J Real-Time Image Proc (2015) 10:273–287
DOI 10.1007/s11554-012-0270-9

SPECIAL ISSUE

High dynamic range imaging pipeline on the GPU

Ahmet Oğuz Akyüz

Received: 27 March 2012 / Accepted: 7 August 2012 / Published online: 12 September 2012
© Springer-Verlag 2012

Abstract Use of high dynamic range (HDR) images and video in image processing and computer graphics applications is rapidly gaining popularity. However, creating and displaying high resolution HDR content on CPUs is a time consuming task. Although some previous work focused on real-time tone mapping, implementation of a full HDR imaging (HDRI) pipeline on the GPU has not been detailed. In this article we aim to fill this gap by providing a detailed description of how the HDRI pipeline, from HDR image assembly to tone mapping, can be implemented exclusively on the GPU. We also explain the trade-offs that need to be made for improving efficiency and show timing comparisons for CPU versus GPU implementations of the HDRI pipeline.

Keywords High dynamic range imaging · GPU programming · Color imaging · Tone mapping · Real-time imaging

1 Introduction

The use of high dynamic range imagery in computer graphics and image processing has gained popularity in recent years. This can be attributed to the increased realism and visual quality that is afforded by use of HDR data. Techniques such as image-based lighting, environment mapping, and special effects such as realistic motion blur and the well known bloom effect all produce improved results if they use HDR data instead of low dynamic range (LDR) data [23].

A. O. Akyüz (✉)
Middle East Technical University, Ankara, Turkey
e-mail: akyuz@ceng.metu.edu.tr

This increased demand for working with HDR content is well matched with the capabilities of modern graphics cards. Currently all modern graphics hardware support floating point textures and renderbuffers. This allows programmers to directly feed in a floating point HDR image and process it on the GPU.

It is, however, often the case that HDR images used in graphics applications are created offline using the CPU, or they are obtained as pre-made from external image databases [8]. However, given the large number of independent pixel operations required to create an HDR image, the process of HDRI assembly is very suitable to be implemented on the GPU. Thus, the first goal of this paper is to demonstrate how to create an HDR image from a set of bracketed low dynamic range (e.g. JPEG) images directly on the GPU by using the OpenGL API.

Due to the limitations of conventional display devices, it is not possible to display HDR imagery directly, although this may change in near future as HDR displays enter the mainstream [1, 25]. Instead, their dynamic range needs to be reduced followed by quantization into an integer 8-bit per color channel data type before they can be shown on a display device. Algorithms that perform dynamic range reduction are called tone mapping (or tone reproduction) operators (TMOs), and they range from simple linear scaling to sophisticated multi-scale approaches that attempt to simulate the human vision (see [5, 10, 23] for excellent reviews).

Similar to the HDR assembly process, most TMOs are comprised of a large number of independent pixel operations which render them suitable for a GPU implementation as well. One of the most popular TMOs that belongs to this category is the photographic tone reproduction operator [21]. Thus, the second goal of this paper is to demonstrate how both the global and local versions of this

operator can be efficiently implemented by using OpenGL fragment shaders. Different from previous work, we will show that the implementation of this operator neither requires expensive convolution nor Fourier transform operations to compute local adaptation luminances.

2 Related work

In this section, we review the previous work that deals with optimizing the HDR imaging pipeline. Cohen et al. [7] introduced the idea of HDR texture mapping on the GPU. As contemporary graphics cards at the time of the study did not support floating point textures, the authors proposed a technique to simulate HDR textures by using multiple 8-bit textures. Battiato et al. [6], on the other hand, provided a state-of-the-art report of the HDRI pipeline from HDR image creation to tone mapping. However, implementation of the pipeline on the GPU was not discussed.

The idea of tone mapping on the GPU was introduced by several authors [3, 11, 12]. In Goodnight et al. [11] and Goodnight et al. [12], the authors implemented Reinhard et al. [21]'s tone mapping operator using fragment shaders. To implement the local version of this operator, they have devised an efficient GPU based convolution operation. Furthermore, the authors have shown how to apply the method to time-varying sequences such as HDR videos. Artusi et al. [3], on the other hand, proposed a general framework to speed-up global tone mapping operators by effectively dividing the workload between the CPU and the GPU.

A real-time tone mapping operator that also models the perception effects was developed by Krawczyk et al. [17]. In this work, the authors modeled several important effects such as visual acuity, glare, and luminance adaptation. Later work implemented a Reinhard-like operator on FPGA architectures [13, 14].

To summarize, previous studies made significant contributions to achieve real-time performance in tone mapping. In this work, however, we explain how the *full* HDR imaging pipeline, from image creation to display, can be implemented in real-time. Different from previous work, we also show how a local tone mapping operator that utilizes local adaptation luminances can be implemented without having to implement neither convolution nor Fourier transform based approaches on the GPU.

3 Theory

In this section, we will briefly explain the theory behind the HDR image generation and tone mapping. Their GPU implementation will be discussed in the following section.

3.1 HDR image assembly

HDR images can be created in several ways: direct capture, rendering, and multiple exposures technique are among the most commonly used ones. Direct capture may become the de facto way of creating HDR images in future, but currently it requires special hardware furnished with HDR sensors and therefore is not commonly used by most photographers. Furthermore, most such devices impose other restrictions such as limited resolution, long capture times, and lack of color support [23]. Rendering, on the other hand, is only suitable for computer generated HDR imagery.

The multiple exposures technique allows photographers to take a bracketed sequence of LDR images using a conventional digital camera, and then merge them into a single HDR image. Figure 1 depicts such a sequence of 9 exposures with each exposure 1-fstop apart from the next one. In that each exposure is properly exposed for a different region in the scene, the final HDR image contains details in both dark and light regions (Fig. 2). Owing to the

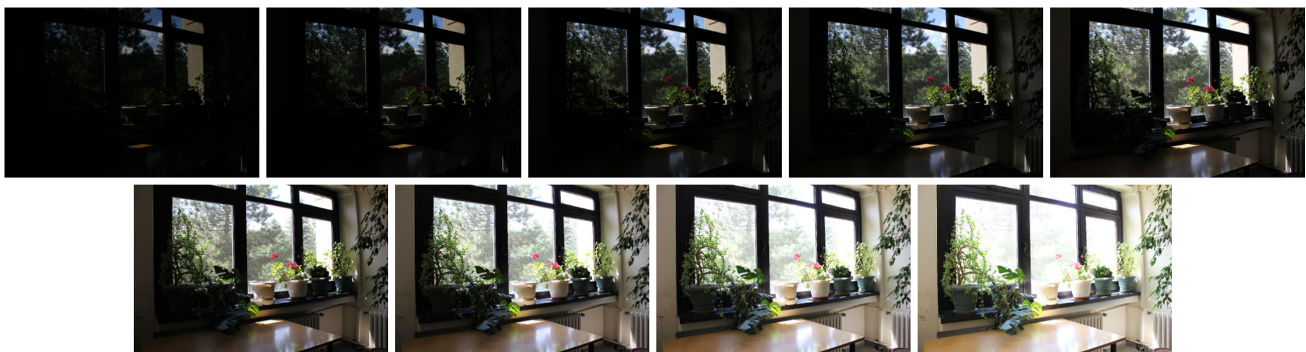

Fig. 1 A bracketed sequence captured with a Canon EOS 550D/T2i digital camera. Each exposure is 1-fstop apart from the next exposure in the series

Fig. 2 The combined result of the sequence in Fig. 1 into a single HDR image which is tone mapped using the technique described in this paper

fact that this technique allows generation of HDR images using off-the-shelf cameras, it is a popular choice among photographers.

