Graz University of Technology

Institute for Computer Graphics and Vision

Institute of Networks and Distributed Systems

Master Thesis

Real Time High Dynamic Range Video on the GPU

Lorenz Jaeger

Graz, Austria, October 2014

Thesis supervisors

Prof. Dr. Dieter Schmalstieg

Prof. Dr. Paal Halvorsen

To Heimo Jaeger

Life is hard and so am I

Mark Oliver Everett

Abstract

The goal of this work was to show that real-time HDR is possible for large resolutions. We implemented algorithms related to HDR content processing because this can greatly improve the image quality of a video under difficult lighting conditions. For practical use in live video capturing it was important for our algorithms to work in real-time. Therefore we decided to implement existing algorithms in Nvidia's CUDA. After selecting and implementing multiple algorithms we extended an existing panoramic image processing pipeline. This pipeline is used to capture panoramic videos of the playing field of a football stadium located in Tromsø, Norway. We performed multiple tests on our extension. These consisted of detailed performance analysis and an evaluation of the visual pleasantness of the output in a small user study. We proved that real-time high resolution HDR Video is possible and that it improves image quality.

Keywords. High Dynamic Range (HDR), panorama video, real-time,

Acknowledgements

I would like to thank my supervisors, Prof. Schmalstieg and Prof. Halvorsen, for their guidance and advice. I owe gratitude to Patrick Kaspar, Bernhard Kerbl and Ragnar Langseth, without whose support this work would not be the same. Furthermore, I want to thank my friends, family, and of course Barbora Orlická, for their encouragement throughout the whole time.

Contents

1 Introduction **1**

2 High Dynamic Range **3**

2.1 Introduction to HDR . . . 3

2.2 Radiance mapper . . . 4

2.2.1 Luminance & Key . . . 5

2.2.2 Response Function . . . 6

2.2.3 Weight Function . . . 6

2.3 Tone mapper . . . 6

2.3.1 Global vs. Local . . . 7

2.4 Pipeline . . . 8

2.4.1 Pipeline Modules . . . 8

2.4.2 Prerequisites for HDR . . . 10

2.4.3 Integration into Pipeline . . . 11

3 Related Work **13**

3.1 General Purpose HDR . . . 13

3.1.1 Radiance mapping . . . 13

3.1.2 Tone mapping . . . 15

3.2 HDR for video . . . 17

4 Algorithms **19**

4.1 Radiance mapper . . . 19

4.1.1 Debevec . . . 19

4.1.1.1 Procedure . . . 19

4.1.2 Robertson . . . 21

4.1.2.1 Procedure . . . 21

4.1.3 Tocci . . . 23

4.1.3.1 Procedure . . . 23

4.1.3.2 Comparison of Radiance mappers . . . 24

4.2 Tone mapper . . . 24

4.2.1 Ward . . . 24

4.2.1.1 Procedure . . . 25
4.2.2 Larson . . . 26
4.2.2.1 Procedure . . . 26
4.2.3 Reinhard . . . 27
4.2.3.1 Procedure . . . 27
4.2.3.2 Comparison of Tone mappers . . . 29
4.2.4 Weight Functions . . . 29
4.2.4.1 Debevec . . . 29
4.2.4.2 Robertson . . . 30
4.2.4.3 Mitsunaga . . . 30
4.2.4.4 Linear . . . 30
4.2.4.5 Comparison of Weight functions . . . 31

5 Implementation 33
5.1 Implementation in C++ . . . 33
5.2 Implementation in CUDA . . . 34
5.2.1 CUDA concepts . . . 35
5.2.2 Common . . . 36
5.2.3 Debevec . . . 37
5.2.4 Robertson . . . 37
5.2.5 Tocci . . . 37
5.2.6 Ward . . . 39
5.2.7 Larson . . . 39
5.2.8 Reinhard . . . 41

6 Performance Measurement and Study 43
6.1 Performance . . . 43
6.1.1 Set-Up . . . 44
6.1.2 Execution Times . . . 44
6.1.3 Detailed Evaluation . . . 45
6.1.3.1 Legend . . . 46
6.1.3.2 Debevec . . . 47
6.1.3.3 Robertson . . . 48
6.1.3.4 Tocci . . . 48
6.1.3.5 Ward . . . 51
6.1.3.6 Larson . . . 53
6.1.3.7 Reinhard . . . 59
6.1.3.8 Common . . . 68
6.1.3.9 Overview . . . 69
6.2 Visual Comparison . . . 70
6.2.1 Set-Up . . . 70

6.2.2 Proceedings 70
6.3 Results of subjective study 71
6.4 Performance vs. Visual Result trade-off 75
6.5 Final Result 76

7 Conclusion 81
7.1 Future Work 81
7.2 Concluding Remarks 82

A Acronyms and Symbols 83

Bibliography 84

List of Figures

2.1 Capturing multiple exposures 4
2.2 HDR example 5
2.3 Local tone mapper example 8
2.4 Pipeline hardware set-up 8
2.5 Pipeline software set-up 8
2.6 Pipeline camera set-up 10
2.7 LDR example 10

6.1 Execution times of pipeline 45
6.2 Execution times of `HDR` module 45
6.3 Execution times of calibration 48
6.4 Evaluation of our implementation of Debevec and Malik's algorithm 49
6.5 Evaluation of our implementation of Robertson et al.'s algorithm 50
6.7 Evaluation of our implementation of Tocci et al.'s algorithm 53
6.8 Evaluation of our implementation of Ward's algorithm 54
6.9 Evaluation of our implementation of the sub step compute-foveal-image in Larson et al.'s algorithm 56
6.10 Evaluation of our implementation of the sub step calculate-histogram in Larson et al.'s algorithm 58
6.11 Evaluation of our implementation of the sub step apply-histogram in Larson et al.'s algorithm 59
6.12 Execution times of pipeline with reinhard 60
6.13 Execution times of `HDR` module with reinhard 60
6.14 Evaluation of our implementation of the sub step create-radiance-image in Reinhard et al.'s algorithm 63
6.15 Evaluation of our implementation of the sub step fft-forward in Reinhard et al.'s algorithm 64
6.16 Evaluation of our implementation of the sub step about processing complex numbers in Reinhard et al.'s algorithm 66
6.17 Evaluation of our implementation of the sub step applying the formula introduced in Section 4.2.3.1 in Reinhard et al.'s algorithm 67

6.18 Evaluation of our implementation of parallel reduction to recover log-average brightness 69
6.19 Results of the subjective user study 75
6.20 Output images acquired using different combinations of algorithms 79

List of Tables

4.1 Radiance mapper Features 24
4.2 Tone mapper Features 29
4.3 Weightfunction Features 31

6.1 Execution times of all modules in different configurations of `HDR` algorithms 46
6.2 Execution times of all modules with different number of Gauss Kernels used by the radiance-mapper proposed by Reinhard et al. in ms 61
6.3 Execution times of all kernels in ms 70

Chapter 1

Introduction

HDR (high dynamic range) video is a young field of research in the area of video processing. There have been significant advances in HDR processing of static images in the past. Transferring these insights into real-time video capture has not been attempted widely. RED inc.* implemented real-time HDR capturing in their EPIC camera series. Here, only the capturing is performed in real-time. Processing the image data in order to be displayed on an LDR (low dynamic range) device still requires a time consuming post-processing. Our goal was to implement all the necessary operations for capturing and displaying HDR content in real-time. We used this implementation as a module to extend an existing pipeline. We have set up an array of cameras that overlook a football stadium located in Tromsø, Norway. By stitching the input of the cameras together we obtain a high resolution panoramic video of the playing field. This video is later used for various post-processing steps including player tracking, and processing for live streams. Tromsø is located north of the Arctic Circle, this creates special conditions. Firstly, the sun there never rises very high. This results in long shadows covering the playing field. Secondly, snow is present for most of the time in the surroundings of the stadium. The difference in brightness between shade, and snow illuminated by the sun cause difficult light conditions. These problems are usually solved by manually adjusting the exposure of the cameras capturing the video. Since our implementation is a fully automatic system, this is not a viable option.

Previous attempts at capturing HDR real-time video were severely limited in resolution because they were bounded by the floating point performance of the CPU . Therefore, we decided to implement multiple existing HDR -processing algorithms in CUDA. The

*http://www.red.com/

processing power available on a GPU enables us to work with higher resolutions. We then performed a comparison of the performances of the algorithms and picked the most suitable one to be used in the pipeline.

