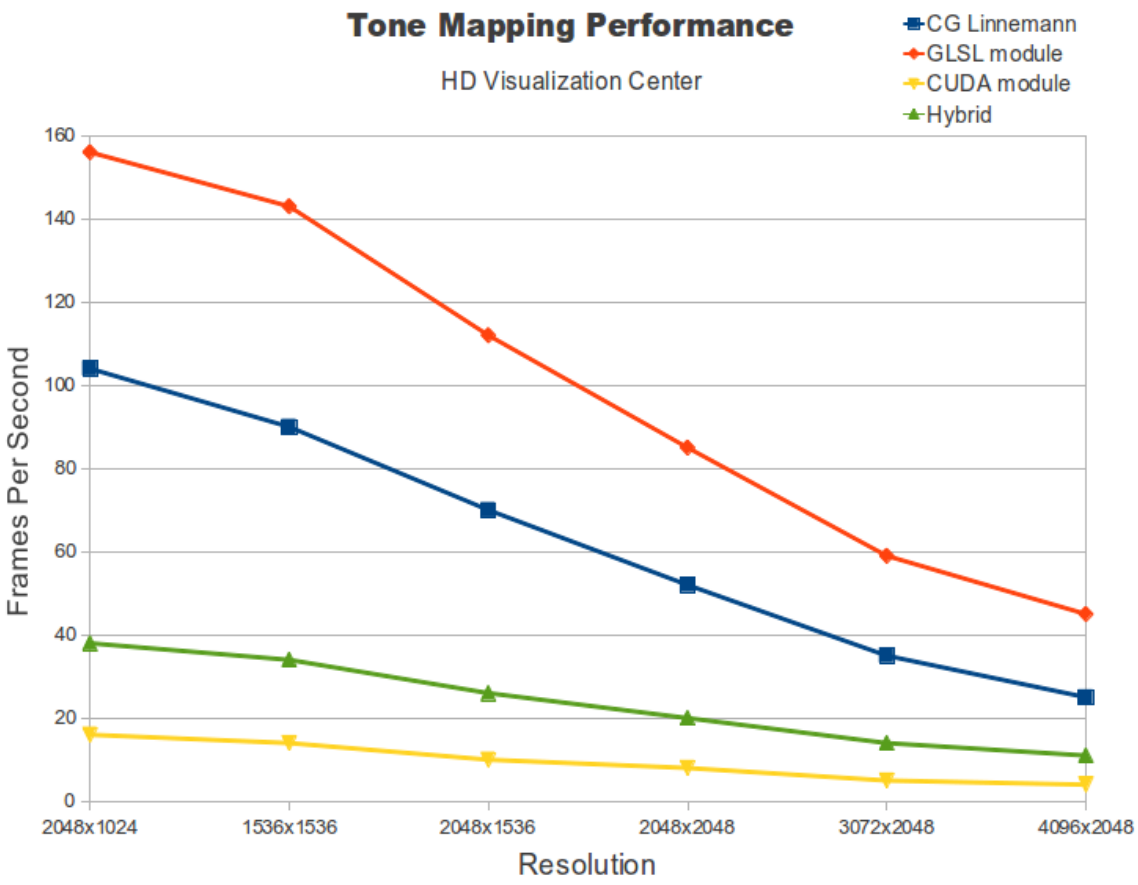


Figure 7-16: Tone mapping performance on the HD Visualization Center.

run, significant degradation of performance can occur. This appears to be the case when running TMStudio with CUDA tone mapping. Since the tone mapping procedure is entireley memory bound (see Section 6.1.5), it is probable that a misconfiguration of GPU driver parameters is causing debilitating copying of frame data from the GPU across the bus to the host system.

Unfortunately, due time constraints, after a number of fruitless attempts at finding the correct driver parameter configuration, correctly deploying CUDA frame processing applications on the HD Visualization Center has been left as future work.

Nonetheless, Figure 7-16 shows that, unlike *Slomp and Oliveira*'s original method (CG Linnemann), the GLSL tone mapping module meets the minimum performance requirements specified by Requirement 2. Perhaps, in future, after some GPU driver parameter tweaking, this accomplishment will be shared by the CUDA-based modules.

7.5 Balancing speed and quality

The performance measurements provided in this chapter were gathered using TMStudio. According to these measurements, all modules developed are capable of meeting the minimim frame rates of 30fps for PC systems called for by Requirement 2, and both the shader and hybrid modules can meet the desired frame rate of 60 fps. Because the OpenGL scene rendered by TMStudio is a very simple one, these performance results are

nearly entirely a measure of the time spent tone mapping.

Unlike TMStudio, VND makes heavy use of the GPU to generate its scene, before any tone mapping is required. Thus, although the tone mapping modules developed in this thesis meet all interactive requirements set forth in Chapter 3, when integrated in VND, it can not be certain that the new, HDR implementation of VND will continue to do so. Due to time constraints, integrating one of the local tone mapping modules into VND remains outside the scope of this thesis and has been left as future work.

As discussed in Chapter 3, Requirement 1, which imposes restrictions on output visual quality, and Requirement 2, which specifies minimum performance expectations, are contradictory; the better the output visual quality, the more computation required. At present, all tone mapping modules have been implemented to preserve the maximum amount of detail possible using *Slomp and Oliveira*'s operator, upon which they are based.

In the case that, once local tone mapping has been integrated, VND is unable to maintain interactive frame rates, additional performance can be attained by limiting the number of box filter scales evaluated during tone mapping, at the expense of output visual quality.

An additional parameter was added to the hybrid tone mapper to control the number of box filter scales to evaluate. Figure 7-17 shows the performance measured on the development system when tone mapping input images of different resolutions using limited scales. The first, 1024x1024 input commonly reported in earlier sections has been omitted in Figure 7-17, since tone mapping this input with all scales results in very high frame rates. Figures 7-18 and 7-19 show the resulting visual output when limiting the scales on a 1024x1024 night driving scene. All scenes in Figures 7-18 and 7-19 were tone mapped using the default local parameters of $\epsilon = 0.025$ and $\phi = 8$ with a global parameter configuration of $\alpha = 1.0$ and $\epsilon = 1.0$.

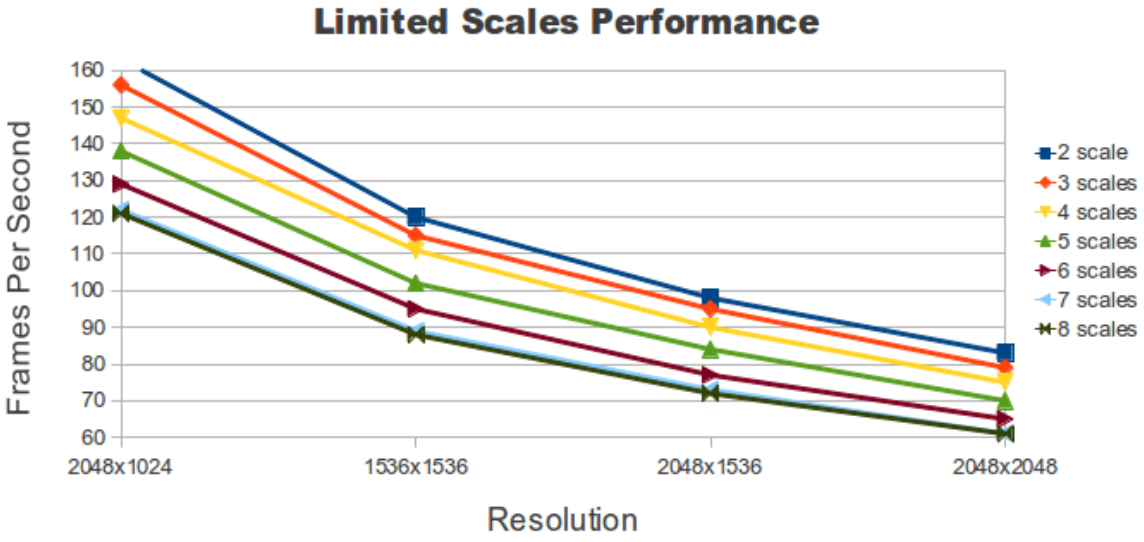
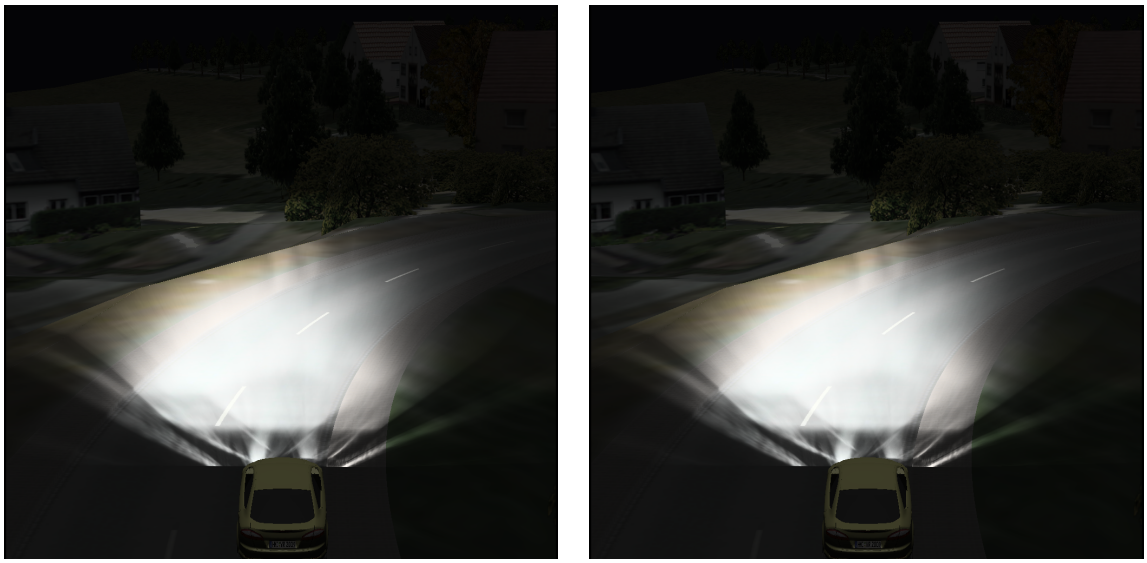
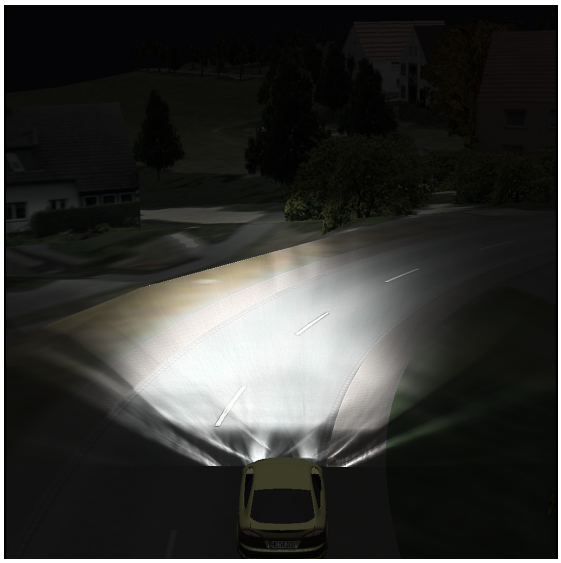