A single pixel, I_j, of an HDR image can be computed by using the following formula in the multiple exposures technique:

$$I_j = \sum_{i=1}^{N} \frac{f^{-1}(p_{ij}) w(p_{ij})}{t_i} \Big/ \sum_{i=1}^{N} w(p_{ij}), \tag{1}$$

where N is the number of LDR images, p_{ij} is the value of pixel j in image i,f is the camera response function, w is a weighting function used to attenuate the contribution of poorly exposed pixels, and t_i is the exposure time of image i. One can obtain an HDR image by computing this equation for all pixels.

In this equation, the inverse of the camera response function, f^{-1}, is used to linearize (i.e. degamma) the LDR images as they are typically captured in the non-linear sRGB color space. f^{-1} can be recovered directly from the bracketed sequence using response curve recovery algorithms [9, 19, 24], or it can be assumed to match the sRGB standard. We adopt the latter approach in this paper to benefit from OpenGL's sRGB texture support.

3.2 Dynamic range reduction

Standard display devices such as televisions and computer monitors are designed to display 8-bit per color channel integer input streams (although video cards that can output 10-bit and monitors that can display them have been in use for some time [2]). Due to this limitation, HDR images and video cannot be directly displayed on standard display devices. To display them, their dynamic range needs to be reduced followed by quantization into 8-bit integers. The algorithms that perform this task are called tone mapping (or tone reproduction) operators.[1]

To date, various tone mapping operators have been proposed each with a different approach to dynamic range reduction. TMOs are generally classified as global and local with global operators applying the same compressive function to each pixel while local operators changing the shape of this function (thus the degree of compression) based on the statistics of the local neighborhood around each pixel.

One of the most popular TMOs that is commonly used in practice, and that ranks high in user studies, is Reinhard et al.'s [21] photographic tone reproduction operator. This operator comes in two flavors, namely the global and the local operator.

3.2.1 Global operator

The global operator starts by computing the *key* of the scene which indicates its overall subjective brightness. The key is approximated by the log-average luminance (see Sect. 3.3 for color space conversions needed to obtain luminance from color and vice-versa), $\bar{L}_{\mathrm{w}}$:

$$\bar{L}_{\mathrm{w}} = \exp\left(\frac{1}{N}\sum_{x,y}\log(\delta + L_{\mathrm{w}}(x,y))\right). \tag{2}$$

Here, $L_{\mathrm{w}}(x,y)$ indicates the world luminance[2] of pixel (x,y) and δ is a small offset added to avoid singularity that may occur at $\log(0)$ if black pixels are present in the image. The summation is performed across the entire image.

Once the log-average luminance is computed, it is mapped to a user defined value, a, based on the desired subjective brightness of the scene. This is accomplished by:

$$L(x,y) = \frac{a}{\bar{L}_{\mathrm{w}}} L_{\mathrm{w}}(x,y). \tag{3}$$

For most scenes illuminated by moderate lighting, a can be set to 0.18. To render darker scenes, it may be reduced to 0.09 or 0.045 (or less), and for lighter scenes it may be increased to 0.36 or 0.72 (or more).

Once the image is scaled in this manner, the actual dynamic range compression is performed using a sigmoidal compression function:

$$L_{\mathrm{d}}(x,y) = \frac{L(x,y)}{1 + L(x,y)}, \tag{4}$$

where $L_{\mathrm{d}}(x,y)$ represents the display luminance. While this equation is guaranteed to bring all pixels into a displayable

[1] Quantization into 8-bits is not part of tone mapping, but it is a necessary step to create displayable images.

[2] The subscript w indicates world luminance which may be in absolute or relative units depending on the calibration of the image.

range, some intentional burning in bright areas may be desired to create a more natural photographic look. The amount of burning can be controlled by a user defined parameter, L_{white}:

$$L_{\mathrm{d}}(x,y)=\frac{L(x,y)\left(1+\frac{L(x,y)}{L_{\mathrm{white}}^2}\right)}{1+L(x,y)}. \quad (5)$$

In this final equation, all luminance values greater than L_{white} will be mapped to 1; that is they will burn out. If L_{white} is set to infinity, this equation will reduce to Eq. 4.

3.2.2 Local operator

The local operator resembles the global operator in that tone mapping is performed via a similar formula:

$$L_{\mathrm{d}}(x,y)=\frac{L(x,y)}{1+V_1(x,y,s)}. \quad (6)$$

The difference, however, is that V_1 represents the local adaptation luminance in the neighborhood around the pixel (x,y). The size of this neighborhood is controlled by the scale parameter, s. V_i can be computed as

$$V_i(x,y,s)=L(x,y)\otimes R_i(x,y,s), \quad (7)$$

where R_i is a Gaussian profile of the form

$$R_i(x,y,s)=\frac{1}{\pi(\alpha_i s)^2}\exp\left(-\frac{x^2+y^2}{(\alpha_i s)^2}\right). \quad (8)$$

To determine the appropriate scale, Reinhard et al. [21] propose to compute the difference of Gaussian convolutions at different scales, V_1 and V_2. When the difference between the two convolution results is above a threshold, the appropriate scale is found. This, in effect, computes the largest uniform region around each pixel, which serves as an adaptation region for that pixel. This can be formalized as:

$$V(x,y,s)=\frac{V_1(x,y,s)-V_2(x,y,s)}{2^{\phi}a/s^2+V_1(x,y,s)}, \quad (9)$$

where ϕ is a sharpening parameter. Here the goal is to find the largest scale s_{m} that satisfies:

$$|V(x,y,s_{\mathrm{m}})|<\epsilon, \quad (10)$$

where ϵ is a user parameter. Larger values give rise to larger adaptation neighborhoods. Reinhard et al. [21] suggests using $\phi=8.0$ and $\epsilon=0.05$ as default parameters.

The photographic tone mapping operator poses two challenges for a GPU implementation. First, the log-average luminance of the whole image needs to be computed—an operation which is not GPU friendly. Second, local adaptation luminances need to be computed for the local operator. This amounts to convolving the image with filters of varying sizes, which is also not a GPU friendly operation. In this paper, we show that both problems can be solved by judicious use of mipmapping.

3.3 Dealing with color

The dynamic range compression described in the previous section expects luminance values as input. However, in practice, we typically deal with color images. To convert color values to luminance, we need to employ color space transformations. After tone mapping we can invert these transformations to retrieve the modified color values. In this section, we briefly highlight the key features of these color space transformations. For a more complete treatment, we refer the reader to literature on color imaging [22, 27].