This work is structured as follows. Chapter 2 introduces HDR and the aforementioned pipeline. In Chapter 3 we discuss existing solutions for HDR processing. Out of these related works we selected six algorithms to implement. These algorithms are discussed in more detail in Chapter 4. How we implemented these algorithms in CUDA is then discussed in Chapter 5. In Chapter 6 we perform a visual and performance evaluation of our implementation. We conclude the work with Chapter 7 where we present the best combination of algorithms.

Chapter 2

High Dynamic Range

Contents

2.1 Introduction to HDR 3

2.2 Radiance mapper 4

2.3 Tone mapper 6

2.4 Pipeline 8

2.1 Introduction to HDR

Brightness that occurs in the real-world can span twelve orders of magnitude. The range of the human visual perception is nine orders of magnitude. If this range is exceeded, the eye adapts using various adaptation techniques*. A film inside a camera can capture up to seven orders of magnitude. Photographic paper can reproduce up to four orders of magnitude [26]. These facts lead to multiple problems when one tries to capture and represent real world luminance values. First, a camera (analogue or digital) cannot capture the full range present. This leads to either under- or overexposed areas in the image. Second, the devices used to display the captured image exhibit an even lower range than the capturing devices resulting in additional loss of information. These problems can be solved using `HDR` (high dynamic range) processing which consists of two steps. The first is called radiance mapping and deals with capturing and storing real world luminance. The second is called tone-mapping which tries to map the

*"Sensory Reception: Human Vision: Structure and Function of the Human Eye" Encyclopædia Britannica, vol. 27, 1987

captured luminance to a range that can be displayed by an LDR (low dynamic range) device.

2.2 Radiance mapper

A radiance mapper tries to capture the existing real world luminance using devices that typically can not capture the entire range at once. An exemplary situation could be a photographer taking a picture inside a church where the bright light shining through the windows contrasts with the dark interior. If the photographer tries to capture both the details of the interior and of the stained-glass windows, one of the following problems occurs:

1. The intricate details of the windows are captured. The rest of the interior is underexposed and barely visible.

2. The inside of the church is visible, but the details in the window are not. The glass is a featureless white area of saturation.

3. The inside of the church is underexposed and not visible. On the other hand the window is overexposed and therefore not visible either.

The different cases mentioned above can be seen in Figure 2.1.

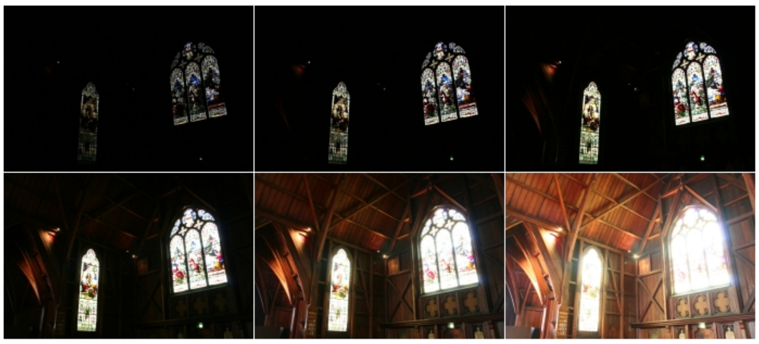


Figure 2.1: Example for capturing multiple exposures. Source: wikipedia.org

The most common approach to solve this problem is to take multiple images with different exposure times. Those multiple exposures are then combined into a single image by a radiance mapper. An example of this process can be seen in Figure 2.2. The way this is exactly accomplished differs greatly from algorithm to algorithm and will be discussed in further detail in Section 4.

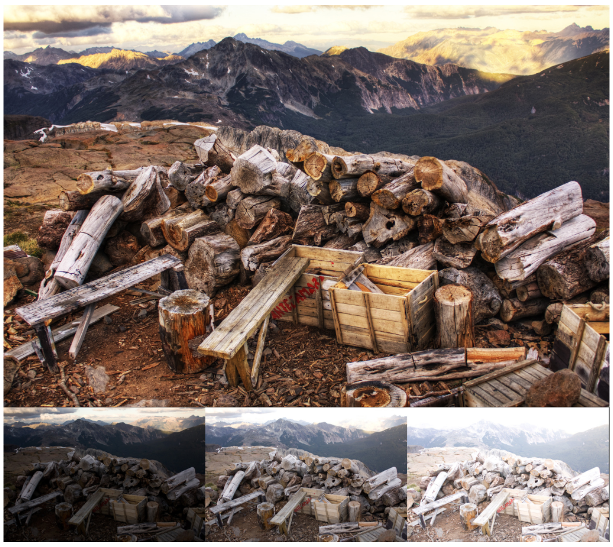


Figure 2.2: Example for merging multiple exposures and performing tone mapping on the resulting HDR image. Source: wikipedia.org

LDR images are typically stored with eight bits per colour channel, resulting in 24-bit colour-depth. This is too little for HDR purposes. Here the value for each colour channel is stored as a floating point value. This leads to increased memory requirements.

2.2.1 Luminance & Key

Luminance is defined as $\frac{lm}{sr \cdot m^2}$ with lm = luminous flux, sr = steradian angle. This represents the amount of light emitted in a certain direction over an angle [28]. This light can have two possible sources. Either the material is emitting light, or it is reflecting incoming light. The colloquial term for luminance is brightness. We will use these two terms interchangeably from here on. The human eye perceives changes in brightness on a $\log_{10}$ scale. This means that luminance of an object has to increase by an order of magnitude to be perceived as twice as bright.
The key of a scene is the real world brightness that is mapped to the middle brightness value in the output format. This means that this real world luminance level represents the middle of the available range. It also describes the overall perceived brightness of the output image. Low key images tend to be very dark, whereas high key images are much brighter. Selecting the right key in traditional photography is often based on the zone system proposed by Adams in [1], [2] and [3]. Here the possible output range is separated into 11 zones. The photographer then tries to map the perceived middle brightness of the scene to the middle zone in the output. In our algorithms the user can set a desired key

value and all other values are then scaled around this middle value. This enables the user to set the tone of the image.

2.2.2 Response Function

The response function is a look-up table that maps real world luminance to stored luminance. In an idealized and simplified model, if one photon hits a pixel on the image-capturing device this pixel returns a brightness of one photon. If two photons hit a pixel, it returns a brightness of two photons. This is a linear response curve. Physical film used in analogue cameras does not exhibit a linear response curve. It is S-shaped. In the lower range a threshold of brightness has to be exceeded in order to be registered, and towards the upper end of the brightness range saturation effects alter the response. Technically, digital photo sensors have an almost linear response, but camera manufacturers modify that response to mimic physical film. Each manufacturer uses a different mapping. Therefore determining the individual response curve of a camera is part of some of the algorithms. We implemented the response function in a discrete fashion. For each possible input value (0-255) in each colour-channel we associate an output brightness. The range of the output brightness spans multiple orders of magnitude as opposed to the range of the input.

2.2.3 Weight Function

As previously mentioned, many radiance mappers merge multiple images with different exposures into a single output image. The pixels of each image are often weighted to prefer exposures with '*useful*' information. This means that the pixel is not all black or all white but contains some sort of information. These extreme values of all black or all white usually mean that a pixel was under- or overexposed, respectively. Therefore it contains no information of interest. What is considered to be '*useful*' data varies between different authors. This will be discussed in further detail in Section 4.2.4.

2.3 Tone mapper

The image produced by a radiance mapper can span up to twelve orders of magnitude. Printing paper can only represent up to four orders of magnitude. Computer monitors also span only a range of four orders of magnitude. The tone mapper tries to compress the existing HDR data into a range suiting the capabilities of the display device. A naïve approach would be to normalize the HDR data. This typically leads to undesirable compression artifacts. A single very high or low value can lead to strong compression where most of the image content gets compressed into a single value.

2.3.1 Global vs. Local

The related literature distinguishes between two kinds of tone mappers:

1. **Global operators:** In this category of tone mappers an operation is applied onto the image as a whole. The operation performed is the same for each pixel. No local neighbourhood is taken into account. If two input values are the same, the output value will be the same too. This kind of operator has the advantage of preserving global contrast in the output image. These global approaches usually have shorter execution times. This is because the complex operations required only need to be performed once per image, not once per pixel. The compression of each pixel is often implemented as a simple look-up operation.