Figure 7-17: The effect of limiting scales used on local tone mapping performance.

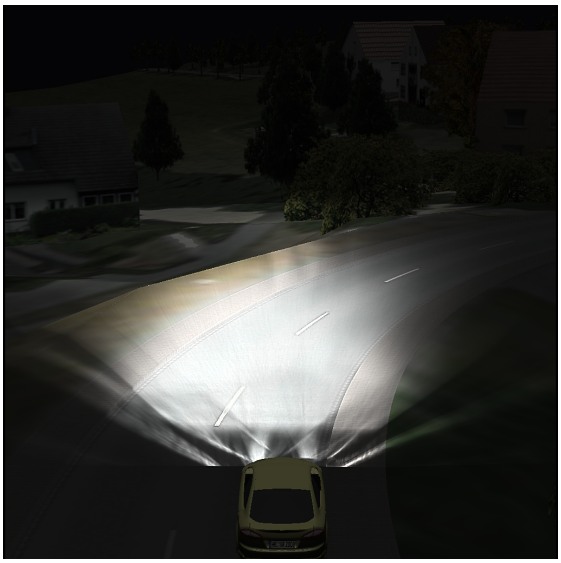


2 scales

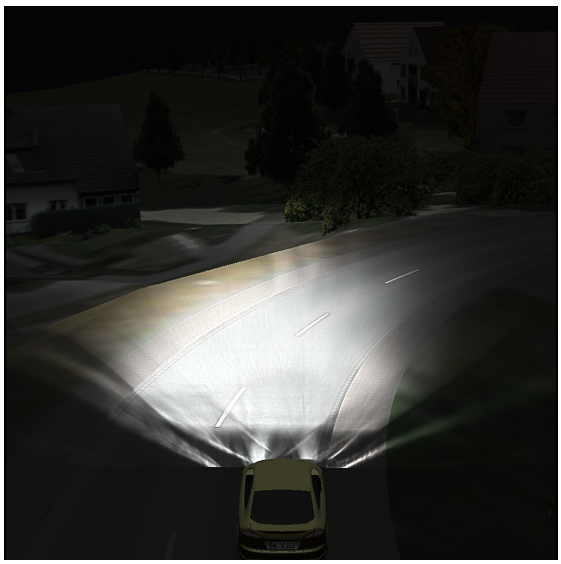
3 scales



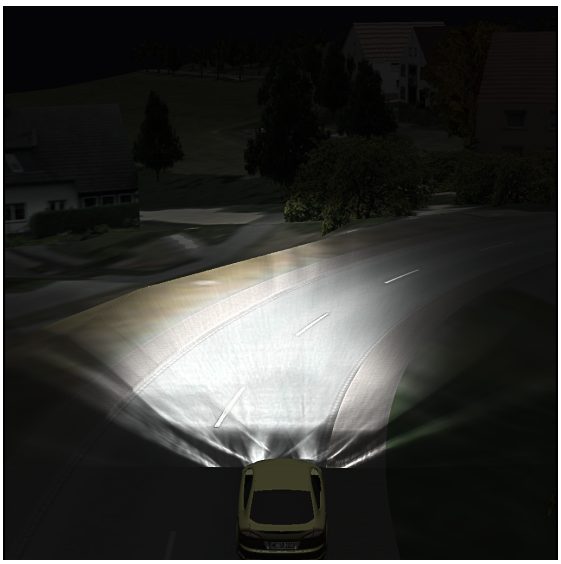
4 scales



5 scales

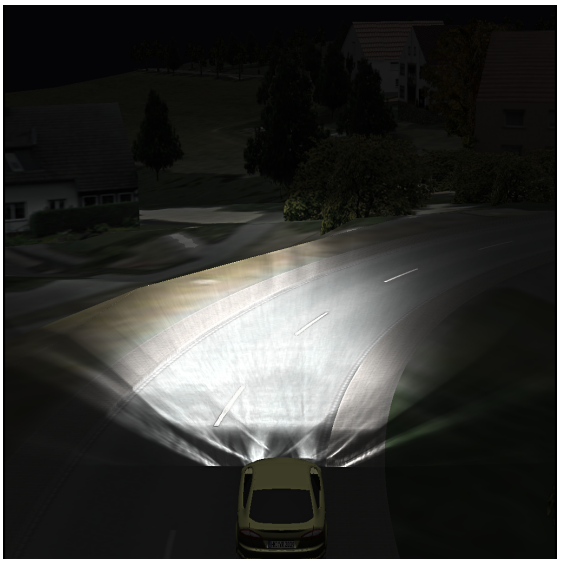


6 scales



7 scales

Figure 7-18: Results of limiting the number of scales used in local tone mapping. The total number of scales evaluated is specifed under each image.



8 scales

Figure 7-19: The same scene as shown in 7-19, tone mapped using a complete, full-scale local procedure.

7.6 Conclusion

This chapter presented the results of the tone mapping methods developed in Chapter 5 and 6.

Although this thesis mainly evaluates results in terms of performance, Section 7.1 began with a qualitative study of the effects of tone mapping implementations on the visual output of VND. The importance of choosing a local operator was argued by comparing locally tone mapped VND scenes with equivalent scenes tone mapped using a global operator, which revealed a noticable difference in beam pattern detail. This was followed by a discussion of each the four parameters that affect visual output, accompanied by a demonstration of their their effect on night driving scenes. For a formal, quantitative evaluation of the visual output of the tone mapping methods used, which are all implementations of *Slomp and Oliveira*'s method, the reader is referred to existing literature [SO08] [LWR+09].

Section 7.2 examined the CUDA tone mapping module developed in Chapter 6. The performance of the module when using implemented with each of the four proposed tone mapping kernels in Section 6.1.5 was measured for a number input sizes. As expected, implementations using the shared memory based tone mapping kernel (Section 6.1.5.5) and the unrolled texture memory based kernel (Section 6.1.5.4) gave the best performance. In addition to overall module performance, the properties of each kernel within the module was discussed, including a presentation of individual execution times and kernel occupancy. In particular, it was discovered that the final, tone mapping kernel, is responsible for 71% of the total module processing time.

To gain an initial insight into the difference between post-processing with CUDA compared with that of shaders, Section 7.3 reported frame rates measured when globally tone

mapping OpenGL applications of various resolutions. Although the shader implementation consistently outperformed its CUDA counterpart, both implementations performed at such high frame rates that the performance discrepancy was attributed to the slight overhead involved in the OpenGL/CUDA context switch.

Section 7.4 presented a comprehensive comparison of the CUDA tone mapping module developed in Chapter 6, the shader tone mapping module, also developed in Chapter 6, and an implementation of *Slomp and Oliveira*'s method, provided by *Linnemann* [LWR+09] and adapted to work with TMStudio. An initial comparison of SAT-generation performance showed that CUDA SAT generation, despite requiring slightly more work than the other methods, provides the best performance, both in terms of speed and memory usage. A subsequent comparison of each tone mapping module, however, showed that the GLSL shader implementation consistently and significantly outperforms both its CUDA counterpart and Linnemann's module. The lacking performance of the CUDA was attributed its final, dodging-and-burning kernel. Therefore, a hybrid method was developed that used the best of both modules: a GLSL tone mapping pipeline, using CUDA for SAT generation. This hybrid method gave excellent results, outperforming the shader module both in terms of speed and memory usage. Finally, the tone mapping experiments were repeated on the HD Visualization Center (Appendix A), showing that the GLSL module meets all requirements set out in Chapter 3, while all implementations using CUDA were impaired by what was presumed to be a suboptimal GPU driver configuration.