To compute the luminance value for a given color triplet, we first need to know its color space. If this information is not available, we can assume that the HDR image is in the sRGB color space as this is the default output color space for most digital cameras. We also assume that the HDR image contains linear color values. This is also a reasonable assumption as the HDR generation process typically linearizes the individual exposures before combining them into the HDR image. We can then convert an sRGB color value into its CIE XYZ representation with the following transformation [15]:

$$\begin{bmatrix}X_{\mathrm{w}}\\Y_{\mathrm{w}}\\Z_{\mathrm{w}}\end{bmatrix}=\begin{bmatrix}0.4124 & 0.3576 & 0.1805\\0.2126 & 0.7152 & 0.0722\\0.0193 & 0.1192 & 0.9505\end{bmatrix}\begin{bmatrix}R_w\\G_w\\B_w\end{bmatrix}. \quad (11)$$

In the CIE XYZ color space, the Y component encodes the luminance. Thus, Y_{w} is equal to the world luminance L_{w} that we used in the previous section. We can now compress Y_{w} to obtain the display luminance Y_{d} which is equal to L_{d} in Eqs. 4 and 5.

The output RGB colors can be computed by:

$$C_{\mathrm{d}}=\left(\frac{C_{\mathrm{w}}}{Y_{\mathrm{w}}}\right)^c Y_{\mathrm{d}} \quad (12)$$

where $C=R,G,B$ and c is used for optional saturation adjustment. Setting $c>1$ increases saturation while $c<1$ decreases it. It is worth noting that all of these transformations described in this section are performed for each pixel independently, and thus are very amenable to benefit from GPU implementation.

4 Mipmapping

As mipmapping constitutes a key part of our algorithm which we use to compute the global average, $\bar{L}_{\mathrm{w}}$, and local

adaptation luminances, V_1 and V_2, a brief review of the concept can be useful. Mipmapping, first introduced by Williams [26], is a commonly used technique to map texture images onto polygonal surfaces. The idea of mipmapping is to store a texture image as a pyramid of multiple levels, where each level contains a progressively lower-resolution version of the original image. During texture mapping, the level which most closely matches the screen size of the polygon that is being textured is chosen as the source image.

In OpenGL, the mipmap levels for two dimensional textures can be explicitly provided by the programmer using `glTexImage2D` or `glTexSubImage2D` calls. Alternatively, the programmer can request automatic generation of mipmaps from the OpenGL server by using the `glGenerateMipmap` function. In this case, each level of the mipmap chain is created from the previous level by using filtered reduction (the first level must be provided by the user). Although no specific filtering algorithm is enforced by the OpenGL standard, most implementations use box filtering [16]. Thus, each pixel in a higher mipmap level represents the local average of pixels in the lower level (see Fig. 3).

A mipmapped 2D texture can be sampled in the fragment shader using the GLSL construct `texture(s, xy, b)`. Here, `s` is a handler to the texture that will be sampled. `xy` indicates the coordinates inside the texture image, and `b` is a bias that will be added to the mipmap level computed by OpenGL. If the texture size is equal to the screen size of the polygon that is being rendered, one can think of `b` equal to the mipmap level index.

As mentioned above, we use mipmapping to efficiently compute a measure of local adaptation luminance around each pixel. In the original algorithm of Reinhard et al. [21], this is performed by computing a Gaussian convolution around each pixel. It is therefore appropriate to discuss the differences between these two approaches. In convolution, each pixel is placed in the center of a convolution kernel and a local average is computed within that kernel. The kernel size can be increased to compute convolution over a larger neighborhood. In mipmapping, however, the downsampled versions of the original image are computed once using filtered reduction. Although this is very efficient as each pixel is used only once, it may give rise to an asymmetrical neighborhood for computing local adaptation luminances. The difference between the two approaches is shown in Fig. 4.

In this figure, R_1, R_2, and R_3 indicate the local adaptation regions around the pixel shown in red. In Gaussian convolution, this pixel is always placed in the center of the convolution kernel. In mipmapping, this is not always the case as shown in the figure. We show in Sect. 6 that this difference has only a minor effect on the quality of the results, and therefore the heavy computational cost of convolution can be avoided in most cases.

5 Practice

In this section, we will demonstrate how the theory described in the previous section can be put into practice by using OpenGL. As it would be impractical to illustrate the entire implementation, we will focus on its most crucial features. In our implementation, we used OpenGL 4.2 which was the latest version of OpenGL at the time of this writing. However, lower versions of the language can also be used as long as they support the required functionality such as mipmapping, floating point textures, sRGB textures and framebuffers, and GLSL. All of these versions are supported in OpenGL version 2.1 with appropriate extensions and natively on 3.0 onwards.

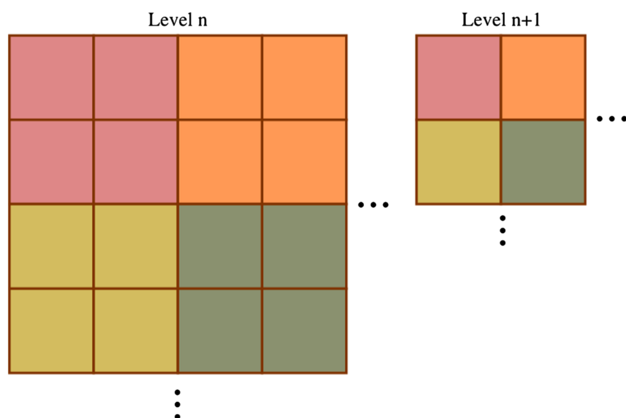

Fig. 3 A higher mipmap level is created from the lower mipmap level by filtered reduction

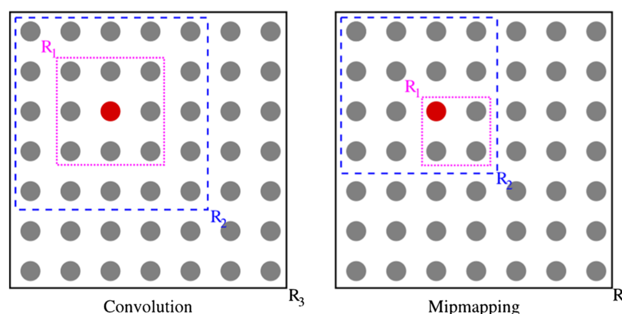

Fig. 4 The difference between proper convolution and mipmapping for computing local adaptation luminances. R_i indicate local adaptation regions at different scales

5.1 OpenGL setup for HDRI assembly

To create an HDR image on the GPU, we need to access the pixels of the bracketed LDR images in the fragment shader. The most convenient way to achieve this is to upload LDR images as textures and sample from them using an appropriate sampler. The code snippet below demonstrates how to create these textures and the sampler:

```
int numImages = 10; // number of LDR images
GLuint *ldrTex = new GLuint [numImages];
GLuint ldrSampler;

glGenSamplers(1, &ldrSampler);
glGenTextures(numImages, ldrTex);

glSamplerParameteri(ldrSampler, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST);
```

Listing 1: Generation of source sampler and textures.

Note that we generate only one sampler as it will be shared by all of the texture units. Once the textures are generated, we can upload the LDR images which are stored as one dimensional arrays with color channels interleaved as red, green, and blue:

```
for (int i = 0; i < numImages; ++i) {
  glBindTexture(GL_TEXTURE_2D, ldrTex[i]);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB8,
       w, h, 0, GL_RGB, GL_UNSIGNED_BYTE,
       ldrImg[i]);
}
```

Listing 2: Uploading of LDR images into textures.