   Properties:

   + fast

   + conserves global contrast

   – may lead to over compression if dynamic range is high

2. **Local operators:** Tone mappers in this category take the local neighbourhood of each pixel into account. This means that two identical input values can result in different output values. These operators are usually more computationally demanding since complex operations have to be performed for each pixel. Reinhard et al. [26] state that human vision is most sensitive to local contrast changes, so these operators can lead to more desirable results according to Yoshida et al. [35]. This is because local operators attempt to maximize local contrast and ignore global contrast. One downside of this approach are the artifacts appearing at contrast edges called ”halos” as can be seen in Figure 2.3.

   + good local contrast

   + extreme values do not degrade whole image

   – slow

   – can produce artifacts/halos

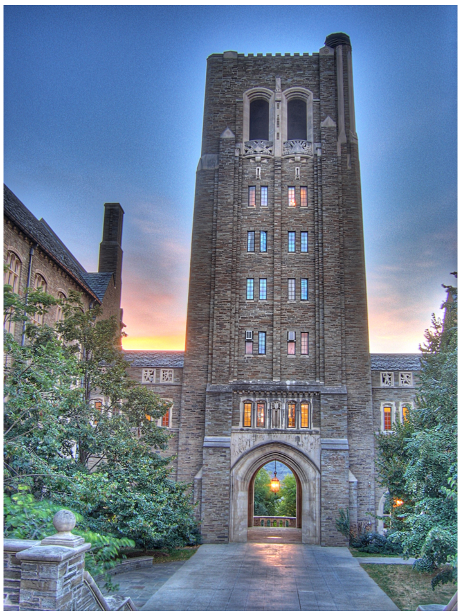


Figure 2.3: Example of a local tone mapper. Source: wikipedia.org

2.4 Pipeline

2.4.1 Pipeline Modules

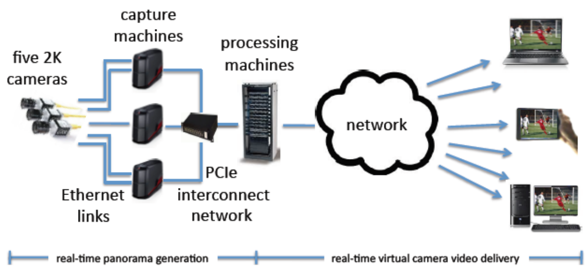

capture machines
five 2K cameras
processing machines
network
Ethernet links
PCIe interconnect network
real-time panorama generation
real-time virtual camera video delivery

Figure 2.4: Overview of the hardware set-up of the panoramic capture pipeline. Source: [10]

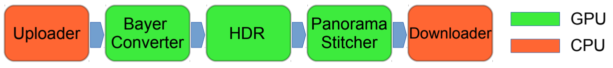


Figure 2.5: Overview of the modules in the pipeline

In the following section we describe the individual modules that make up the previously mentioned pipeline. The goal of this pipeline is to capture and distribute a real-time panoramic video of the playing field of a football stadium. We extend this pipeline with our HDR processing module.

recorder: This module runs on multiple machines in a distributed fashion. It records the raw output of five cameras overlooking a football stadium. The cameras are mounted on a custom made rack as seen in Figure 2.6. This module has to synchronize the shutters of all cameras, manage white balance and detect dropped video frames. The processing load required for these operations is too high to be efficiently handled by a single machine, thus requiring a distributed implementation. Synchronisation between the machines involved is achieved using a PCI express local area network provided by Dolphin [*]. A more detailed view of the set-up can be seen in Figure 2.4.

Uploader: This module is responsible for uploading the captured images to the device memory space of the GPU. This task is simple, but copying data to and from the GPU is a costly operation. This module therefore attempts to minimize the amount of data transferred.

Bayer Converter: Here the incoming raw data is converted. Initially, each pixel only consists of a single colour channel. During this process, four of those pixels are combined to result in a single RGB pixel. The demosaicing pattern used was first proposed by Bayer [4]. This pattern is used in most digital image capturing devices. Here the contribution to the final colour consists of one red, one blue, and two green pixels. The colour green is represented more because the human vision is most sensitive to this colour.

Panorama Stitcher: This module is responsible for merging the individual images of the five cameras into one single, continuous image. Each FOV (field of view) of the five cameras overlaps with the FOV of the neighbouring cameras. This way an overlapping area between each of the images is created. The module performs dynamic seam creation that merges the images in those areas, trying to avoid visible seams. For this purpose an approach proposed by Xiong and Pulli [33] was adapted to be executed on the GPU .

Downloader: This module performs the inverse operation of the Uploader module. It copies the processed image data from the device memory located on the GPU to host memory to be accessed by the CPU .

[*]http://www.dolphinics.com/

Encoder: The processed video frames are encoded as a video stream by the Encoder module. This stream is then either stored on a hard drive to be used later, or immediately streamed to viewers.

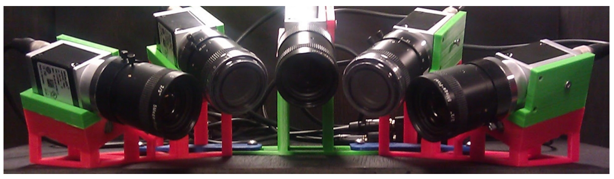


Figure 2.6: Mounted camera array Source: [10]

2.4.2 Prerequisites for HDR

In this section we will briefly discuss why we decided to implement an `HDR` module in our pipeline. The stadium where the cameras are currently installed has two properties that make capturing videos challenging:

1. It is located in Tromsø, Norway. This town is situated north of the Arctic Circle. Because of this, the sun does not rise very high, causing long shadows.

2. The roof of the stadium only covers the audience seats but not the playing field. This circumstance coupled with the low sun causes parts of the playing area being covered in shadow and other parts being brightly lit by the sun. This can be seen in Figure 2.7.

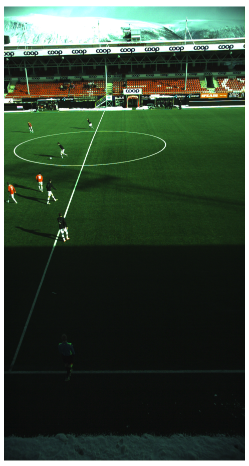


Figure 2.7: Example image that shows shadow covering play area.

Television broadcasters trying to film a football game face the same problems, but they employ people to man the cameras. This enables them to dynamically adjust the exposure of the camera so show only the area of interest. Our system on the other hand works without human intervention and covers the whole playing field instead of only an area of interest.

Difficult light conditions coupled with the need for an autonomous solution lead us to the conclusion, that we must perform HDR video capturing followed by tone mapping. The resulting contrast compression enables us to make the entire play area visible at once, independent of present light conditions.

2.4.3 Integration into Pipeline

The individual modules of the pipeline are loosely coupled with each other. We implemented inter module communication based on the producer-consumer architectural model. Each module only knows of its predecessor within the pipeline. Each time a module is finished with an operation it requests new data to process from its predecessor. This loose coupling enables us to change the composition of modules within the pipeline easily. One such use-case would be the position of the HDR module in relation to other modules. There are two positions that are viable. Either HDR is performed before or after the stitching module. Executing the HDR -module after the Stitching module causes the input, which needs to be processed by the HDR -module, to be slightly smaller. Executing the HDR module before the Stitching module yields the following advantages: The stitching module has to perform its operation only once on the merged data, not on each exposure individually. Furthermore the stitching module has more data in a useful range to use for seam creation. Therefore we decided to perform HDR related operations before the stitching module.

Data exchange between individual modules is facilitated using two data structures. The image data is transmitted using CUDA-arrays. This data-structure is an opaque memory layout that offers spatial based caching. Since many operations on the images depend on their local neighbourhood this choice exhibits advantages over plain, linear global memory. The properties of CUDA-arrays will be discussed in more detail in Section 5.2.1. Metadata, like frame number and exposure time for each image, is stored in a C++ `struct`. When multiple images are fused together, the related meta data is changed accordingly.

Chapter 3

Related Work

Contents

3.1 General Purpose HDR 13
3.2 HDR for video 17

3.1 General Purpose HDR

In this section we will discuss general-purpose `HDR` algorithms that are applicable in a wide range of situations. We try to divide existing literature into two categories. First, publications that aim to create `HDR` images, and second the ones that deal with displaying `HDR` content. This clear distinction is not always possible since some authors tackle both problems at once. For these cases we categorize the work based on what we deem to be the crucial contribution of it.