Finally, to assist in a possible future need to find a balance between visual quality and tone mapping performance, Section 7.5 presented the effects of limiting the number of scales used during tone mapping, both in terms of performance and visual output.

8 Conclusion and Outlook

This chapter reviews the important accomplishments presented in this thesis (Section 8.1), and outlines ideas for future work (Section 8.2).

8.1 Thesis summary

The goal of this thesis was to develop a tone mapping procedure suitable for use with Virtual Night Drive, a high resolution, Virtual Reality tool for prototyping automotive headlights. To qualify for use with VND, it is necessary that this operator preserves detail in high contrast regions, performs at interactive frame rates (defined as a minimum of 30 fps in Chapter 3) at high resolutions, and be implemented such that it can easily be integrated into the existing version of VND as a post-processing module.

To develop a suitable operator, a state of the art local tone mapping method was optimized using work-efficient parallel scan algorithms. Two GPU implementations of this optimized method were developed: one using CUDA and one using GLSL shaders.

CUDA was chosen for the first implementation because it exposes GPU hardware not available to traditional shader languages; work-efficient parallel scan algorithms operate faster and with less memory overhead when implemented in CUDA than in shader languages. Since all previous work on implementation of high speed local tone mapping operators has been accomplished using shaders, using CUDA represents a novel investigation of the suitability of CUDA for local tone mapping. The second, GLSL implementation was required to validate the CUDA results: since the tone mapping algorithm used is a novel modification of the current state of the art, a GLSL implementation of the same algorithm is necessary to determine whether performance gains can be attributed to the modified algorithm or its implementation in CUDA.

Both implementations perform better than the current state of the art, indicating that the use of work-efficient parallel scan algorithms was a successful optimization. In terms tone mapping speed, GLSL implementation consistently and significantly outperforms its CUDA counterpart. In terms of GPU memory usage, the converse is true. Motivated by these results, as well as an analysis revealing a misalignment of bottlenecks in the respective implementations, a hybrid method using GLSL for image processing and CUDA for general purpose computation was conceived (Section 7.4.4). The hybrid method achieved an optimum of both performance and memory usage.

Despite the promise of the hybrid method, however, at the time of writing, only the GLSL tone mapping implementation can claim to meet all requirements for use with VND. This is because, despite running well on the development system, experiments in the HD Visualization Center (Section A.2), one of VND's primary target platforms, showed that all methods involving CUDA computation, including the hybrid method, operate unaccept-

ably slowly. It is suspected that the cause of this is an incorrect configuration of GPU driver parameters.

Notwithstanding the HD Visualization Center results, the CUDA tone mapping implementation has shown that, unlike earlier generations [LU08], modern CUDA releases are capable of complex post-processing of OpenGL applications at speeds comparable to those achievable with shaders.

8.2 Future work

Integration into VND
The tone mapping modules developed in this thesis were implemented and evaluated within a custom, HDR prototyping platform. After resolving the driver issues in the HD Visualization Center, the most suitable of the tone mapping implementations can be decisivley identified. The next step is to integrate the selected module into VND, and extend VND to use HDR beam pattern data for scene illumination.

Once integrated into VND, to account for temporal brightness discontinuities that arise is dynamic HDR environments [GWW+03] [KMS05], the tone mapping module will need to be extended to support *temporal adaption* [DD00]. Furthermore, to increase the realism of the driving simulation, perceptual effects that commonly occur in the human visual system when driving at night, such as *glare* [WWB+07] and *scotopic vision* [KMS05] should be simulated during the tone mapping process.

HDR video with CUDA
CUDA is commonly used to accelerate the computationally expensive task of video encoding and decoding [War10]. Using the CUDA tone mapping module developed in this thesis, existing CUDA video processing tools can be extented to support high resolution, HDR video.

Automatic tone mapping parameter computation
In all implementations in this thesis, the parameters that control tone mapping are initialized to the default values specified in the Literature and are modified only by manual user intervention via the protoyping platform. As was seen in Section 7.1, the default parameters are not necessarily ideal for night driving scenes. Future research could be untertaken to identify the most suitable parameters for night driving. In particular, it would be useful to identify HDR scene metrics that can be used for automatic parameter adjustment.

9 Bibliography

[Ada80] ADAMS, A., The Camera, The Ansel Adams Photography series, Little, Brown and Company, 1980

[Ada81] ADAMS, A., The Negative, The Ansel Adams Photography series, Little, Brown and Company, 1981

[Ada83] ADAMS, A., The Print, The Ansel Adams Photography series, Little, Brown and Company, 1983

[Ang11] ANGEL, E., Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL, Addison Wesley, 2011

[Ash02] ASHIKHMIN, M., A Tone Mapping Algorithm for High Contrast Images, Proceedings of Thirteenth Eurographics Workshop on Rendering (2002), 2002, P. 145–155

[BER05] BERSSENBRÜGGE, J., Ein Verfahren zur Darstellung der komplexen Lichtverteilungen moderner Scheinwerfersysteme im Rahmen einer virtuellen Nachtfahrt, Dissertation, Fakultät für Maschinenbau, Universität Paderborn, 2005

[Ble90] BLELLOCH, G.E., Prefix Sums and Their Applications., Technical Report, School of Computer Science, Carnegie Mellon University, 1990, technical Report CMU-CS-90-190

[BO04] BURNS, D.; OSFIELD, R., Open Scene Graph A: Introduction, B: Examples and Applications, Proceedings of the IEEE Virtual Reality 2004, IEEE Computer Society, Washington, DC, USA, 2004, P. 265–, unter: http://dl.acm.org/citation.cfm?id=1009389.1010422

[Bov08] BOVIK, A., The Essential Guide to Image Processing, Academic Press, 2008

[Che08] CHEN, H., Lighting and Material of HALO 3, 2008, bungie Publication

[Cro84] CROW, F.C., Summed-area tables for texture mapping, SIGGRAPH Comput. Graph., 18, 1984:P. 207–212, unter: http://doi.acm.org/10.1145/964965.808600

[CWN+08] CADÍK, M.; WIMMER, M.; NEUMANN, L.; ARTUSI, A., Evaluation of HDR tone mapping methods using essential perceptual attributes, Computers & Graphics, 32(3), 2008:P. 330–349

[DCW+02] DEVLIN, K.; CHALMERS, A.; WILKIE, A.; PURGATHOFER, W., Tone Reproduction and Physically Based Spectral Rendering, Eurographics, 2002

[DD00] DURAND, F.; DORSEY, J., Interactive Tone Mapping, Eurographics Workshop on Rendering, 2000, P. 219–230

[DR77] DUBOIS, P.; RODRIGUE, G., High Speed Computer and Algorithm Organization, Academic Press, 1977, P. 299–305

[Fer01] FERWERDA, J., Elements of early vision for computer graphics, Computer Graphics and Applications, IEEE, 21(5), 2001:P. 22–33

[FJ02] FAIRCHILD, M.D.; JOHNSON, G.M., Meet iCAM: An Image Color Appearance Model, IS&T/SID 10th Color Imaging Conference, 2002, P. 33–38

[FLW02] FATTAL, R.; LISCHINSKI, D.; WERMAN, M., Gradient domain high dynamic range compression, Proceedings of ACM SIGGRAPH 2002, ACM SIGGRAPH, 2002

[GC] GREEN, S.; CEBENOYAN, C., High Dynamic Range Rendering on the GeForce 6800, nVIDIA Presentation

[GGN+08] GARLAND, M.; GRAND, S.; NICKOLLS, J.; ANDERSSON, J.; HARDWICK, J.; MORTON, S.; PHILLIPS, E.; ZHANG, Y.; VOLKOV, V., Parallel Computing Experiences with CUDA, IEEE Micro, 2008:P. 13–27

[GV97] GOMES, J.; VELHO, L., Image Processing for Computer Graphics, Springer Berlin, 1997

[GWH05] GOODNIGHT, N.; WANG, R.; HUMPHREYS, G., Projects in VR, IEEE Computer Graphics and Applications, 2005:P. 12–15

[GWW+03] GOODNIGHT, N.; WANG, R.; WOOLLEY, C.; HUMPHREYS, G., Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware, P. CHRISTENSEN; D. COHEN-OR (Editor.), Eurographics Symposium on Rendering 2003, The Eurographics Association, 2003, P. 26–37

[Har08] HARRIS, M., Optimizing Parallel Reduction in CUDA, NVIDIA Developer Technology, 2008, unter: `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`

[HDR] HDRSOFT, HDR images for Photography FAQ, unter: `http://www.hdrsoft.com/resources/dri.html`