Here, `w` and `h` denote the dimensions of the LDR images. It is important to note that the internal format of the textures is set to sRGB. This will allow us to retrieve the linearized color values when we sample from these textures in the fragment shader. In other words, sampling from an sRGB texture will approximate the result of $f^{-1}(p_{ij})$ in Eq. 1.

We can now set up the source texture and sampler bindings. First, we need to bind the LDR sampler into all of the texture units as we want to use the same sampler for all units. Second, we need to bind each LDR texture into a different texture unit to be able to access them simultaneously in the fragment shader. These settings can be achieved by:

```
GLint ldrSamplerUnits[16];
for (int i = 0; i < numImages; ++i) {
  ldrSamplerUnits[i] = i;
  glBindSampler(i, ldrSampler);

  glActiveTexture(GL_TEXTURE0 + i);
  glBindTexture(GL_TEXTURE_2D, ldrTex[i]);
}
```

Listing 3: Sampler and texture bindings.

Here, note that in addition to binding textures and samplers, we initialize an array called `ldrSamplerUnits` with sequential integers from 0 to `numImages` −1. This array will later be used to specify which sampler will fetch data from which texture unit in the fragment shader.

The settings above complete the source texture and sampler setup. We can now perform the destination setup which is necessary to store the resulting HDR pixel values. To achieve this, we can create a floating point texture and attach it to one of the color attachment points of a framebuffer object (FBO), and make that FBO the current render target as shown in Listing 4:

```
GLuint hdrTex, fbo;

glGenTextures(1, &hdrTex);
glBindTexture(GL_TEXTURE_2D, hdrTex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
    w, h, 0, GL_RGBA, GL_FLOAT, NULL);
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
glFramebufferTexture(GL_DRAW_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0, hdrTex, 0);
```

Listing 4: Destination setup to write out resulting HDR pixel values.

The last operation we need to perform before the render call is to update the two uniform variables that

will be used in the fragment shader. To this end, we first need to obtain the locations of these uniform variables, bind the HDR creation program, and then upload the values:

```
// Get uniform locations
GLint samplerLoc = glGetUniformLocation(prgHDRCreate, "
    ldrSampler");
GLint imagesLoc = glGetUniformLocation(prgHDRCreate, "
    numImages");

// Switch to the HDR creation program
glUseProgram(prgHDRCreate);

// Initialize uniform variables
glUniform1iv(samplerLoc, numImages, ldrSamplerUnits);
glUniform1i(imagesLoc, numImages);
```

Listing 5: Updating of uniform variables.

It is important to note the usage of `ldrSamplerUnits` which was initialized with sequential integers in Listing 3. By writing its value into the uniform array sampler variable `ldrSampler`, we establish a contract that in the fragment shader `ldrSampler[0]` will sample from texture unit 0, `ldrSampler[1]` will sample from texture unit 1, and so on.

At this point we have completed all the necessary OpenGL API setup for HDR assembly. We can start the process by setting the viewport size equal to the image resolution, and drawing a quad to touch all pixels:

```
// Draw a w by h quad to touch all pixels
glViewport(0, 0, w, h);
DrawQuad();
```

Listing 6: Draw call.

This will initiate the execution of vertex and fragment shaders whose details are provided in the following section.

5.2 Shader setup for HDR assembly

The vertex shader that we need for HDR assembly is a simple pass-through shader which updates the position and texture coordinate attributes of each vertex:

```
#version 420

// Input position and texture coordinates
layout(location=0) in vec2 inPos;
layout(location=1) in vec2 inTexCoord;

out vec2 texCoord;

void main() {
  gl_Position = vec4(inPos, 0.0f, 1.0f);
  texCoord = inTexCoord;
}
```

Listing 7: Vertex shader for HDR assembly.

Note that this vertex shader is not specific for HDR assembly. In fact, we will use the same shader for tone mapping. The heart of the HDR assembly process is implemented in the fragment shader shown in Listing 8:

```
#version 420

in vec2 texCoord;
uniform sampler2D ldrSampler[16];
uniform int numImages; layout(location=0)
 out vec4 fragColor;///<Destination color

void main() {
  const int refId = numImages / 2;

  float weightSum = 0.0;
  vec4 hdr = vec4(0.0, 0.0, 0.0, 0.0);

  for (int i = 0; i < 16; i++) {
    if (i < numImages) {
      vec3 ldr = texture(ldrSampler[i], texCoord).rgb;
      float lum = luminance(ldr);
      float w = weight(lum);
      float exposure = pow(2.0, i - refId);

      hdr.rgb += (ldr / exposure) * w;
      weightSum += w;
    }
  }

  hdr.rgb /= (weightSum + 1e-6);
  hdr.a = log(luminance(hdr.rgb) + 1e-6);
  fragColor = hdr;
}
```

Listing 8: Fragment shader for HDR assembly.

The main function above calls two other functions namely luminance and weight to compute the luminance of each pixel and its contribution to the corresponding HDR value. Because we assume that the LDR images are captured in sRGB color space, the computation of luminance is based on the ITU-R BT.709 primaries [15]:

```
float luminance(vec3 color) {
  return color.r * 0.2126 + color.g * 0.7152 + color.b *
     0.0722;
}
```

Listing 9: Computation of luminance

As for the weighting function, we need to use a function which attenuates the contribution of over- and under-exposed pixels while emphasizing the effect of properly exposed pixels. Several weighting functions are proposed in literature. We choose the tent function proposed by Debevec and Malik [9] due to its simplicity:

```
float weight(float val) {
  if (val <= 0.5) return val * 2.0;
  else return (1.0 - val) * 2.0;
}
```

Listing 10: Weighting function

This function assigns the highest weight for the pixels in the middle of the input range, and linearly decreases it for lower and higher pixel values.

We note that Listing 8 closely adheres to the HDR assembly equation shown in Eq. 1. The main difference is that we assume the LDR exposures to be separated by 1-fstop apart. This allows us to compute the exposure ratios in the pixel shader directly (note the use of refId), instead of getting them from the application. A second difference is that, we let the OpenGL do the linearization of LDR images for us by specifying an internal format of sRGB as shown in Listing 2. If more accuracy is desired, the precomputed actual camera response can be provided to the shader through a uniform array variable.

Finally, it is important to note that we write out the logarithm of the luminance into the alpha channel of the HDR image. This will be useful to compute the log-average luminance via mipmapping as explained in the next section.

5.3 OpenGL setup for tone mapping

Once the draw call in the previous section completes, the HDR image will be stored in the texture hdrTex. For tone mapping, we can bind this as a source texture and sample from it to access the HDR color values. We can then perform dynamic range compression, and write out the resulting compressed pixel values into an sRGB texture to obtain the final displayable image. First let us demonstrate the generation of the tone map output texture, and its binding to the target FBO:

```
GLuint tmTex;

glGenTextures(1, &tmTex);
glBindTexture(GL_TEXTURE_2D, tmTex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB8, w, h, 0, GL_RGB,
    GL_UNSIGNED_BYTE, NULL);
glFramebufferTexture(GL_DRAW_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0, tmTex, 0);
```

Listing 11: Generation of tone map output texture.