3.1.1 Radiance mapping

Yamada et al. [34] propose a very simple way of combining multiple exposures. Instead of merging the pixel values of different exposures, only the value of a single exposure is used. The algorithm starts off with the value of the pixel in the longest exposure. It checks if the pixel is saturated or not. If the pixel is not saturated, this value is used. If on the other hand it is saturated, the next shorter exposure is considered. The same check of saturation is applied to this new pixel value. This process is iteratively continued until all pixels have non saturated values. The approach proposed is simple and quick, but since no merging is performed, image noise can become an issue that has derogatory effects on the output image.

Tocci et al. [29] extend the basic idea proposed by Yamada et al.. Under certain circumstances pixels of different exposures are merged instead of using a single value. The algorithm takes the local neighbourhood of each pixel into account. Again the pixel values of the longest exposure are considered first. If the pixel itself or one of its neighbours is above a certain saturation threshold, the value is merged with the value from the next shorter exposure. If no saturation is observed, the pixel is used as it is. The weight function used is based on the number of saturated pixels in the local neighbourhood, and only penalizes pixels that are very close to over- or undersaturation. This process is performed iteratively until there are no saturated pixels left, or all available exposures have been considered.

Mann and Mann [21] were the first to propose recovering the response function of the camera used. Authors in the past assumed direct access to the capturing chip. They were able to assume a linear response, since no post-processing had been performed on the pixel values. Reading RAW image data is not always possible and too slow for video capturing. Therefore the authors introduce a way to approximate the response function of a camera. This “calibration” has to be performed once in the lifetime of the camera used.

Debevec and Malik [7] build upon the idea introduced by Mann and Mann [21] and propose multiple novel concepts in their landmark paper. Different exposures of the scene are merged together, instead of using the pixel value of a single exposure. In order to facilitate this merging, they introduce weight functions that give different priorities to the exposures being merged. The weight function proposed by Debevec and Malik is a triangular function. Here values towards the middle of the range get the highest weight and therefore the most influence on the output. Debevec and Malik argue that pixels close to black and white tend to contain less useful information because they are likely to be over- or underexposed. This claim is refuted by other papers on this list, namely [27], [12] and [29]. Furthermore the authors introduce a way to recover a camera’s response function. For this, a number of sample images with known exposures are needed. This operation only needs to be performed once for each camera used.

The approach put forth by Robertson et al. [27] is similar to the one proposed by Debevec and Malik. The algorithm also contains a ”calibration” process for each camera used. An arbitrary response function is recovered from a set of sample images at different exposures. Unlike the weight function used by Debevec and Malik, Robertson et al. suggest a dynamic weight function. It is based on the confidence that a value was represented correctly using the given response function. When merging multiple exposures the formula proposed by Robertson et al. gives more weight to higher exposures because the authors argue that those are more likely to contain useful information.

The algorithm suggested by Mann and Picard [22] is very similar to the one proposed by Robertson et al.. In both a response function is recovered from sample images prior to performing the radiance mapping step. Multiple exposures are merged using a weight function that is based on confidence in the correctness of a measurement. The main difference in the approach of Mann and Picard to the one of Robertson et al. is that the response function is assumed to be parametric, not arbitrary. This limits accuracy and can not represent every digital image sensor.

Mann [20] introduces multiple ideas in this work. First he dismisses the idea to perform homomorphic filtering, but instead to work right in the real-world brightness domain. He argues that many devices perform a form of logarithmic compression themselves. Performing another logarithmic compression on top of that can lead to undesirable results. Furthermore, he proposes using comparametrics. Here, one exposure is chosen as a base image. Images with different exposures are represented as their difference to that base image. By using this form or representation it is possible to split the process of recovering a camera's response function into two steps. Not all operations require a full response function, so only the first step needs to be performed, thus requiring fewer computations.

Granados et al. [12] discuss various weight functions and compare them with each other. Weight functions are used by some radiance mappers when merging multiple exposures. Based on the saturation of a pixel in a certain exposure the pixel is given mor,e or less weight in the final, merged image. Debevec and Malik, for example, argue that the most useful data for a pixel is in the middle range of the pixel. Granados et al. argue that most of the useful information is towards the upper end of the range before the pixel is saturated. Granados et al. propose a weight function of their own, which takes various noise terms into account. Because of this property the weight function needs to be "calibrated" to the camera prior to use. This way the individual noise properties of the camera are incorporated.

3.1.2 Tone mapping

One of the first works concerning tone mapping was written by Ward [32]. His proposed tone mapping solution is a global operator (See Section 2.3.1). In his approach the author takes the brightness capabilities of the displaying medium into account. Each pixel is scaled with a global scaling factor that is based on the overall brightness of the incoming image and the highest possible output brightness. The other values in the formula proposed by Ward are magic numbers that were determined by the author in an empirical fashion. These numbers represent the relation between the minimum discernible difference and the input brightness. The goal of this formula is to preserve the relative contrast of the HDR input image. One downside of this algorithm is that compression is performed linearly. If the dynamic range is very high, this can lead to undesirable results. Only the very bright and dark parts are visible while most of the details in the middle range are lost due to over-compression.

Reinhard et al. [26] propose a sophisticated local tone mapping operator. They try to simulate a technique called “dodging & burning” which is well established in traditional photography [3]. When using this technique, different areas of the image are exposed longer, or shorter when transferring the image from the negative to photographic paper. This way areas that are too dark on the negative are made brighter in the resulting image and overly bright areas are darkened. To transfer this technique to digital images the authors propose to perform dodging & burning on a per pixel basis. For this, the local neighbourhood of each pixel is taken into account by using an average of the neighbourhood. If the neighbourhood is dark, then the pixel is made darker, thus performing a “dodging” operation. On the other hand, if the neighbourhood is very bright, then the pixel itself is made brighter.

Larson et al. [18] suggest a global tone mapping operator that is based on histogram equalisation. As opposed to naïve equalisation, they propose “histogram adjustment”. Using this approach human contrast thresholds and the visual acuity of the eye are considered when adjusting the histogram. The authors also simulate some of the perceptual effects of the human visual system observed by Krawczyk et al. [17].

Drago et al. [8] propose a hybrid approach between local and global tone mapping. First, a global brightness adjustment is performed based on a key brightness value set by the user. This step is similar to a key adjustment performed in [27]. Then, a local operator is applied to each pixel. Using this operator, each incoming HDR pixel is compressed by applying a logarithmic operation. Depending on the brightness of the incoming pixel the base of the logarithm is changed to either perform more, or less compression. Logarithmic compression approximates how the human eye perceives brightness. This form of compression leads to a “natural” look of the compressed image.

An approach that is neither a local nor a global approach, but instead a gradient domain based solution is proposed by Fattal et al. [9]. Here the gradients in brightness for the whole image are calculated. They are calculated in multiple resolutions to represent different frequencies of contrast changes. This way local noise, and the overall, more global, brightness changes of the image are accounted for. The higher the brightness gradient the more compression is performed. This way small contrast changes are not changed and big differences are adequately compressed into the displayable range. According to Yoshida et al. [35] humans are most sensitive to small, high frequency contrast changes. These are preserved when using this technique.

Mantiuk et al. [23] propose a novel approach where the light conditions surrounding the displaying device are accounted for. The key brightness of the output image is adjusted depending on the ambient luminance. The operator proposed is a global operator. When performing the brightness adjustments to the image, the contrast behaviour of the human eye is incorporated.

Krawczyk et al. [17] do not propose a tone mapping operator themselves. Instead they observe various phenomena that occur in the human visual system. For example how it adjusts to changes in brightness and how very dark and bright scenes are perceived differently. The insights published in this work are used by some of the other authors mentioned in this section. The findings include, but are not limited to: glare, blur, reduced colour differentiation in dark areas and local contrast sensitivity. Their work should be seen as supplementary work for the previously mentioned tone mappers.

3.2 HDR for video

In this section we introduce algorithms that specifically deal with performing high dynamic range (HDR) operations on videos. This implies trying to maintain consistency across multiple frames of the video. Benoit et al. [5] propose a tone mapping algorithm that closely models how the human eye handles brightness compression. The algorithm consists of multiple steps that each mimic a part of the eye. One step of this algorithm performs temporal and spatial filtering. This temporal property is well suited for HDR video content. Instead of optimizing for each frame independently, some knowledge of previous frames is preserved. This enables smoother transitions between frames in which light conditions change.