[HSC+05] HENSLEY, J.; SCHEUERMANN, T.; COOMBE, G.; SINGH, M.; LASTRA, A., Fast Summed-Area Tabel Generation and its Applications, Proceedings of Eurographics 2005, Dublin, Ireland, 2005, P. 547–555

[HSO07] HARRIS, M.; SENGUPTA, S.; OWENS, J.D., GPU Gems 3, Chapter 39, Addison-Wesley Professional, 2007

[ITU90] ITU-R Recommendation BT.709, Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange., 1990, iTU (International Telecommunication Union), Geneva

[KBH07] KAINZ, F.; BOGART, R.; HESS, D., GPU Gems 3, Chapter 26, Addison-Wesley Professional, 2007

[KMS05] KRAWCZYK, G.; MYSZKOWSKI, K.; SEIDEL, H.P., Perceptual Effects in Real-time Tone Mapping, Proceedings of the 21st Spring Conference on Computer Graphics, 2005, P. 195–202

[LCT+05] LEDDA, P.; CHALMERS, A.; TROSCIANKO, T.; SEETZEN, H., Evaluation of Tone Mapping Operators using a High Dynamic Range Display, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005), Volume 24, 2005, P. 640–648

[LS98] LARSON, G.W.; SHAKESPEARE, R., Rendering with radiance: the art and science of lighting visualization, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998

[LU08] LÖNROTH, P.; UNGER, M., Advanced Real-time Post-Processing using GPGPU techniques, Master's thesis, Department of Science and Technology Linköping University, SE-601 74 Norrköping, Sweden, 2008

[LWR+09] LINNEMANN, M.; WASSMANN, H.; RADKOWSKI, R.; SZWILLUS, G.; DOMIK, G., Entwicklung eines Tone Mapping- Verfahrens für die Lichtsimulation in Echtzeit-Augmented Reality- Anwendungen, Master's thesis, University of Paderborn, 2009

[McT06] MCTAGGART, G., HDR in Valves Source Engine, 2006, presentation at SIGGRAPH 06

[Mic10] MICIKEVICIUS, P., Analysis-Driven Optimization, GTC, 2010

[MMS04] MANTIUK, R.; MYSZKOWSKI, K.; SEIDEL, H.P., Visible Difference Predicator for High Dynamic Range Images, Proceedings of IEEE International Conference on Systems, Man and Cybernetics, 2004, P. 2763–2769

[Nie05] NIELSEN, F., Visual Computing: Geometry, Graphics, and Vision, Charles River Media / Thomson Delmar Learning, 2005, unter: `http://www.sonycsl.co.jp/person/nielsen/visualcomputing`

[NVI11] NVIDIA Corporation, NVIDIA CUDA C Programming Guide, 2011

[PB10] PETIT, J.; BREMOND, R., A high dynamic range rendering pipeline for interactive applications, The Visual Computer: International Journal of Computer Graphics, Volume 26, 2010:P. 533–542

[PFF+99] PATTANAIK, S.N.; FERWERDA, J.A.; FAIRCHILD, M.D.; GREENBERG, D., A Multiscale Model of Adaptiona nd Spatial Vision for Realistic Image Display, ACM SIGGRAPH, 1999

[Pod07] PODLOZHNYUK, V., Image Convolution with CUDA, NVIDIA Whitepaper, 2007, unter: `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/convolutionSeparable/doc/convolutionSeparable.pdf`

[PY02] PATTANAIK, S.N.; YEE, H., Adaptive Gain Control for High Dynamic Range Imaging, Proceedings of Spring Conference in Computer Graphics (SCCG2002), 2002, P. 24–27

[RF93] RAU, B.R.; FISCHER, J.A., Instruction Level Paralellism, A Special Issue of The Journal of Supercomputing, Springer, 1993

[RGR+06] RAFAŁ MANTIUK, A.; GRZEGORZ KRAWCZYK, A.; RADOSLAW MANTIUK, B.; HANS-PETER SEIDEL, A., High Dynamic Range Imaging Pipeline: Perception-Motivated Representation of Visual Content, Forschung und wissenschaftliches Rechnen, 2006:P. 11–27

[RM10] RUETSCH, G.; MICIKEVICIUS, P., Optimizing Matrix Transpose in CUDA, NVIDIA Developer Technology, 2010

[RSS+02] REINHARD, E.; STARK, M.; SHERLEY, P.; FERWERDA, J., Photographic Tone Reproduction for Digital Images, Proceedings of ACM Siggraph, Volume 21, 2002, P. 267–276

[RWP+06] REINHARD, E.; WARD, G.; PATTANAIK, S.; DEBEVEC, P., High Dynamic Range Imaging - Acquisition, Display and Image-Based Lighting, Morgan Kaufmann, 2006

[SHG08] SENGUPTA, S.; HARRIS, M.; GARLAND, M., M.: Efficient parallel scan algorithms for GPUs. NVIDIA, Technical Report, 2008

[SHG09] SATISH, N.; HARRIS, M.; GARLAND, M., Designing efficient sorting algorithms for manycore GPUs, Parallel and Distributed Processing Symposium, International, 0, 2009:P. 1–10

[SHZ+07] SENGUPTA, S.; HARRIS, M.; ZHANG, Y.; OWENS, J.D., Scan Primitives for GPU Computing, Graphics Hardware 2007, ACM, 2007, P. 97–106

[SLO06] SENGUPTA, S.; LEFOHN, A.E.; OWENS, J.D., A Work-Efficient Step-Efficient Prefix Sum Algorithm., Proceedings of Workshop on Edge Computing Using New Commodity Architectures, 2006, P. 26–27

[SO08] SLOMP, M.; OLIVEIRA, M.M., Real-time photographic local tone reproduction using Summed-Area Tables, Computer Graphics International 200, 2008, P. 82–91

[Tön05] TÖNNIES, K.D., Grundlagen der Bildverarbeitung, Pearson Studium, 2005

[TT99] TUMBLIN, J.; TURK, G., LCIS: a boundary hierarchy for detail-preserving contrast reduction, Proceedings of the 26th annual conference on Computer graphics and interactive techniques, SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999, P. 83–90, unter: `http://dx.doi.org/10.1145/311535.311544`

[TW08] TZENG, S.; WEI, L.Y., Parallel white noise generation on a GPU via cryptographic hash, Proceedings of the 2008 symposium on Interactive 3D graphics and games, I3D '08, ACM, New York, NY, USA, 2008, P. 79–87, unter: `http://doi.acm.org/10.1145/1342250.1342263`

[Vis10] Compute Visual Profiler User Guide, NVIDIA Whitepaper, 2010, supplied with CUDA 3.2 SDK

[Wan95] WANDELL, B.A., Foundations of Vision, Sinauer Associates Inc, 1995

[War10] WARREN, S., VDPAU: PureVideo on Unix, 2010, unter: `http://www.gputechconf.com/page/gtc-on-demand.html#session2011`, presentation at NVIDIA GTC2010

[WWB+07] WÖRDENWEBER, B.; WALLASCHEK, J.; BOYCE, P.; HOFFMANN, D., Automotive Lighting and Human Vision, Springer-Verlag Berlin Heidelberg, 2007

[YNC+06] YUAN, X.; NGUYEN, M.X.; CHEN, B.; PORTER, D.H., HDR VolVis: High Dynamic Range Volume Visualization, IEEE Transactions on Visualization and Computer Graphics, 12, 2006:P. 433–445

[ZW97] ZHANG, X.; WANDELL, B., A spatial extension of CIELAB for digital color image reproduction, SID Journal, 5, 1997, P. 61–63

A Tools and Equipment

This appendix details the tools and equipment used to support the development of this thesis.

Section A.1 specifies, in detail, the setup of the PC system used for software development and the majority of results gathering.

In addition to standard PC platforms, the tone mapping software developed in this thesis has been designed to run on a custom Virtual Reality system at the Heinz Nixdorf Institute, known as the *High Definition (HD) Visualization Center*. This system is detailed in Section A.2.

Finally, throughout this thesis, a number of HDR scenes were used to test and demonstrate tone mapping. All scenes that were not extracted from VND were created by a various researchers active in HDR rendering, who made their scenes available for public use. Section displays (a tone mapped version) of each HDR scene used, and specifies its source.

A.1 The development system

The system used for development and the majority of results gathering throughout this thesis is referred to as the *development system*.

The development system is a desktop PC running the Ubuntu 11.04 Operating System, with an Intel Xenon 2.33 GHz Quadcore CPU with 2GB main memory and an NVIDIA GeForce 8800 GTX GPU.

Since the modules developed in this thesis are specifically designed to run on the GPU, Figure 1-1 lists the important details of NVIDIA GeForce 8800 GTX GPU hardware.

Property	**Value**
Memory	768 MB
Memory clock speed	900 MHz
Memory bandwidth	86.4 GB/s
CUDA cores	128
CUDA core clock speed	575 MHz
Compute capability	1.0

Figure 1-1: NVIDIA GeForce 8800 GTX hardware properties.

A.2 The HD Visualization Center

The HD Visualization Center is an immersive, VR simulation system at the Heinz Nixdorf Institute. It consists of front, side and floor projection surfaces, a surround sound system and head tracking harware. The projection set up is shown in Figure 1-2.