We can now create a sampler to sample from the HDR image in the fragment shader. The reason that we cannot use the LDR sampler that we already created is that we need the HDR sampler to have mipmapping enabled. By sampling from the highest mipmap level we can obtain the log-average luminance of the HDR image which is needed for tone mapping.

```
GLuint  hdrSampler;

glGenSamplers(1, &hdrSampler);
glSamplerParameteri(hdrSampler, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_NEAREST);
glBindSampler(0, hdrSampler);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, hdrTex);

// Create a mipmap chain
glGenerateMipmap(GL_TEXTURE_2D);
```

Listing 12: Sampler and texture setup for tone mapping.

Finally, we can update our uniform variables that will be used in the fragment shader and draw a quad to initiate tone mapping.

```
float key = 0.18f;
float Ywhite = 1e6f;
float sat = 1.0f;

GLint hdrSamplerLoc = glGetUniformLocation(prgTonemap, "
    hdrSampler");
GLint keyLoc = glGetUniformLocation(prgTonemap, "key");
GLint YwhiteLoc = glGetUniformLocation(prgTonemap, "Ywhite"
    );
GLint satLoc = glGetUniformLocation(prgTonemap, "sat");

glUseProgram(prgTonemap);

// Update the uniform variables
glUniform1i(hdrSamplerLoc, 0);
glUniform1f(keyLoc, key);
glUniform1f(YwhiteLoc, Ywhite);
glUniform1f(satLoc, sat);

// Enable sRGB framebuffer output and draw
glEnable(GL_FRAMEBUFFER_SRGB);
glViewport(0, 0, w, h);
DrawQuad();
```

Listing 13: Sampler and texture setup for tone mapping.

Here, `key`, `Ywhite`, and `sat` are user-defined parameters and can be modified to change the appearance of the tone mapping result as will be demonstrated in Sect. 6.

5.4 Shader setup for tone mapping

The vertex shader that we use for tone mapping is identical to the vertex shader for HDR assembly (see Listing 7). The main work for tone mapping is performed inside the fragment shader as shown below:

```
#version 420

uniform float key;
uniform float Ywhite;
uniform float sat;

uniform sampler2D hdrSampler;

in vec2 texCoord;
layout(location=0) out vec4 fragColor;

void main() {
  vec3 hdr = texture(hdrSampler, texCoord).rgb;

  float logAvgLum = exp(texture(hdrSampler, texCoord, 20.0)
      .a);

  fragColor.rgb = tonemap(hdr, logAvgLum);
}
```

Listing 14: Sampler and texture setup for tone mapping.

The tone mapping routine closely follows the description in Sect. 3.2. First the linear sRGB values are converted to XYZ. Tone mapping is then performed to compress the luminance. Finally, the compressed luminance is used to obtain the displayable RGB values with an optional saturation adjustment:

```
vec3 tonemap(vec3 RGB, float logAvgLum) {
  vec3 XYZ = RGB2XYZ(RGB);

  float Y = (key / logAvgLum) * XYZ.y;
  float Yd = (Y * (1.0 + Y / (Ywhite * Ywhite))) / (1.0 + Y
      );

  return pow(RGB / XYZ.y,
      vec3(sat, sat, sat)) * Yd;
}
```

Listing 15: Global tone mapping implementation.

The implementation of the `RGB2XYZ` routine is straightforward and omitted for brevity.

The tone mapping implementation in Listing 15 performs global tone mapping. For some applications, it may be desirable to perform local tone mapping as it better preserves the visibility of details. Previous GPU-based approaches for local tone mapping implemented convolution operations on the GPU. Here, we demonstrate that reasonable results can be obtained by simply using OpenGL's mipmapping ability in lieu of convolutions.

```
vec3 tonemap_lc(vec3 RGB, float logAvgLum) {
  ...
  float La; // local adaptation luminance
  float factor = key / logAvgLum;
  float epsilon = 0.05, phi = 8.0;
  float scale[7] = float[7](1, 2, 4, 8, 16, 32, 64);
  for (int i = 0; i < 7; ++i) {
    float v1 = exp(texture(hdrSampler, texCoord, i).a) *
        factor;
    float v2 = exp(texture(hdrSampler, texCoord, i+1).a) *
        factor;

    if (abs(v1 - v2) / ((key * pow(2, phi) / (scale[i] *
        scale[i])) + v1) > epsilon) {
        La = v1;
        break;
    }
    else
        La = v2;
  }

  float Yd = Y / (1.0 + La);
  ...
}
```

Listing 16: Local tone mapping implementation.

To approximate convolutions, we compute V_1 and V_2 from consecutive mip levels. For this approach to work it is important to set the minification parameter of the sampler as `GL_LINEAR_MIPMAP_NEAREST` as shown in Listing 12. Note that this approach is not only faster than computing convolutions as was done in Goodnight et al. [11], but also much easier to implement.

Once the HDR image is created and tone mapped, the results can be downloaded back to CPU using the `glGetTexImage` function of OpenGL.

6 Results

In this section, we demonstrate representative results that were obtained by using the algorithms described in this paper. We will first show the effect of changing the tone mapping parameters on the resulting images, and then demonstrate that our GPU implementation produces similar results to two reference CPU implementations. We will then illustrate the performance that can be gained by using our method and then compare it with a standard convolution based approach.

Figure 5 depicts tone mapped versions of two HDR images that were created using 9 exposures captured with a Canon EOS550D/T2i digital SLR camera. On the left column, we demonstrate the effect of changing the *key* parameter of the tone mapping operator. As it can be seen, increasing the key value results in progressively brighter images. On the right column, we demonstrate the effect of changing the burn-out threshold, or L_{white} in Eq. 5. As this threshold is reduced we can see more pixels getting clamped at the highest possible value. For instance, while the details outside the window is visible in the top image, this region burns-out in the bottom image. Thus, this parameter can be used to controllably burn bright regions in an image to create an artistic effect. An automatic method to estimate reasonable values for these parameters is explained by Reinhard [20].

We also illustrate the influence of the saturation parameter in Fig. 6. We remind that saturation adjustment is not part of tone mapping, but can be applied as a post

Fig. 5 The *left column* depicts the tone mapping results with increasing key value in each row (0.18, 0.36, and 0.72 from *top to bottom*). The *right column*, on the other hand, depicts tone mapping results with decreasing burn-out threshold (10^6, 5, and 2 from *top to bottom*)

Fig. 6 The effect of changing the saturation parameter. The image on the *left* has the saturation parameter set to 0.5 and the image on the *right* to 1.5. The *center* image has no post tone mapping saturation adjustment (i.e. parameter set to 1.0)