Guthier et al. [13] introduce a way of handling changing light conditions when capturing a video. A tone mapping operator always tries to achieve the perfect result for each frame and does not take any information from the previous frames into account. This can lead to flicker over the duration of a video, even if the light conditions change only marginally..

The authors propose a smooth brightness transition between frames. This is achieved by limiting the absolute brightness change possible between consecutive frames. The target brightness set by the tone mapper is reached gradually over the course of multiple frames. This corresponds to how the human eye adjusts to changes in brightness. Adaptation in the eye is not instant, but instead is performed gradually over a short period of time.

Another publication by Guthier et al. [14] gives an overview of a real-time HDR video pipeline that the authors have created. This work covers multiple previously published papers by the authors. First, Guthier et al. tackle the capturing of the individual LDR images. In their approach only a single picture is taken and analysed. An algorithm detects any areas that are over or under exposed. These areas are captured again using a different exposure time during a step called "partial re-exposure". This technique helps improve capture time, since only one full image has to be read. All subsequent reads are performed for smaller parts of the image, thus reducing the amount of data that needs to be processed. The authors use a hand-held camera that can move between capturing different exposures. This circumstance necessitates image registration between individual captures to detect any camera movement that might have occurred in the meantime. The authors propose image registration based on histograms. The next two steps in the pipeline are radiance mapping, and tone mapping. Here the authors are using algorithms proposed by Debevec and Malik [7], and Larson et al. [18] respectively. Finally, the authors suggest applying flicker reduction as a post-processing step on the output. This approach had been previously introduced by them in [13].

Our approach to HDR was greatly influenced by this last paper by Guthier et al.. Because the input we process is more static, we only perform a subset of the operations mentioned.

Chapter 4

Algorithms

Contents

4.1 Radiance mapper 19
4.2 Tone mapper 24

In the following chapter we will give an overview of the algorithms we chose to implement for our HDR processing module.

4.1 Radiance mapper

In this section we will introduce various algorithms that deal with creating HDR content. This is the first step in our HDR module.

4.1.1 Debevec

We chose this algorithm because it is very robust. By merging multiple exposures the resulting HDR image is less prone to noise artifacts that are present in a single image. The assumption of an arbitrary response function gives this algorithm a wide range of applications and ensures compatibility with different kinds of cameras. Most of the tone mapping algorithms we encountered used HDR data generated by using the approach introduced by Debevec and Malik. This is a highly cited work in the area of HDR processing.

4.1.1.1 Procedure

The process for creating HDR images in this algorithms consists of two major steps, one off-line step and an on-line step. The off-line step needs to be performed once in the lifetime of the camera used. The on-line step is performed for every frame of the video.

Off-line Step: The goal of the off-line step is to recover the arbitrary response function of the camera being used. The input required for this process consists of two types of data. One is multiple images of the same scene with different exposure times. The other is the different exposure times of each of these images. All following steps could be performed for every pixel of the incoming images. This is not necessary to get satisfying results and in order to speed up computation only a sampled set of pixels are used in the computations. For the sampled pixel positions their values in different exposures are read. These values are used to solve a `SVD` system. By minimizing the following formula the response function can be recovered.

$$\mathcal{O} = \sum_{i=1}^{N} \sum_{j=1}^{p} [g(Z_{ij}) - lnE_i - ln\Delta t_j]^2 + \lambda \sum_{z=Z_{min}+1}^{Z_{max}} g''(z)^2$$

where:

N	=	Number of pixel locations
P	=	Number of photographs
$g()$	=	response function
Z_{ij}	=	Value of incoming `LDR` values per colour channel (0 - 255)
E_i	=	Real world luminance of pixel i
t_j	=	exposure time of photograph j in seconds
λ	=	smoothness term

By using the well known Gauss-Seidl algorithm [11] to solve this `SVD` , the response function of the camera can be recovered

On-line step: The on-line step has to be performed for every frame of the video. Here the goal is to merge multiple incoming `LDR` images into a single `HDR` output image. For each pixel position the corresponding values in the differently exposed input images are read. For each value the associated weight is looked up in the weight-function and the response in the response-function. Furthermore, each exposure is scaled by the logarithm of its exposure time in seconds. This process is described in the following equation:

$$\ln E_i = \frac{\sum_{j=1}^{P} w(Z_{ij})(g(Z_{ij}) - ln\Delta t_j)}{\sum_{j=1}^{P} w(Z_{ij})}$$

where:

P	$=$	Number of photographs
$g()$	$=$	response function
Z_{ij}	$=$	Value of incoming LDR values per colour channel (0 - 255)
E_i	$=$	Real world luminance of pixel i
t_j	$=$	exposure time of photograph j in seconds
$w()$	$=$	weight function

The value of one pixel is independent from its neighbouring pixels. Because of this fact we decided to implement this step in CUDA to be executed on a GPU , thus enabling us to utilize the parallel computing capabilities of the GPU .

4.1.2 Robertson

The approach by Robertson et al. enables recovering an arbitrary response function from a given set of differently exposed images. The method proposed by Debevec and Malik [7] is widely used and we wanted a different algorithm with an arbitrary response function to compare it to. The concept of an adaptive weight function promised to be an interesting addition to enhance the quality of the output.

4.1.2.1 Procedure

Like the solution proposed by Debevec and Malik the process proposed by Robertson et al. is split up into an off-line and on-line step. The off-line step is performed once in the lifetime of the camera used and the on-line step is performed for every frame of the video.

Off-Line step The goal of the off-line step is to recover the response function of the camera and calibrate the weight function to that response function. Every camera has different quantization effects and noise properties. This means that the most useful range for captured data differs from camera to camera. Therefore a dynamic weight function based on confidence in the correctness of a value given by the response-function is proposed.

The iterative offline step consists of the following sub-steps:

1. First, an HDR image is created using the weight and response function from the previous iteration. For the initial iteration the response-function is a linearly increasing function, and the weight-function is a Gaussian bell curve. A bell curve is used because it best accounts for image noise artifacts in the low and high regions of the input data according to Robertson et al..

2. In the next step a new response function is calculated using the previously created `HDR` image using the following formula:

$$\hat{I}_m = \frac{1}{Card(E_m)} \sum_{(i,j \in E_m)} t_i \hat{x}_j^{l-1}$$

where:

$\hat{I}_m$	$=$	Value of response function at position m.
E_m	$=$	`LDR` input value for one colour channel (0-255).
$Card(E_m)$	$=$	how often this value occurs in the image.
t_i	$=$	exposure time in seconds.
$\hat{x}_j^{l-1}$	$=$	`HDR` input value.

3. Afterwards, the weight function is updated. The value of a weight function at a given position is determined by the gradient of the response function at the same position. The authors argue that most of the noise and image quantization errors occur towards the upper and lower bounds of the range of the sensor. Typically the response curve is flattened in these regions. The output of the response function does not change a lot for different input values in these regions. This leads to a higher inaccuracy in flat regions of the response curve. Therefore the steepness of the response curve is a good indicator for the correctness of a value returned by the response function.

After completing the steps listed above a single iteration is complete. The process of iterating is complete when one of the two following cases is fulfilled:

1. A user defined number of iterations is complete.
2. A user defined convergence rate is reached. The rate is determined by the difference of the response function to the response function of the previous iteration expressed as a percentage.

This series of steps only needs to be performed once in the lifetime of the camera.

On-line step The following on-line step is repeated for each frame of the video. Here for each colour-channel of each pixel the following equation is applied:

$$\hat{x}_j = \frac{\sum_i w_{ij} t_i \hat{I}_{y_{ij}}}{\sum_i w_{ij} t_i^2}$$

where:

$\hat{x}_j$	$=$	HDR output value.
w_ij	$=$	weighting value according to weight function.
$\hat{I}_{y_{ij}}$	$=$	response according to response function for incoming LDR value.
t_i	$=$	exposure time in seconds.