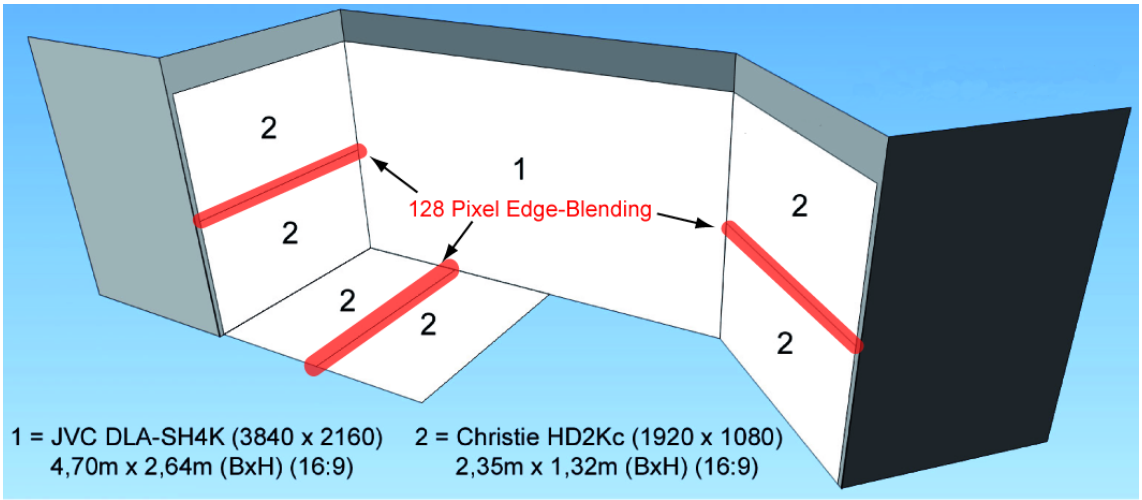


Figure 1-2: HD Visualization projection setup.

The scene rendered on each of the projection surfaces in Figure 1-2 is generated by a system of projectors, which are driven by a cluster of networked PCs. Each PC in the cluster drives an individual NVIDA QuadroPlex system, which contains two NVIDIA GeForce Quadro FX 5800 GPUs connected via Scalable Link Interface (SLI).

In the context of this thesis, the most interesting aspect of the HD Visualization Center is the GPU hardware used. Therefore, Figure 1-3 lists the relevant hardware properties of the NVIDIA Quadro FX 5800 GPU.

Property	**Value**
Memory	4 GB
Memory clock speed	1632 MHz
Memory bandwidth	102 GB/s
CUDA cores	240
CUDA core clock speed	650 MHz
Compute capability	1.0

Figure 1-3: NVIDIA Quadro 5800 FX hardware properties.

A powerful tool for Virtual Prototyping, the HD Visualization Center is one of the main platforms for which VND is designed. Figure 1-4 shows VND running in this immersive environment.

Figure 1-4: VND running in the HD Visualization Center

A.3 HDR images

Throughout this thesis, a number of HDR scenes supplied by third parties were used. This section shows each scene used, designating it a name and specifying its source. It is intended that this section be used an “image bibliography”. The remaining figures in this thesis each detail a HDR scene used in this thesis.

“MPI Atrium”	
Creator:	Rafal Mantiuk
Location:	www.mpi-inf.mpg.de/resources/hdr/gallery.html
Dynamic range:	1 : 2 877

“Snow”	
Creator:	Rafal Mantiuk
Location:	www.mpi-inf.mpg.de/resources/hdr/gallery.html
Dynamic range:	1 : 601

“Iwate”	
Creator:	Frédéric Drago
Location:	www.mpi-inf.mpg.de/resources/hdr/gallery.html
Dynamic range:	1 : 725

“Vine Sunset”	
Creator:	Paul E. Debevec
Location:	http://ict.debevec.org/~debevec/Research/HDR/
Dynamic range:	1 : 5 868

“Memorial Church”	
Creator:	Paul E. Debevec
Location:	http://ict.debevec.org/~debevec/Research/HDR/
Dynamic range:	1 : 342 590

“Napa Valley”	
Creator:	Spheron AG
Location:	http://www.mpi-inf.mpg.de/resources/tmo/NewExperiment/TmoOverview.html
Dynamic range:	1 : 67 780

B Source Code

This appendix lists the CUDA (Section B.1) and GLSL (Section B.2) code used by the tone mapping modules developed in the thesis.

B.1 CUDA

This section lists the CUDA and C++ code used to implement the five module kernels or submodules shown in Figure 5-16 in Chapter 5.

B.1.1 Luminance computation

A luminance map is generated from a full precision, 32-bit-per-channel RGBA HDR input texture by calling the `computeLuminance` kernel shown in Listing B.1.

```
__global__ void computeLuminance(float* lumMap,
                                 int lumMapPitch)
{
  // map current thread to a texel in the input texture
  int tx = threadIdx.x; // thread x position within block
  int ty = threadIdx.y; // thread y position within block
  int bx = blockIdx.x;  // block x position within grid
  int by = blockIdx.y;  // block y position within grid
  int bw = blockDim.x;  // block width
  int bh = blockDim.y;  // block height
  int x = bx*bw + tx;   // map thread x to texture x
  int y = by*bh + ty;   // map thread y to texture y

  // get current pixel value
  float4 res = tex2D(inTex, x, y);

  // compute luminance
  float lum = RGB2LUM(res.x, res.y, res.z);

  // compute linear output index
  // note: we are using pitch-linear memory
  int i = y*lumMapPitch+x;

  // save logarithm of luminance in the luminance map
  lumMap[i] = log(lum+1.0);
}
```

Listing B.1: CUDA code for luminance computation kernel.

Line 18 in Listing B.1 uses the macro `RGB2LUM(...)`, which is defined in Listing B.2.

```
1 #define RGB2LUM(r,g,b) (r * 0.2125 + g * 0.7154 + b * 0.0721)
```

Listing B.2: Macro for computing luminance from RGB input.

B.1.2 Reduction

The kernel used for array reduction is given in Listing B.3.

```
1
2  template <unsigned int blockWidth>
3  __global__ void reduce( float *idata,
4                          float *odata,
5                          int num_remaining,
6                          int imgw) {
7
8    // shared memory cache
9    // for computation of partial result
10   __shared__ float cache[threadsPerBlock];
11
12   // element in global array to load
13   int tid = threadIdx.x + blockIdx.x *blockWidth*2;
14
15   // index within shared memory cache that this
16   // thread will work on
17   int cacheIndex = threadIdx.x;
18
19   // compute first sum during load
20   float firstSum = 0;
21   firstSum = (tid < num_remaining) ? idata[tid] : 0;
22   firstSum += (tid+blockWidth < num_remaining)?
23                       idata[tid+blockWidth]:
24                       0;
25
26   // load the first pairwise sum into cache
27   cache[cacheIndex] = firstSum;
28   __syncthreads();
29
30   // reduce manually (no loops)
31   if (blockWidth >= 512) {
32       if(cacheIndex < 256)
33           cache[cacheIndex] += cache[cacheIndex + 256];
34       __syncthreads();
35   }
36   if (blockWidth >= 256) {
37       if(cacheIndex < 128)
38           cache[cacheIndex] += cache[cacheIndex + 128];
39       __syncthreads();
40   }
41   if (blockWidth >= 128) {
```

```
42          if(cacheIndex < 64)
43              cache[cacheIndex] += cache[cacheIndex + 64];
44          __syncthreads();
45      }
46
47      // once i <= 32 we are in the same warp.
48      // No need to call __syncthreads anymore
49      if (cacheIndex < 32)
50      {
51          volatile float *v = cache;
52          v[cacheIndex] += v[cacheIndex + 32];
53          v[cacheIndex] += v[cacheIndex + 16];
54          v[cacheIndex] += v[cacheIndex + 8];
55          v[cacheIndex] += v[cacheIndex + 4];
56          v[cacheIndex] += v[cacheIndex + 2];
57          v[cacheIndex] += v[cacheIndex + 1];
58      }
59
60      // write the result to global memory
61      if (cacheIndex == 0)
62          odata[blockIdx.x] = cache[0];
63  }
```

Listing B.3: Reduction kernel.

As discussed in Section 5.4.2, the reduction kernel does not completely reduce an input array to a single sum. Instead, each invocation of the reduction kernel divides the input array into segments, reduces these segments within shared memory, and writes the results back into global memory. To completely reduce an array of n elements, the `reduction` kernel must be repeatedly called $\lceil log_2(n) \rceil$ times.

The CPU-side C code for repeatedly invoking the `reduction` code to reduce a luminance map is given in Listing B.4 and uses the result to compute the scene key (defined in Section 4.1.1).

```
1   int i = 0;
2   float remaining = lumMapPitch*height;
3   while(remaining > 1){
4
5       // invoke the reduction kernel with sufficient threads
6       // to process the remaining partial sums
7       int nBlocks = ceil((remaining/(float)threadsPerBlock)*0.5);
8       reduce<threadsPerBlock><<< nBlocks, threadsPerBlock >>>
9         (
10          (i==0)?lumMap:reductionBuffer,
11          reductionBuffer,
12          (int)remaining,
13          width
14        );
15
16      // update number of remaining partial results
17      remaining = nBlocks;
```

```
18        i++;
19    }
20
21    // copy front element of GPU reductionBuffer to CPU
22    cudaMemcpy(&sum, (const void*)reductionBuffer,
23               sizeof(float), cudaMemcpyDeviceToHost);
24
25    // compute scene key from result
26    float avg = exp(sum/(width*height));
```

Listing B.4: Using the reduction kernel to compute the scene key.