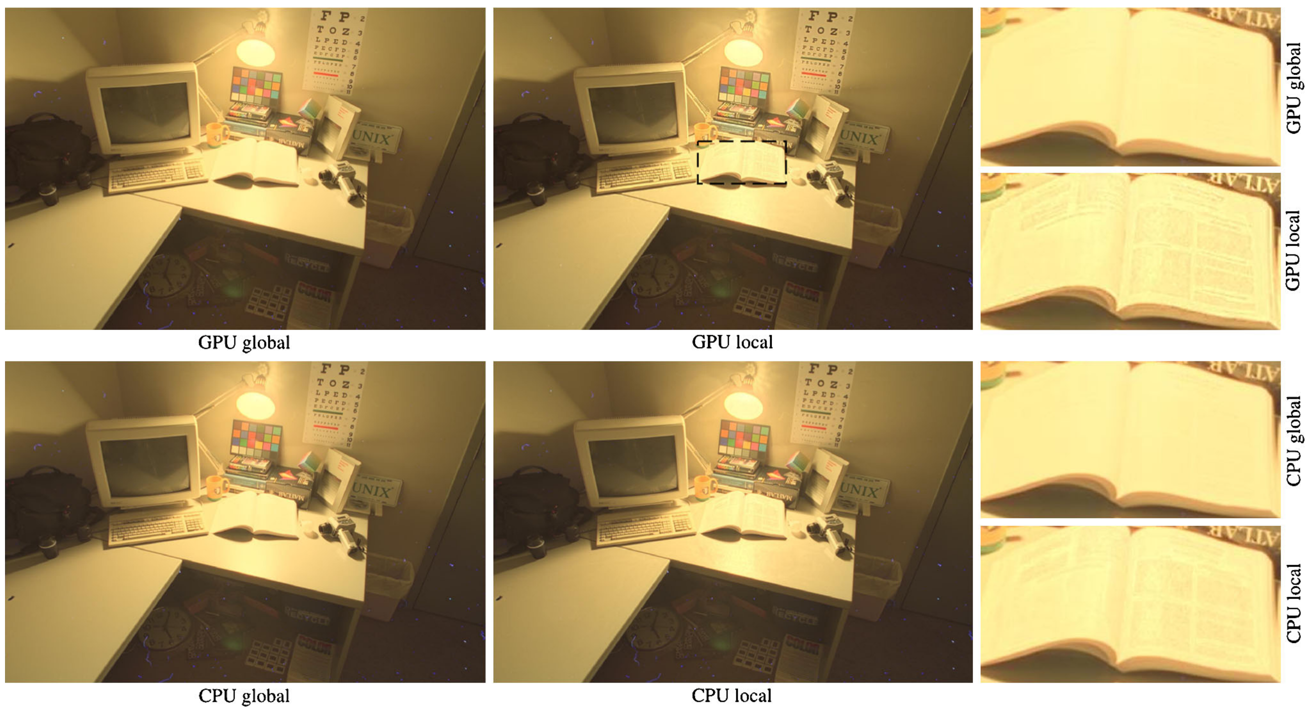

Fig. 7 Global (*left*) versus local (*middle*) tone mapping. As can be seen in the close-ups, the local operator better preserves the visibility of details. The *top row* shows the results obtained by our GPU implementation, whereas the *bottom row* shows the results of a reference CPU implementation [18]

processing operation to create the desired final look. As can be seen from the figure, setting a low saturation parameter such as 0.5 yields a more grayscale result while a high saturation parameter such as 1.5 exaggerates the color saturation.

The difference between the global and local operators is depicted at the top row of Fig. 7. As expected, the local operator better preserves the visibility of details as can be seen in the close-ups on the right. At the bottom row of the same figure, we show the results generated by using a reference CPU implementation.[3] As the figure shows, our results are very similar to the CPU implementation. In fact, the visibility of the details on the book appears to have been better preserved by our method. The difference could be attributed to using different scale factors. Whereas the reference implementation uses 1.6 as the ratio of two scales, we had to use 2.0 due to mipmapping.

Next, we compare our results with two reference CPU implementations using a qualitative metric (Fig. 8). In this figure, the top row shows the global photographic tone mapping results obtained by our method as well as the implementation of the same method in the pfstmo package (pfstmo_reinhard02) and the original implementation of Reinhard et al. [21]. On the second row, we can see the visible differences as detected by the dynamic range independent visual quality assessment metric [4]. Here, the

[3] We used pfstmo_reinhard02 operator from pfstmo package [18].

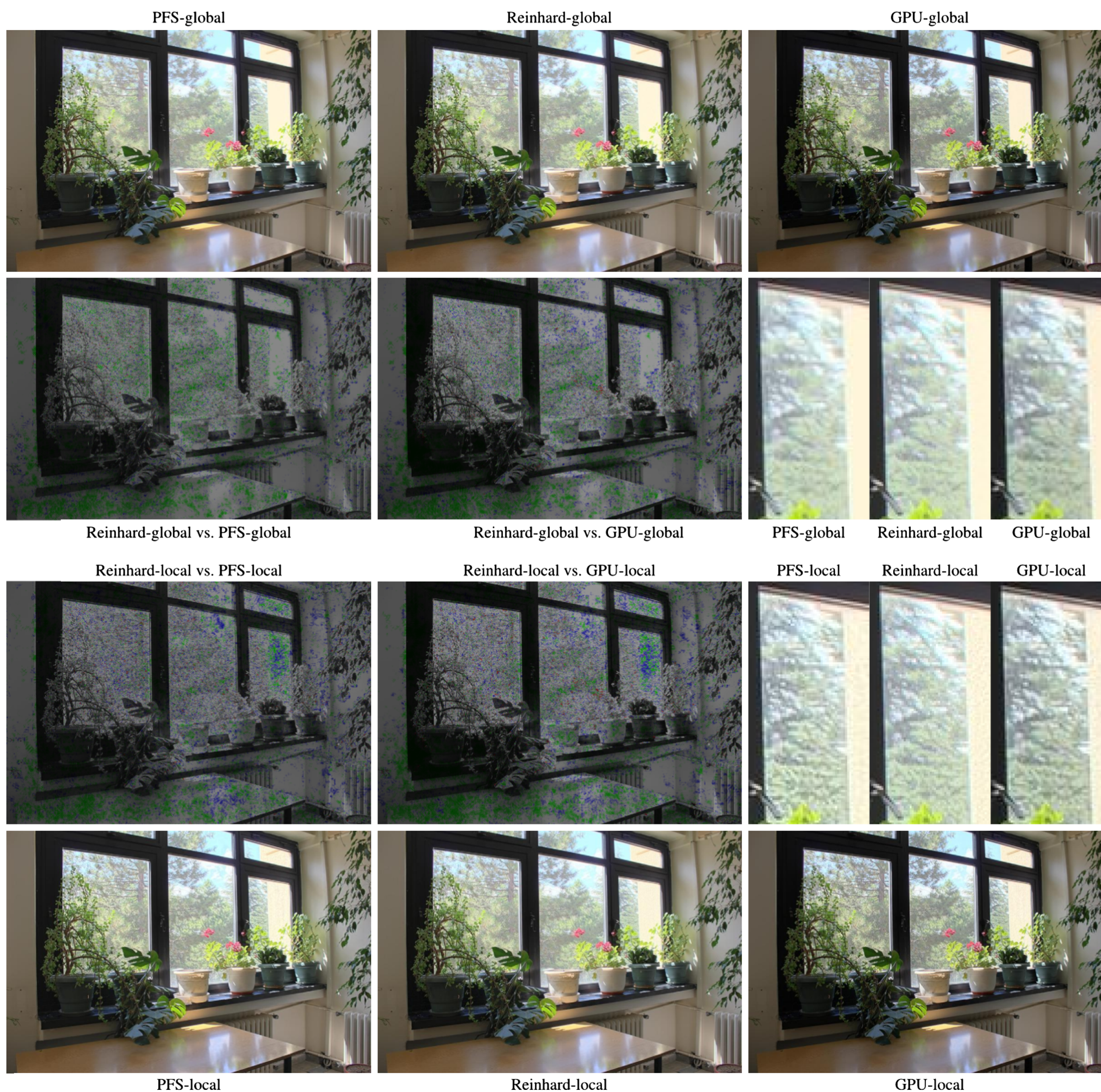

Fig. 8 Qualitative comparison using the dynamic range independent image quality metric [4]. *Top row* shows the results of global tone mapping using different implementations of the photographic tone mapping operator: pfstmo_reinhard02, Reinhard et al.'s original implementation (acquired from:http://www.cs.utah.edu/~reinhard/cdrom), and our GPU implementation. The *bottom row* shows the same for the local operator. Close-ups are also shown for visual inspection. Refer to text for more details

green color indicates the loss of contrast, blue indicates the amplification of contrast, and red indicates the reversal of contrast. We can see that the differences between the two CPU implementations are minor[4] and similar to the differences between our result and Reinhard et al.'s [21] original implementation. Visual inspection of a selected region confirms this similarity. At the bottom two rows, we show the same result but this time for the local operator. Again the differences between the two CPU methods and our GPU method are comparable. The close-ups show the enhanced details.