Similar to the approach proposed by Debevec and Malik, multiple exposures are merged into a single HDR image. The advantage of this approach lies in noise reduction. When only using a single exposure the output is fully dependent on the value of the pixel at that exposure. If the value of the pixel happens to be falsified by any type of noise, the resulting HDR image is affected by that noise term too. This problem can be mitigated by taking an average of multiple input exposures. Including the exposure time in the numerator of the formula leads to values of pixels with longer exposures to have a bigger influence on the result. This is done to utilize quantization effects described by Madden in [19]. Madden states, that higher exposures are more likely to contain useful data. This is because at lower exposures noise effects tend to contribute more to the captured values.

4.1.3 Tocci

The algorithms mentioned above only work on a single pixel and merge all available exposures. In contrast to this the approach proposed by Tocci et al. also takes local neighbourhood into account. Furthermore, not all exposures are used for each pixel but only the ones containing the most information. Noise reduction is further improved by taking local neighbourhood into account. This algorithm has a very complex on-line step compared to the other algorithms. The complexity of the step made it a challenge to efficiently port the process to CUDA.

4.1.3.1 Procedure

The approach proposed by Tocci et al. is only concerned with the on-line aspect of the HDR image creation process. It is assumed that a response function has already been recovered to be used by the algorithm. The authors suggest using the method proposed by Debevec and Malik to determine the response function. The basic idea is the same as in the approach proposed by Yamada et al. [34]. If possible, only the values of the highest exposure are used. But the idea is extended by sophisticated merging that is performed when the used image is close to saturation.

For each pixel position the value of the longest exposure and its neighbourhood are examined. This leads to four possible cases for each pixel. Each case requires a different operation to be performed:

- **Case 1: no pixels are saturated:** use the value of the highest exposure.
- **Case 2: main pixel is not saturated but local neighbourhood has one or more saturated pixels:** Merge the high exposure pixel with the next shorter exposure using a factor based on the number of saturated neighbours.
- **Case 3: main pixel is saturated but local neighbourhood has no saturated pixels:** Blend values of neighbours to be then merged with the main pixel.
- **Case 4: main pixel is saturated and local neighbourhood contains saturated pixels:** Use value of the next lower exposure.

If there are more than two exposures available, this process is repeated for the next pair of exposures.

4.1.3.2 Comparison of Radiance mappers

A brief overview of features of the radiance mappers mentioned in this chapter.

Algorithms	Arbitrary Response	Adaptive Weight	Neighbourhood	Prefer Higher Exposures
Debevec and Malik	●			
Robertson et al.	●	●		●
Tocci et al.	●[a]	-[b]	●	●

[a] depends on the algorithm chosen for recovering the response function
[b] has no explicit weight function

Table 4.1: Overview of radiance mapper features

4.2 Tone mapper

4.2.1 Ward

This algorithm consists of simple operations and is fast to execute. The results resemble results obtained by simply normalizing the HDR input. This can lead to undesirable results if the dynamic range is very high as described in Section 2.3. Contrast differences of the HDR input are preserved. This gives a good possibility to view raw HDR output without falsifying contrast. Knowing that the results will probably not be satisfying we still decided to implement the algorithm for the following reasons:

1. Quick to implement. Used to quickly test our radiance mapper implementations by providing a possibility to visualize the output.
2. Since no colour correction or contrast distortion is performed, we used it as a debugging tool.
3. Serves as baseline for comparisons with other, more elaborate algorithms.

4.2.1.1 Procedure

First the average brightness of the incoming HDR image needs to determined. Ward propose using the log-average brightness. It is defined as following:

$$\bar{L}_w = \frac{1}{N} \exp\left(\sum_{x,y} \log(\delta + L_w(x,y))\right)$$

where:

$\bar{L}_w$	$=$	Average Brightness.
N	$=$	Number of samples/pixels
δ	$=$	small value to avoid singularity if black pixels are encountered
$L_w()$	$=$	“world” luminance of incoming HDR -pixel

The logarithm is used here because it best represents how the human eye perceives brightness.

After the world luminance has been determined, a global scaling factor is calculated. This factor is then applied to every incoming HDR -pixel. The scaling factor is calculated using the following equation:

$$sf = \frac{1}{L_{dmax}} \left[\frac{1.219 + (L_{dmax}/2)^{0.4}}{1.219 + L_{wa}^{0.4}}\right]$$

where:

sf	$=$	Output display brightness
L_{dmax}	$=$	Maximum brightness of the displaying device
L_{wa}	$=$	Log-average brightness of incoming HDR image

The numbers in this formula are magic numbers that were determined by the authors in an empirical manner. These numbers represent the relation between the minimum discernible difference and the input brightness.

After this scale factor has been applied, some pixels might still lie outside of the range that the output device is capable to display. Therefore clipping is performed on the output values. This can lead to bloom artifacts for very bright areas. Ward argues that this is a desirable effect since it resembles effects of human vision as stated by Krawczyk et al. [17].

4.2.2 Larson

We chose this algorithm because it is a global tone mapper that can be used for comparison with the one proposed by Ward. Among the more sophisticated global operators we found this one to have very "pleasing" results. This algorithm is widely used in HDR -processing software. It offers a good trade-off between quality and resource consumption.

4.2.2.1 Procedure

First a low resolution version of the incoming HDR -image is created. This images is called the foveal image. The name stems from the fovea in the human eye. This is the region of the retina where humans focus and perceive details. This area roughly corresponds to one degree in our total FOV (field of view). The resolution of the foveal image is determined by the FOV of the cameras used to capture the high resolution image. Each pixel in the foveal image is covering one degree in the FOV of the camera. For example, if the camera has a horizontal FOV of 60 degrees, then the width of the foveal image is 60 pixels.

In the next step a histogram of that foveal image is created. Once the histogram is acquired, iterative histogram adjustment is performed. Here for each bin of the histogram a ceiling is calculated. If the number of elements is above the ceiling, the count of elements in the current bin is reduced to the ceiling determined for this bin. Larson et al. propose the following equation for calculating the ceiling for each bin:

$$\frac{\Delta L_t(L_d)}{\Delta L_t(L_w)} \cdot \frac{T \Delta b L_w}{[\log(L_{dmax}) - \log(L_{dmin})] L_d}$$

where:

L_t	=	Just noticeable difference in brightness
L_d	=	Display brightness (in candelas/meter^2)
L_w	=	World luminance (in candelas/meter^2)
T	=	Total number of samples in the histogram
L_{dmax}	=	Maximum brightness of the displaying device
L_{dmin}	=	Minimum brightness of the displaying device

This needs to be performed in an iterative fashion since part of the formula is the number of elements in the histogram. This count changes during each iteration if the number of elements in a bin are reduced. The process is repeated until convergence is achieved or a user determined number of iterations have been performed.

The next step consists of calculating the `CDF` (cumulative distribution function) to be used in the final step of the process. This last step is performed on the full resolution input image. Based on its brightness for each incoming pixel its corresponding histogram bin is determined. The value in the `CDF` of this bin is then used as a normalized output value. This normalized brightness needs to be mapped to the output device capabilities. In our case a normalized output value of zero corresponds to zero, and the normalized output value of one to 255.

4.2.3 Reinhard

In contrast to the other algorithms mentioned above this is a local tone mapper. We wanted to include at least one local tone mapper to perform a more inclusive comparison. Among local tone mappers the one proposed by Robertson et al. is widely used in `HDR` -processing software. It promised pleasing results because it is emulating a technique that has been used in analogue photography for many years with great success [2].

4.2.3.1 Procedure

The first step of this algorithm is to determine the average brightness of the incoming image. The same log-averages as mentioned in Section 4.2.1.1 are used here. Using the previously determined average brightness the incoming image is now scaled around a key-brightness value. The key brightness determines which value gets mapped to middle gray in the output (See Section 2.2.1). All other values of the image are scaled accordingly so that the key value is in the middle of the range.