B.1.3 Scaled luminance computation

The kernel for computing scaled luminance, given a luminance map and scene key, is shown in Listing B.5.

```
1
2  __global__ void scaleLuminance(float* sLumMap,
3                                 float* lumMap,
4                                 float key,
5                                 float alpha,
6                                 int pitch)
7  {
8
9      // map current thread to an element in- and output arrays
10     int tx = threadIdx.x;    // thread x pos within thread block
11     int ty = threadIdx.y;    // thread y pos within thread block
12     int bw = blockDim.x;     // block width
13     int bh = blockDim.y;     // block height
14     int x = blockIdx.x*bw + tx; // thread x position in dataset
15     int y = blockIdx.y*bh + ty; // thread y position in dataset
16
17     // compute index to modify in global arrays
18     int i = y*pitch+x;
19
20     // compute scaled
21     sLumMap[i] = exp(lumMap[i])*(alpha/key);
22 }
```

Listing B.5: Scaled luminance computation kernel.

B.1.4 SAT generation

As described in Section 6.1.4, SAT generation is accomplished using a C++ class, `SATHarris`, which uses routines from the CUDA library CUDPP to assist in generating SATs. Listing B.6 shows the two most important methods of the `SATHarris` class: `init` and `compute`.

```
void SATCUDA::init()
{
  // allocate SAT for first set of row scans
  CUDA_SAFE_CALL( cudaMallocPitch( (void**) &_sat[0],
                                   &_outRowPitchInBytes,
                                   _w*sizeof(float),
                                   _h));

  // allocate SAT for second set of row scans
  // (post-transpose)
  //
  // note that _sat[1] has transposed dimension of _sat[0]
  // so we use the transposed dimension variables _ow and _oh
  CUDA_SAFE_CALL( cudaMallocPitch( (void**) &_sat[1],
                                   &_outRowPitchInBytesT,
                                   _ow*sizeof(float),
                                   _oh));

  // fill both SAT dataset with 0
  CUDA_SAFE_CALL( cudaMemset2D(_sat[0], _outRowPitchInBytes, 0,
                               _outRowPitchInBytes, _h));
  CUDA_SAFE_CALL( cudaMemset2D(_sat[1], _outRowPitchInBytesT, 0,
                               _outRowPitchInBytesT, _oh));


  // get row pitch in elements
  _outRowPitchInElements = _outRowPitchInBytes/sizeof(float);
  _outRowPitchInElementsT = _outRowPitchInBytesT/sizeof(float);
  _inRowPitchInElements = _inRowPitchInBytes/sizeof(float);

  // set up CUDPP configuration for (inclusive) scan.
  CUDPPConfiguration config;
  config.op = CUDPP_ADD;
  config.datatype = CUDPP_FLOAT;
  config.algorithm = CUDPP_SCAN;
  config.options = CUDPP_OPTION_FORWARD | CUDPP_OPTION_INCLUSIVE;

  // create a CUDPP plan object for the first set of row scans
  CUDPPResult result = cudppPlan(&_scanPlan1, config, _w, _h,
                                 _inRowPitchInElements);
  if (CUDPP_SUCCESS != result)
  {
      LOG("Error creating CUDPPPlan 1");
      exit(-1);
  }

  // create a CUDPP plan object for the second set of row scans
  result = cudppPlan(&_scanPlan2, config, _ow, _oh,
                     _outRowPitchInElementsT);
  if (CUDPP_SUCCESS != result)
  {
```

```
52        LOG("Error creating CUDPPPlan 2");
53        exit(-1);
54    }
55
56    _initialized = true;
57
58    //_ow = _outRowPitchInElements;
59
60    // create the timers
61    CUT_SAFE_CALL(cutCreateTimer(&_scanTimer));
62    CUT_SAFE_CALL(cutCreateTimer(&_transposeTimer));
63
64 } // end SATHarris::init()
65
66 // ...
67
68 void SATCUDA::compute()
69 {
70    if (!_input){
71        LOG("SATCUDA::compute(): input has not been set!");
72        return;
73    }
74    if (!_initialized){
75        LOG("SATCUDA::compute(): SAT has not been initilized!");
76        return;
77    }
78
79    // set the in/out buffer indices
80    _in = 0;
81    _out = 1;
82    _ow = _h;
83    _oh = _w;
84
85    _scanTime = 0;
86    _transposeTime = 0;
87
88    CUT_SAFE_CALL(cutResetTimer(_scanTimer));
89    CUT_SAFE_CALL(cutResetTimer(_transposeTimer));
90
91    // first set of row scans
92    {
93      CUT_SAFE_CALL(cutStartTimer(_scanTimer));
94
95      // Run the scan
96      cudppMultiScan(_scanPlan1, _sat[_in], _input, _w, _h);
97      CUDA_SAFE_CALL( cudaThreadSynchronize());
98
99      CUT_SAFE_CALL(cutStopTimer(_scanTimer));
100   }
101
102   // transpose
```

```
103   {
104     CUT_SAFE_CALL(cutStartTimer(_transposeTimer));
105
106     // now transpose
107     SAT_transpose_fast(_sat[_out], _sat[_in],
108                        _outRowPitchInElementsT,
109                        _outRowPitchInElements,
110                        _w, _h);
111     CUDA_SAFE_CALL( cudaThreadSynchronize());
112
113     CUT_SAFE_CALL(cutStopTimer(_transposeTimer));
114   }
115
116   // second set of row scans
117   {
118     CUT_SAFE_CALL(cutStartTimer(_scanTimer));
119
120     // result may either be written directly to a dedicated PBO
121     if(_outPBO)
122       cudppMultiScan(_scanPlan2, _outPBO, _sat[_out],
123                      _ow, _oh);
124     // or to the dataset _sat[1]
125     else
126       cudppMultiScan(_scanPlan2, _sat[_out], _sat[_out],
127                      _ow, _oh);
128
129     CUDA_SAFE_CALL( cudaThreadSynchronize());
130
131     CUT_SAFE_CALL(cutStopTimer(_scanTimer));
132   }
133
134   // compute total scan and transpose times
135   _scanTime = cutGetTimerValue(_scanTimer);
136   _transposeTime = cutGetTimerValue(_transposeTimer);
137
138 }
```

Listing B.6: SAT generation C++ code.

The transpose function called on Line 13 in Listing B.6 interfaces an efficient, shared memory based transpose kernel supplied with the CUDA 3.2 SDK.

B.1.5 Dodging-and-burning

Section 6.1.5 proposed four implementations of the dodging-and-burning kernel. The two most efficient implementations used texture memory and shared memory for SAT access, both with unrolled loops.

The dodging-and-burning kernel using texture memory for SAT access is shown in Listing B.7.

```
__global__ void dodgeAndBurn_TexUnrolled(uchar4* out_data,
                                         float* lumMap,
                                         float* sLumMap,
                                         int mapPitch,
                                         float alpha,
                                         float phi,
                                         float epsilon,
                                         float gamma,
                                         int imgw)
{

  float scales[8] = {1.0, 3.0,5.0,7.0,11.0,17.0,25.0,31.0};

  // map thread to (x,y) coordinate
  int bx = blockDim.x*blockIdx.x;
  int by = blockDim.y*blockIdx.y;
  int x = bx + threadIdx.x;
  int y = by + threadIdx.y;

  // compute global array in and out indices
  int mapIndex = y*mapPitch + x;
  int i = y*imgw + x;

  // retrieve input HDR pixel,
  // luminance and scaled luminance
  float4 res = tex2D(inTex, x, y);
  float lum = exp(lumMap[mapIndex]);
  float ls = sLumMap[mapIndex];

  // init adjacent levels of box filter pyramid
  float center = ls;
  float surround = getAverageTex(y, x, scales[1]);

  float w;
  COMPUTE_W(1);

  // proceed with dodging-and-burning
  if (fabs(w) < epsilon){
      center = surround;
      surround = getAverageTex(y, x, scales[2]);
      COMPUTE_W(scales[1]);
  }
  if (fabs(w) < epsilon){
      center = surround;
      surround = getAverageTex(y, x, scales[3]);
      COMPUTE_W(scales[2]);
  }
  if (fabs(w) < epsilon){
      center = surround;
      surround = getAverageTex(y, x, scales[4]);
      COMPUTE_W(scales[3]);
```

```
52    }
53    if (fabs(w) < epsilon){
54         center = surround;
55         surround = getAverageTex(y, x, scales[5]);
56        COMPUTE_W(scales[4]);
57    }
58    if (fabs(w) < epsilon){
59         center = surround;
60         surround = getAverageTex(y, x, scales[6]);
61        COMPUTE_W(scales[5]);
62    }
63         if (fabs(w) < epsilon){
64         center = surround;
65    }
66
67    // now do local tone mapping
68    float ldr = (ls/(1+center))*255;
69
70    // write tone mapped RGB values to output PBO
71    out_data[i] = make_uchar4(
72                  min(ldr*__powf(res.x/lum, gamma), 255.0f),
73                  min(ldr*__powf(res.y/lum, gamma), 255.0f),
74                  min(ldr*__powf(res.z/lum, gamma), 255.0f),
75                  255);
76 }
```

Listing B.7: Dodging-and-burning kernel using texture memory for SAT access.