We show further set of results obtained by our method together with the reference implementation in Fig. 9 using

[4] Such differences between two implementations can be caused by different post processing operations after tone mapping such as normalization, clamping, quantization which are not elaborated in the original paper but are nevertheless used in the implementations.

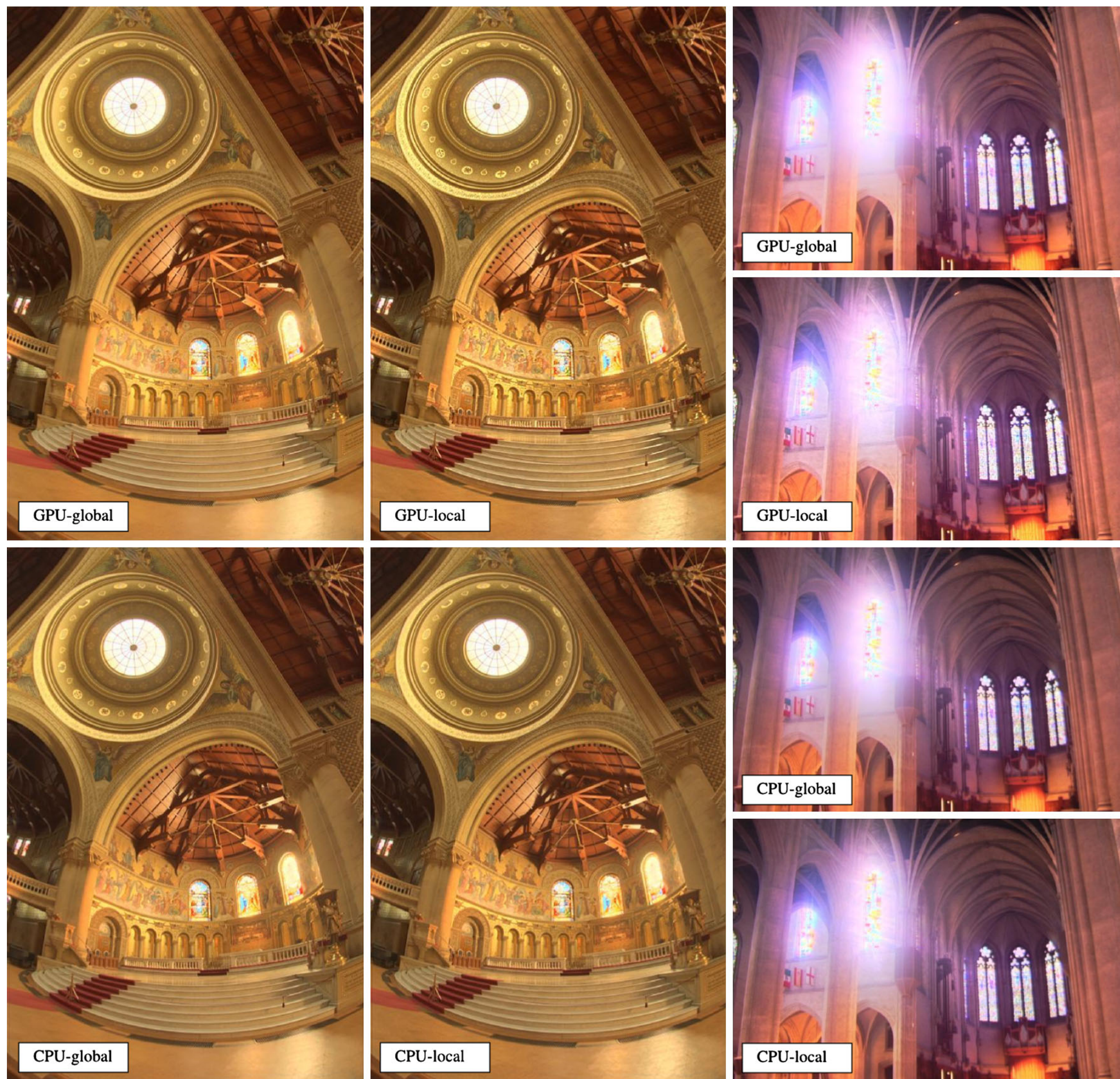

Fig. 9 Comparison of our results (*top*) with a reference CPU implementation (*bottom*). Results of the global operator are shown on the left for the memorial image and top for the nave image. The dynamic ranges of the images are 5.53 and 8.50 orders of magnitude respectively

well-known HDR images. As can be seen from the figure, our results are qualitatively similar to the reference CPU implementation, but are obtained at a fraction of the time of the latter.

We provide a run time comparison to illustrate the performance benefits of our method. Table 1 lists the results of such a comparison obtained by creating and tone mapping an 18 megapixels (MP) HDR image created from 9 exposures captured by a Canon EOS 550D (Fig. 1). In this test we used a high end CPU and a GPU. As the results indicate, both creating and tone mapping an HDR image on the GPU yields immense performance benefits. HDR assembly, on average, yields 2–3 orders of magnitude improvement, while tone mapping yields 3–4 orders of magnitude. If disk I/O and GPU texture upload times are included in the timings, creating an HDR image takes about 13.8 s on the CPU whereas it takes only 4.4 s on the GPU.

Table 1 Performance comparison of creating and tone mapping an HDR image on the CPU versus GPU in frames per second

Device	HDR gen.	Global TM	Local TM
CPU[a]	0.18 fps	0.12 fps	0.015 fps
GPU[b]	65 fps	137 fps	103 fps

Timings do not include disk I/O and GPU texture upload times

[a] Intel Core i7 at 3.20 GHz

[b] Nvidia GeForce GTX 590

As for the memory consumption, the total GPU memory in bytes required for storing LDR images is given by $N \times w \times h \times 3$, where N is the number of exposures, and w and h are the dimensions of the images. The HDR image occupies $w \times h \times 4 \times 4$ bytes of memory as it needs to be

Table 2 Performance effect of the individual parts on HDR generation on the GPU

Configuration	Frame rate
Full	65 fps
No mipmap generation	84 fps
No texture look-up	107 fps
No luminance computation	69 fps
No weight computation	69 fps

Table 3 Performance of convolving an 18 MP image with varying sized kernels on the GPU

Kernel size	Frame rate
3×3	210 fps
7×7	78 fps
11×11	54 fps
15×15	41 fps
19×19	33 fps

in four component per pixel floating point format. The full mipmap chain requires approximately 1.33 times this number.