To transfer the technique of “dodging & burning” to digital images, the authors propose taking an average of the local neighbourhood. Instead of performing this operation for entire areas of the image it is performed at a pixel granularity. This average is calculated using Gaussian kernels of different sizes. If a very bright pixel is surrounded by dark pixels, its brightness gets scaled down more than when surrounded by bright pixels. The opposite applies for dark pixels in a bright neighbourhood. The quality of the output depends on choosing the right size for the Gaussian kernel, which is used to perform the operation. This size is based on local contrast characteristics. The goal is to find the biggest gauss kernel that lies in a region of uniform contrast. To accomplish this a set of different sized Gauss kernels is generated. The size of the smallest kernel is one pixel, all following kernels are 1.6 times bigger than the previous one. Each pixel is convolved with this set of kernels. In the next step the resulting averages of these convolutions are compared using a centre-surround function. This comparison process starts with setting the smallest kernel as the centre and the second smallest as the surround. In the next iteration the previous surround is set as centre and the next bigger scaled kernel as the new surround. If the centre-surround function returns a high value, the averages of the two kernels have a large difference. This means that there is a change in contrast which is only present in the larger of the two kernels. Parts of the bigger kernel lie in a region of the image where brightness changes drastically. Therefore the smaller of the two kernels is the ideal kernel because it is the biggest within a uniform region of contrast, and the algorithm uses the result of this kernel to perform the final compression. In this final compression each pixel is compressed using the following formula:

$$L_d(x,y) = \frac{L(x,y)}{1+V_1(x,y,s_m(x,y))}$$

where:

$$\begin{aligned} L_d &= \text{output luminance.} \\ L &= \text{input luminance.} \\ V_1 &= \text{average of gauss kernel.} \\ s_m &= \text{size of gauss kernel.} \end{aligned}$$

This shows, that the incoming `HDR` luminance is scaled by the average given by the Gauss kernel with the ideal size. There are multiple ways how to perform the convolution of the pixel with the gauss kernels. The authors propose to transform both the image and the Gauss kernels into their frequency domain using a `FFT` (fast Fourier transform). The convolution is then performed within the frequency domain. The result is then transformed back using a reverse `FFT` .

4.2.3.2 Comparison of Tone mappers

A brief overview of the features and metrics of the previously mentioned tone mappers

Algorithms	Local/Global	Noise reduction	Contrast preserving	requires clamping
Ward	global		●	●
Larson et al.	global			
Robertson et al.	local	●		●

Table 4.2: Overview of tone mapper features

4.2.4 Weight Functions

The tone mappers proposed by Debevec and Malik, and Robertson et al. merge multiple exposures using weight-functions. These functions are used to give input with more “useful” information more weight compared to over- or under saturated pixels. We implemented different weight functions that can all be used in conjunction with those radiance mapping algorithms. In the following section we will briefly discuss these weight-functions.

4.2.4.1 Debevec

The function proposed by Debevec and Malik is a hat function that linearly increases towards the middle of the range and then linearly decreases towards the upper end of the range. The authors argue that pixels captured close to the upper and lower bounds of the range of the sensors contain less useful information for the following reasons:

1. The sensor is more prone to noise artifacts towards the limits of its range.
2. Pixels that are black or white are likely to be results of the image being over- or under exposed.
3. Colour intensity is subtly boosted by giving pixels in the middle range more weight. According to Petit and Mantiuk [25] this leads to an increased perceived pleasantness of the resulting image.

We had to slightly adapt the formula proposed by Debevec and Malik. If a pixel is perfectly white in all available exposures, it would be given a weight of zero in each exposure. This would lead to white surfaces turning black in the HDR output. To avoid this we add an offset of one to the weights used. By applying this offset we can preserve the colour white in the HDR output.

4.2.4.2 Robertson

Here the authors propose taking into account the confidence in a value being correct. They argue that a value is more likely to be correct if the response function of the camera at that position is steeper. Different real world brightness values are more distinctly detectable this way. Furthermore, the authors argue that instead of using a linear function like Debevec and Malik, using a Gaussian bell curve best approximates noise characteristics of digital image sensors. For this reason the initial weight function used for calibration is a Gaussian bell curve.

4.2.4.3 Mitsunaga

As opposed to the previous authors, Mitsunaga and Nayar argue that the most useful values are towards the upper range, not the centre. Similar to [27], the weight function is based on the slope of the response function.

4.2.4.4 Linear

We also implemented a linear weight function. Here for each incoming value the middle of the range is returned. In our case, this weight function always returns 127.5. We use this weight function as a baseline against which the other proposed functions are compared.

4.2.4.5 Comparison of Weight functions

A short overview of the functionality and features of the presented weight functions:

Algorithm	Linear	Confidence	Adaptive	Exposure Time
Debevec and Malik	●			
Robertson et al.		●	●	●
Mitsunaga and Nayar		●	●	
Linear	●			

Table 4.3: Overview of Weightfunctions

Chapter 5

Implementation

Contents

5.1 Implementation in C++ 33

5.2 Implementation in CUDA 34

5.1 Implementation in C++

All host code of the pipeline is implemented in C++. As mentioned in Section 2.4.3 the modules within the pipeline are loosely coupled. The individual components of the `HDR` module extend this idea. The user can select which radiance mapper, tone mapper and weight function are used at start-up. Each component shares a common interface to enable communication with the other components of the module.

We also decided to implement the off-line steps of the algorithms proposed by Debevec and Malik, and Robertson et al. in C++ for the following reasons

1. There is no real-time constraint and the step only needs to be performed once in the lifetime of the camera.

2. Implementing in C++ enables us to use existing libraries to solve complex mathematical operations.

3. It is easier to implement the processing as single threaded operation because many steps require iterating and using results from previous iterations. Performing such operations in parallel on a `GPU` introduces unnecessary synchronization complexity.

In the approach proposed by Debevec and Malik, a SVD is created and solved in order to recover the response function. For this purpose we employ the EIGEN-Library *. This is a template library that easily integrates into existing code bases. We use a highly optimized version of the algorithm proposed by Jacobi [16] that is offered by EIGEN-Library.

To obtain data which is used to fill the SVD we gather random sample points of the incoming images. For each chosen sampling pixel position we read the values from all exposures. Therefore we know the value of each pixel in each of its exposures. To avoid uneven sampling, which is inherent in random sampling, we divide the input images into zones. One random pixel is chosen within each zone. Users can specify how many sample points to take. Too few might cause under sampling, thus causing an imprecise recovery of the response function. On the other hand too many sample points cause a long execution time. Our experience shows that about 800 sample points are a good midpoint.

The algorithm put forward by Robertson et al. requires performing Gauss-Seidl[11] relaxation. Instead of opting to use the EIGEN-Libray to compute this step we wrote our own solver. It was less time consuming to write our own implementation than convert the data to a format compatible with the library. This implementation is likely to be less efficient than the optimized versions provided by EIGEN-Library, but it is sufficient for our purposes.

The algorithm furthermore requires a radiance-mapping step to be performed. We use our implementation of the on-line step that executes on the GPU for this purpose. This means that for each iteration we have to upload the input images and download the results. Although copying data to and from the GPU takes a long time, it is still quicker than executing the on-line step on the CPU . This is caused by the process working on high resolution input, as opposed to low resolution created by the sampling method used by Debevec and Malik. In the case of high resolution input we can fully leverage the parallel capabilities offered by the GPU .

5.2 Implementation in CUDA

In this section we discuss how we implemented the algorithms introduced in Section 4. As previously mentioned, we implement the operations that need to be performed for each frame of the video in Nvidia's CUDA to be executed on a GPU . This enables us to leverage the parallel computing capabilities of a GPU . It is necessary in order to achieve computation times below the real-time threshold for our high resolution input.

*http://eigen.tuxfamily.org

5.2.1 CUDA concepts

First we want to briefly describe some features offered by CUDA that we utilize across our different implementations.

CUDA Arrays CUDA-Arrays are an opaque memory layout where the content can only be accessed via special handles. The individual pixel values are accessed along a Z-curve. This way, 2D spatial relations can be simulated within a 1D-data structure. Because of this special layout, the content of CUDA-Arrays cannot be accessed directly. Nvidia offers either texture- or surface-objects for that purpose. Individual values of the arrays can be addressed using these objects.
Furthermore, CUDA provides specialized implementations of standard C routines `malloc, memcpy & free` to handle data within CUDA-Arrays.

Texture Objects [†] Texture objects are a read-only data structure that enables accessing CUDA-Arrays. They offer three addressing modes: 1D, 2D, and 3D. This is important since some features take locality into account, and their behaviour changes according to the dimension used. Texture memory offers the following features:

- **Caching:** Caching is performed to exploit spatial locality. For each data access in an image its local neighbourhood is cached. If a different thread accesses this local neighbourhood the value is read right from the cache and thus the necessity for an expensive global memory read is avoided. Developers have to pay attention to how individual threads access image data to utilise this kind of caching.
- **Interpolation:** When reading data, texture objects can perform linear interpolation between the value read and its neighbours. This operation is implemented on a hardware level and therefore does not affect reading time.
- **Bounding:** As opposed to normal linear memory access, out of bounds access is handled automatically. Instead of causing a segmentation fault, texture objects offer the following options when out of bounds access occurs:
  - **Wrap:** The values of the image are repeated along all borders.
  - **Clamp:** The value at the border is extended into the direction of the edge. Therefore reading out of bounds equals to reading along the border.