The `COMPUTE_W()` macro first used on Line 35 in Listing B.7 is used to hide arithmetic clutter required for computing the center-surround function defined in Section 4.1.2, as well to simplify kernel-wide modification whenever this function is changed. Its definitions is given in Listing B.8.

```
1 #define COMPUTE_W(sc) (w = (center-surround) / \
2                       (__powf(2.0, phi)*(alpha/((sc)*(sc))) \
3                        + center))
```

Listing B.8: Center-surround function computation.

Another important function used in Listing B.7 is `getAverageTex`, first used on Line 32. This function uses the SAT texture to compute the average scaled luminance within a square region surrounding a given position. Its definition is given in Listing B.9. Note that trasposed coordinates are used whenever it is called from Listing B.7.

```
1 __device__ float getAverageTex(int x, int y, float size)
2 {
3    // compute distance in each direction from center (x,y)
4    int offset=(int)(0.5*size);
5
6    // for storing result
7    float result = 0.0;
8
9    // compute integral of scaled luminance within
```

```
10    // specified region
11    result = tex2D(satTex, x+offset,y+offset);
12    result -= tex2D(satTex, x+offset, y-offset-1);
13    result -= tex2D(satTex, x-offset-1, y+offset);
14    result += tex2D(satTex, x-offset-1, y-offset-1);
15
16    // return average scaled luminance
17    return result/(size*size);
18 }
```

Listing B.9: A function that accesses a SAT stored in texture memroy to computing the average scaled luminance in a square region surrounding a given location.

The shared memory dodging-and-burning implementation is displayed in Listing B.10

```
1  __global__ void dodgeAndBurn_SM_Unrolled(uchar4* out_data,
2                                           float lavg,
3                                           float alpha,
4                                           float phi,
5                                           float epsilon,
6                                           float gamma,
7                                           int imgw)
8  {
9     // shared memory cache
10    // column allocated with 48+1 elements
11    // to reduce shared memory bank conflicts
12    __shared__ float cache[48][48+1];
13    float scales[8] = {1.0, 3.0,5.0,7.0,11.0,17.0,25.0,33.0};
14
15    // map thread to 2D image pixel pos
16    int bx = blockDim.x*blockIdx.x;
17    int by = blockDim.y*blockIdx.y;
18    int tx = threadIdx.x;
19    int ty = threadIdx.y;
20    int bw = blockDim.x;
21    int bh = blockDim.y;
22    int x = bx + threadIdx.x;
23    int y = by + threadIdx.y;
24
25    // compute input sat element position
26    int satx = by + threadIdx.x;
27    int saty = bx + threadIdx.y;
28
29    // position of current pixel inside SM cache
30    int xc = tx+bw;
31    int yc = ty+bh;
32
33    int i = y*imgw + x;
34
35    // retrieve input HDR pixel,
36    // luminance and scaled luminance
37    //
```

```
38  // note: instead of reading from luminance and
39  //  scaled luminance maps, we recompute the appropriate values.
40  //  This has little impact on performance and reduces
41  //  kernel register usage.
42  float4 res = tex2D(inTex, x, y);
43  float lum = RGB2LUM(res.x, res.y, res.z);
44  float ls = lum*(alpha/lavg);
45
46  // fill apron
47  CACHE(ty+bw, tx+bw) = SATTEX (satx, saty);
48  CACHE(ty+bw, tx) = SATTEX(satx-bw, saty);
49  CACHE(ty+bw, tx+2*bh) = SATTEX(satx+bw, saty);
50
51  CACHE(ty, tx) = SATTEX( satx-bw, saty-bw );
52  CACHE(ty, tx+bh) = SATTEX( satx, saty-bw );
53  CACHE(ty, tx+2*bh) = SATTEX( satx+bw, saty-bw );
54
55  CACHE(ty+2*bw, tx) = SATTEX(satx-bw, saty+bw);
56  CACHE(ty+2*bw, tx+bh) = SATTEX(satx, saty+bw);
57  CACHE(ty+2*bw, tx+2*bh) = SATTEX(satx+bw, saty+bw);
58
59  // barrier: move on only once all thread have loaded their
60  // part of the apron
61  __syncthreads();
62
63  // init adjacent levels of box filter pyramid
64  float center = ls;
65  float surround = BOX(xc,yc,scales[1]);
66
67  float w;
68  COMPUTE_W(1.0);
69
70  // proceed with dodging-and-burning
71  if (abs(w) < epsilon){
72      center = surround;
73      surround = BOX(xc,yc,scales[2]);
74      COMPUTE_W(scales[1]);
75  }
76  if (abs(w) < epsilon){
77      center = surround;
78      surround = BOX(xc,yc,scales[3]);
79      COMPUTE_W(scales[2]);
80  }
81  if (abs(w) < epsilon){
82      center = surround;
83      surround = BOX(xc,yc,scales[4]);
84      COMPUTE_W(scales[3]);
85  }
86  if (abs(w) < epsilon){
87      center = surround;
88      surround = BOX(xc,yc,scales[5]);
```

```
89          COMPUTE_W(scales[4]);
90      }
91      if (abs(w) < epsilon){
92          center = surround;
93          surround = BOX(xc,yc,scales[6]);
94          COMPUTE_W(scales[5]);
95      }
96      if (abs(w) < epsilon){
97          center = surround;
98      }
99
100     // now do local tone mapping
101     float ldr = (ls/(1+center))*255;
102
103     // write tone mapped RGB values to output PBO
104     out_data[i] = make_uchar4(
105                 min(ldr*__powf(res.x/lum, gamma), 255.0f),
106                 min(ldr*__powf(res.y/lum, gamma), 255.0f),
107                 min(ldr*__powf(res.z/lum, gamma), 255.0f),
108                 255);
109 }
```

Listing B.10: Dodging-and-burning kernel using shared memory for SAT access.

Once again, the code in Listing B.10 uses macros to reduce clutter and simplify modification. The macro `CACHE`, first used on Line 47, is used to simplify access to the `cache` shared memory array, and the macro `BOX`, first used on Line 65, uses the array `cache` to compute the average scaled luminance surrounding a given location. These macros are defined in Listing B.11.

```
1 #define CACHE(x,y) (cache[y][x])
2 #define BOX(x,y,radius) ( \
3   ( \
4    CACHE((x)+((int)(0.5*radius)),(y)+((int)(0.5*radius))) - \
5    CACHE((x)+((int)(0.5*radius)), y-((int)(0.5*radius))-1) - \
6    CACHE( (x)-((int)(0.5*radius))-1, (y)+((int)(0.5*radius))) + \
7    CACHE( (x)-((int)(0.5*radius))-1, (y)-((int)(0.5*radius))-1)
8   ) / (radius*radius) \
9 )
```

Listing B.11: Macros used in the shared memory dodging-and-burning implementation.

B.2 GLSL

This section lists the GLSL and C++ code used to implement the four shader programs or submodules shown in Figure 5-19 in Chapter 5.

B.2.1 Luminance compuation

The GLSL fragment shader code used for luminance computation is given in Listing B.12.

```
// hdr input texture
uniform sampler2D hdrTex;

void main(void)
{
  // get color from the texture
  vec4 currTexel = texture2D(hdrTex, gl_TexCoord[0].st);
  float lum = dot(vec3(0.2125,0.7154,0.0721), currTexel.rgb);

  // save luminance and log-luminance
  // in output channels
  gl_FragColor.r = log(lum + 1.0f);
  gl_FragColor.y = lum;
}
```

Listing B.12: A GLSL fragment shader for computing luminance.

B.2.2 Scaled luminance computation

Scaled luminance is computed with the fragment shader shown in Listing B.13.

```
uniform sampler2D lumTex; // luminance map
uniform float alpha;

void main(void)
{
  // get luminance for current texel
  float l = texture2D(lumTex, gl_TexCoord[0].st).g;

  // retrieve average log-lum from top of mipmap
  float lMean = tex2Dlod(lumTex, float4 (0.5, 0.5, 0.0, 100)).r;
  lMean = exp(lMean);

  // compute scaled luminance
  float ls = (alpha / lMean) * l;

  // save output
  gl_FragColor.x = ls;
  gl_FragColor.y = l;
}
```

Listing B.13: A GLSL fragment shader for computing scaled luminance.