We conducted more experiments to understand which part of the algorithm takes the most GPU time. As accurate measurement of timings on the GPU is not straightforward, we omitted individual parts of our algorithm to see its effect on the frame rate. The results are reported in Table 2. We can see that the majority of the time is spent on texture look-ups from the source exposures. This is followed by the time it takes to generate a mipmap chain which approximately reduces the frame rate by 23 %. Luminance and weight computations have very small impact on the performance.

Finally, we investigated how long a standard convolution operation on the GPU takes. As can be seen in Table 3, when the kernel size is 7×7 or greater, the convolution alone takes more time than our mipmap optimized implementation. Given that typically larger kernels would be required to compute local adaptation luminances, the performance of the convolution approach is likely to be even lower in practice. This indicates that our algorithm is not only simpler to implement, but also outperforms convolution without compromising quality.

These results underline the importance of transitioning to a full GPU pipeline for both creating and tone mapping high resolution HDR images.

7 Conclusions

With high resolution HDR images becoming more common in image processing and computer graphics applications, their rapid processing is gaining importance. In this paper, we have shown how one can achieve real-time performances by implementing the full HDRI pipeline on the GPU. We demonstrated the feasibility of the approach as well as the improved performance that it affords. We emphasized the key features of the implementation to facilitate its reproduction by other researchers and programmers. While the full HDRI pipeline may contain other operations such as the camera response recovery, image alignment, ghost removal, etc., the skeletal implementation provided here can serve as a basis to implement these other functionality as well.[5]

References

1. Akyüz, A.O., Fleming, R., Riecke, B.E., Reinhard, E., Bülthoff, H.H.: Do hdr displays support ldr content? a psychophysical evaluation. ACM Trans Graph **26**(3), 38:1–38:7 (2007)
2. AMD: Amd’s 10-bit video output technology (2008). http://developer.amd.com/gpu_assets/10-Bit.pdf
3. Artusi, A., Bittner, J., Wimmer, M., Wilkie, A.: Delivering Interactivity to Complex Tone Mapping Operators. Eurographics Association, Leuven (2003)
4. Aydin, T.O., Mantiuk, R., Myszkowski, K., Seidel, H.P.: Dynamic range independent image quality assessment. ACM Trans Graph **27**(3), 69:1–69:10 (2008). doi:10.1145/1360612.1360668
5. Banterle, F., Artusi, A., Debattista, K., Chalmers, A.: Advanced High Dynamic Range Imaging: Theory and Practice. CRC Press (AK Peters), Natick, MA (2011)
6. Battiato, S., Castorina, A., Mancuso, M.: High dynamic range imaging for digital still camera: an overview. J. Electr. Imaging **12**(3), 459–469 (2003). doi:10.1117/1.1580829. http://link.aip.org/link/?JEI/12/459/1
7. Cohen, J., Tchou, C., Hawkins, T., Debevec, P.: Real-Time high dynamic range texture mapping. In: Gortler, S.J., Myszkowski, K. (eds.) Rendering, Techniques 2001, pp. 313–320 (2001)
8. Debevec, P.: Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography, pp. 189–198. In: SIGGRAPH ’98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, ACM Press, New York, NY, USA (1998)
9. Debevec, P.E., Malik, J.: Recovering high dynamic range radiance maps from photographs. In: SIGGRAPH 97 Conference Proceedings, pp. 369–378 (1997)
10. Devlin, K.: A Review of Tone Reproduction Techniques. Technical Report CSTR-02-005, Computer Science, University of Bristol (2002)
11. Goodnight, N., Wang, R., Woolley, C., Humphreys, G.: Interactive time-dependent tone mapping using programmable graphics hardware. In: Proceedings of the 13th Eurographics Workshop on Rendering, Eurographics Association, pp. 26–37 (2003)
12. Goodnight, N., Wang, R., Humphreys, G.: Computation on programmable graphics hardware. Comput. Graph. Appl. IEEE **25**(5), 12–15 (2005). doi:10.1109/MCG.2005.101

[5] The full implementation of the presented algorithms are made available at: http://www.ceng.metu.edu.tr/~akyuz/hdrgpu/index.html.

13. Hassan, F., Carletta, J.: An fpga-based architecture for a local tone-mapping operator. J. Real-Time Image Process. **2**, 293–308 (2007a). doi:10.1007/s11554-007-0056-7
14. Hassan, F., Carletta, J.E.: A real-time fpga-based architecture for a reinhard-like tone mapping operator, pp. 65–71. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, GH '07 (2007b)
15. ITU: ITU-R Recommendation BT.709-5, Parameter Values for the HDTV Standards for Production and for International Programme Exchange. ITU (International Telecommunication Union), Geneva (2002)
16. Khronos Group: Opengl 4.2 specification (2012). http://www.opengl.org/registry/doc/glspec42.core.20120427.pdf
17. Krawczyk, G., Myszkowski, K., Seidel, H.P.: Perceptual effects in real-time tone mapping. In: Proceedings of the 21st Spring Conference on Computer Graphics, ACM, New York, NY, USA, SCCG '05, pp. 195–202 (2005). doi:10.1145/1090122.1090154
18. Mantiuk, R., Krawczyk, G., Mantiuk, R., Seidel, H.P.: High dynamic range imaging pipeline: perception-motivated representation of visual content. In: Rogowitz, B.E., Pappas, T.N., Daly, S.J. (eds.) Proceedings of SPIE, Human Vision and Electronic Imaging XII, SPIE, Vol. 6492, San Jose, USA (2007)
19. Mitsunaga, T., Nayar, S.K.: Radiometric self calibration. In: Proceedings of CVPR, Vol. 2, pp. 374–380 (1999)
20. Reinhard, E.: Parameter estimation for photographic tone reproduction. J. Graph. Tools **7**(1), 45–51 (2003)
21. Reinhard, E., Stark, M., Shirley, P., Ferwerda, J.: Photographic tone reproduction for digital images. ACM Trans. Graph. **21**(3), 267–276 (2002)
22. Reinhard, E., Khan, E.A., Akyüz, A.O., Johnson, G.M.: Color Imaging: Fundamentals and Applications. A K Peters, Wellesley, MA (2008)
23. Reinhard, E., Ward, G., Pattanaik, S., Debevec, P.: High Dynamic Range Imaging: Acquisition, Display and Image-Based Lighting, 2nd edn. Morgan Kaufmann, San Francisco (2010)
24. Robertson, M., Borman, S., Stevenson, R.: Estimation-theoretic approach to dynamic range enhancement using multiple exposures. J. Electr. Imaging **12**(2), 219–228 (2003)
25. Seetzen, H., Heidrich, W., Stuerzlinger, W., Ward, G., Whitehead, L., Trentacoste, M., Ghosh, A., Vorozcovs, A.: High dynamic range display systems. ACM Trans. Graph. **23**(3), 760–768 (2004)
26. Williams, L.: Pyramidal parametrics. ACM Comput. Graph. **17**(3), 1–11 (1983)
27. Wyszecki, G., Stiles, W.S.: Color Science: Concepts and Methods, Quantitative Data and Formulae. Wiley, New York (2000)