B.2.3 SAT generation

SAT generation in the shader tone mapping module is accomplished using a combination of GLSL shaders and C++ code. The C++ code responsible for manipulating the shaders is gathered in the class `SATSengupta`, described in Section 6.2.4. The most important code in the `SATSengupta::compute()` function, which invokes a series of GLSL shaders to generate a SAT from an input texture, is shown in Listing B.14.

```
1
2  // ------------------
3  // Horizontal pass
4  // ------------------
5  int topLevel = ceil(log((double)_w)/log(2.0));
6  libGL::GLTexture* in;
7  libGL::GLTexture* out;
8
9  // screen-aligned quad for invoking shaders
10 libGL::GLCanvas* canvas;
11
12 // up sweep: reduction
13 for (int d = 1; d <= topLevel; d++)
14 {
15    if (d == 1) in = _inTex;
16    else in = _texLevelsH_PartialSum[d-1];
17    out = _texLevelsH_PartialSum[d];
18    canvas = _canvasHLevels[d-1];
19
20    // bind appropriate Frame Buffer Object
21    _fboHLevels[d]->bind();
22    _fboHLevels[d]->setTarget(out);
23
24    // enable the reduction shader
25    _shaderHorUp->enable();
26
27    glActiveTexture(GL_TEXTURE0);
28    _shaderHorUp->setUniform1f("imgw", in->w());
29
30    in->bind();
31    _shaderHorUp->setUniform1i("tSrc", 0);
32    canvas->draw();
33    in->release();
34
35    // done
36    _shaderHorUp->disable();
37    _fboHLevels[d]->release();
38 }
39
40 // down sweep
41 libGL::GLTexture* parentTex;
42 libGL::GLTexture* currTex;
43 for (int d = topLevel-1; d >= 0; d--)
```

```
{
  // set outpu
  out = _texLevelsH_Result[d];

  // set parent texture
  if (d == topLevel-1) parentTex = _texLevelsH_PartialSum[d+1];
  else                 parentTex = _texLevelsH_Result[d+1];

  // set current texture
  if (d == 0) currTex = _inTex;
  else        currTex = _texLevelsH_PartialSum[d];

  // set the input canvas
  canvas = _canvasHLevels[d];

  // bind appropriate Frame Buffer Object
  _fboHLevels[d]->bind();
  _fboHLevels[d]->setTarget(out);

  // enable the down pass shader
  _shaderHorDown->enable();
  _shaderHorDown->setUniform1f("imgw", currTex->w());
  _shaderHorDown->setUniform1f("parentw", parentTex->w());

  // bind the input textures
  glActiveTexture(GL_TEXTURE0);
  parentTex->bind();
  _shaderHorDown->setUniform1i("tParentLevel", 0);

  glActiveTexture(GL_TEXTURE1);
  currTex->bind();
  _shaderHorDown->setUniform1i("tCurrLevel", 1);

  // invoke the program
  canvas->draw();

  // cleanup
  _shaderHorDown->disable();
  _fboHLevels[d]->release();
}

// ...
// Vertical pass is similar to horizontal one
// ...

// point output at the final result
_outTex = _texLevelsV_Result[0];
```

Listing B.14: A GLSL fragment shader for computing scaled luminance.

The code in Listing B.14 uses Sengupta's algorithm to for scan computation (see Section 5.2.4), where each parallel algorithm step is performed by a shader program.

As explained in Section 5.2.4, scan computation using Sengupta's algorithm is accomplished using two phases: a reduction phase and a down-sweep phase. Therefore, a total of four GLSL shader programs are used for SAT generation: two reduction shaders, each customized to operate on columns or rows, and two down-sweep shaders, also invidually modified to operate on columns or rows. Since column scans are a simple permutation of row scans, only the row scan shaders are shown here. The following subsections detail the GLSL shader programs used for each phase of row scan computation.

B.2.3.1 Reduction phase

The vertex and fragment shaders used for reduction scan phase are shown in Listing B.15 and B.16, respectively. These shaders are interfaced through the object `_shaderHorUp` in Listing B.14.

```
uniform float imgw;

void main ( )
{
  gl_Position = ftransform();

  float tw = 1.0/imgw;     // texel width
  vec2 uv = gl_MultiTexCoord0.xy;

  gl_TexCoord[0].xy = vec2(2*uv.x-0.5*tw, uv.y);
  gl_TexCoord[0].wz = vec2(2*uv.x+0.5*tw, uv.y);
};
```

Listing B.15: GLSL vertex shader code for the reduction phase of Sengupta's scan algorithm.

```
uniform sampler2D tSrc;
void main ( )
{
  vec4 uv = gl_TexCoord[0];
  float curr = texture2D(tSrc, uv.xy).r;
  float next = texture2D(tSrc, uv.wz).r;

  // remember, output is a single-channel texture
  gl_FragColor = curr+next;
};
```

Listing B.16: GLSL fragment shader code for the reduction phase of Sengupta's scan algorithm.

B.2.3.2 Down-sweep phase

All important work involved in the down-sweep phase is computed using fragment shaders. The fragment shader program used for the down-sweep phase is given in Listing B.17. To better understand the code in Listing B.17, compare it to its algorithm specification, given in Section 5.2.4.

```
uniform sampler2D tParentLevel;
uniform sampler2D tCurrLevel;
uniform float imgw;
uniform float parentw;

void main ( )
{
  vec4 uv = gl_TexCoord[0];
  float tw = 1.0/parentw;

  // use de-normalized x value to make life simpler
  int x = (int)(uv*imgw);

  float t;
  vec2 p = uv;

  if(x > 0)
  {
    if ((int)x % 2 != 0){
      // if the parent texture width is 0
      // uv.x == 0.5 will address the border between the two
      // center texels.
      // If this is the case, make small adjustment
      if(uv.x == 0.5 && (int)parentw % 2 == 0)
        p -= vec2(0.5*tw, 0);

      t = texture2D(tParentLevel, p);
    }
    else{
      p -= vec2(1.0/imgw, 0);
      t = texture2D(tCurrLevel, uv.xy) +
          texture2D(tParentLevel, p);
    }

  }
  else{
    t = texture2D(tCurrLevel, uv.xy);
  }

  gl_FragColor.r = t;

};
```

Listing B.17: GLSL fragment shader code for the reduction phase of Sengupta's scan algorithm.

B.2.4 Dodging-and-burning

The final shader program, which uses the partial results generated by the shaders listed in previous subsections to apply the final local tone mapping, is given in Listing B.18.

```
uniform sampler2D satTex;
uniform sampler2D lumTex;
uniform sampler2D hdrTex;

uniform float alpha;
uniform float phi;
uniform float epsilon;
uniform float gamma;

uniform float imgw;
uniform float imgh;

float box (float s, vec2 uv)
{
  // compute normalized size
  vec2 nSize = vec2(s/imgw, s/imgh);

  // retrieve integral of scaled luminance in given region
  // using SAT
  float result = 0;
  result =  texture2D(satTex, uv + 0.5 * nSize).r;
  result -= texture2D(satTex, uv + vec2(0.5, -0.5) * nSize).r;
  result -= texture2D(satTex, uv + vec2(-0.5, 0.5) * nSize).r;
  result += texture2D(satTex, uv - 0.5 * nSize).r;

  // return average scaled luminance in region
  return result/(s*s);
}

void main(void)
{
  float scale[8] = {1.0,3.0,5.0,7.0,11.0,17.0,25.0,39.0};
  vec2 uv = gl_TexCoord[0].xy;

  // retrieve the scaled luminance from red channel
  // of lum tex
  float ls = texture2D(lumTex, uv).r;

  // init first two levels of box filter pyramid
  float center = ls;
  float surround = box(scale[1], uv);

  // compute center-surround
  float dNum = pow(2.0,phi)*alpha;
  float denominator = dNum + center;
  float w = (center - surround) / denominator;

```

```
48  // proceed with dodging-and-burning
49  if ( abs(w) < epsilon)
50  {
51      center = surround;
52      surround = box(scale[2], uv);
53
54      denominator = dNum / pow(scale[1] ,2.0) + center;
55      w = (center - surround) / denominator;
56  }
57  if ( abs(w) < epsilon)
58  {
59      center = surround;
60      surround = box(scale[3], uv);
61
62      denominator = dNum / pow(scale[2] ,2.0) + center;
63      w = (center - surround) / denominator;
64
65  }
66  if ( abs(w) < epsilon)
67  {
68      center = surround;
69      surround = box(scale[4], uv);
70
71      denominator = dNum / pow(scale[3] ,2.0) + center;
72      w = (center - surround) / denominator;
73
74  }
75  if ( abs(w) < epsilon)
76  {
77      center = surround;
78      surround = box(scale[5], uv);
79
80      denominator = dNum / pow(scale[4] ,2.0) + center;
81      w = (center - surround) / denominator;
82
83  }
84  if ( abs(w) < epsilon)
85  {
86      center = surround;
87      surround = box(scale[6], uv);
88
89      denominator = dNum / pow(scale[5] ,2.0) + center;
90      w = (center - surround) / denominator;
91
92  }
93  if ( abs(w) < epsilon)
94  {
95      center = surround;
96  }
97
98  // compress scaled luminance
```

```
99    float ldr = ls / (1.0 + center);
100
101   // retrieve input RGB and luminance values
102   vec4 res = texture2D(hdrTex, uv);
103   float lum = texture2D(lumTex, uv).g; // use green channel
104
105   // save the compressed, RGB result.
106   gl_FragColor = vec4(ldr*pow(res.x/lum, gamma),
107                       ldr*pow(res.y/lum, gamma),
108                       ldr*pow(res.z/lum, gamma),
109                       1);
110 }
```

Listing B.18: GLSL fragment shader code for dodging-and-burning.