[†] *Note:* Texture and Surface Objects are only available for CUDA 5.0 and Compute Capability 3.0 and above.

  - **Mirror:** Similar to wrap, but instead of simply repeating the values, the image is mirrored along the border.
  - **Border:** When reading out of bounds, a predefined value is read.
- **Normalization:** All data can be read using normalized coordinates. This enables algorithms to work across different images without assumptions about the size of the input.
- **Type conversion:** When reading from a texture, the value can be converted from `int` to `float` without affecting performance.

Surface Objects Similar to texture objects, surface objects offer a way of accessing CUDA arrays. The main difference to texture objects is that surface objects also offer write access. In order to enable write access, the special texture fetch hardware cannot be used. Regular global memory reads are performed instead. Therefore spatial caching is not available. Traditional caching of global memory is used. It the data is only being read, texture objects are more useful and offer better caching.

5.2.2 Common

The following approaches are shared by all the implemented modules. Since the `HDR` module was a part of a bigger pipeline, clear interfaces for other modules were defined. The following pipeline details mostly deal with inter-module communication and the way incoming and outgoing data is processed.

- All image data passed from and to the `HDR` module is stored in CUDA arrays. This is for the benefit of the Stitcher Module, following our `HDR` module (See Figure 2.5). The stitcher performs many reads in the local neighbourhood of a pixel. Therefore the texture objects were a better choice in order to utilize the spatial caching they offer.

- The output of the radiance mapper consists of a float value per pixel. This means that instead of eight bits, there are 32 bits of information per pixel. Transferring data between the radiance mapper and tone mapper threatened to become a bottleneck. Therefore we decided to store `HDR` data using only 32 bits of memory encoded as “Real Pixels” proposed by Ward [31]. All three colour channels share a common 8-bit exponent, and for each pixel an 8-bit mantissa is stored. If the values of the pixels are far apart, some precision is lost, but in our case this did not become an issue.

- The `HDR` module offers reading and writing all data in different colour-spaces. Some of the implemented algorithms work in RGB colour-space whereas others work in YUV. In order to simplify inter-module communication, all read and write operations perform colour-space conversion when necessary. The user can specify the colour-space of the input and output of the `HDR` module.

5.2.3 Debevec

The on-line step of the algorithm is executed for every frame of the video. Computationally this step is simple. For each pixel position the values at different exposures are read. This is performed by a texture read. We made sure that adjacent threads read adjacent pixels. This enables us to fully utilize spatial caching provided by CUDA texture objects. Furthermore, lookups in pre-calculated response and weight functions are performed. These functions are stored as two arrays in global memory. Since they contain only few often read values (usually 255), we can utilize caching performed to global memory reads. Computationally the most expensive operations are logarithms and exponential functions required by the equation introduced in Section 4.1.1.1.
The result of the operation is written to a surface object. The value being written is encoded as a “real pixel” as mentioned in Section 5.2.2.

5.2.4 Robertson

The implementation of the on-line step of this algorithm is very similar to our implementation of the algorithm suggested by Debevec and Malik. Similarly texture reads, lookups in weight and response functions, and writes of “real pixels” are performed. The difference lies in the formula used to merge the exposures. As described in Section 2, the calculation requires power functions. These operations are slower compared to the logarithms used by Debevec and Malik. As can be seen in Figure 6.1, this causes slightly longer execution times when compared to [7].

5.2.5 Tocci

This algorithm consists of a single, complex step. Each pixel can have multiple states and complicated lookups have to be performed. These operations can cause a lot of thread divergence, which has a negative impact on execution time. These properties seem to favour an implementation executed on a `CPU` . The reason we decided to implement it in CUDA is the size of the input. When the pipeline is fully deployed, the input consists of 10 images with a resolution of 2040 × 1080 pixels each. This high resolution favours the parallel capabilities of a `GPU` . We decided to implement this algorithm to be executed on the `GPU` since the incoming data is residing in the memory of the `GPU` . Copying this data into host memory, processing it, and uploading it would have added too much of a performance penalty.

We will now go through the different steps of the algorithms and explain the optimizations and compromises we applied.

1. The first step is to determine the state of the pixel. As mentioned in Section 4.1.3.1 the state of a pixel is based on the saturation of the pixel itself and its local neighbourhood. We iterate over the local neighbourhood of each pixel, counting the number of saturated pixels. Here the local caching property exhibited by texture objects helps reduce read times.

2. In the next step, based on the state of the pixel itself and its neighbourhood, different functions that handle the states are called. This can cause a slowdown since different execution paths are taken for each pixel. This cannot be avoided, but we managed to limit the divergence. We work only on the luminance channel, not manipulating the chroma channels. Under these circumstances each pixel can only have four states. If we worked on all three colour-channels, each pixel could have 81 different states that would alter the execution paths. Depending on the state of the pixel one of the following operations is performed:

State 1: No operation is performed. The current value of the pixel is passed on to be used in the next iteration.

State 2: A scaling factor is calculated based on the number of saturated pixels in the local neighbourhood. The value of the pixel in a lower exposure is read. Then the value of the low exposure pixel is "converted" to a higher exposure. This is done via a reverse lookup in the response function. This lookup requires finding the nearest float value to the given one. To perform this lookup we employ quick search paired with nearest neighbour comparisons. Comparing each value with adjacent values creates issues with thread divergence. This conversion operation turned out to be very computationally expensive as can be seen in Section 6.1.3.4. Eventually the converted low exposure value is merged with the current high exposure value using the previously calculated scaling factor.

State 3: First the entire neighbourhood in a lower exposure is read. Then for each pixel in the low exposure neighbourhood the difference to the centre pixels is calculated and stored. Next the higher exposed neighbourhood is scaled using the previously calculated difference. Then the high exposure neighbourhood is further scaled using the same factor as in state two. Eventually this scaled neighbourhood is merged with the high exposure centre pixel.

State 4: The value of the pixel in a lower exposure is read and used for the next iteration.

3. When all available exposures are merged as explained in step 2 the final step is performed. In this step we look up the corresponding output value in the response function for each merged pixel. The result is encoded in the “real pixel” format (see 5.2.2) and written to a surface object to be passed on to the tone mapping module.

5.2.6 Ward

The biggest challenge in this implementation was to determine the log-average brightness of the incoming image in a timely fashion. We decided to implement this step using the paradigm of parallel reduction. We use a highly optimized version of this approach proposed by Harris et al. in [15]. In this approach each thread performs multiple reductions instead of one. This enables us to use the number of threads that is optimal for the GPU being used. As can be seen in Section 6.1.3.8, determining the incoming brightness takes up the majority of computation time of this algorithm. The log-average brightness is then used to calculate a scaling factor that is applied to every pixel. Since this factor is the same for every pixel, we perform this calculation in a single thread on the CPU . The value is passed to the GPU as a parameter during kernel launch. In the final step we apply this scaling factor to the incoming HDR values. This step is the same for each pixel and is performed in parallel. For each pixel we first convert the incoming real pixel value to RGB. Then we multiply the pixel value with the scaling factor. Finally we perform clamping on the resulting values to fit into the displayable range of the device. The result is written to a surface object for further processing.

5.2.7 Larson

Here for each frame we have to calculate a histogram and perform iterative equalisation on it. We took the following measures to improve performance:

1. First we reduce the problem size by creating a scaled down version of the input image. The size of this low resolution version is determined by the `FOV` of the camera. One pixel equals one degree in horizontal and vertical `FOV` . We launch one thread per pixel in the low resolution image. Each kernel uses a box filter to accumulate the pixels within the region it is responsible for. Again, we utilize texture objects and their spatial caching property to increase read throughput.

2. In the next step we calculate the histogram of the low resolution image. Here we employ a parallel approach. For each pixel a thread is launched. We calculate the corresponding bin in the histogram. When increasing the value of a bin we use an atomic add operation. This might cause some serialization of the parallel execution, thus slowing down the operation. We could not find a better way to ensure data coherence among the individual threads.

3. In order to calculate the histogram in the previous step, we needed to get the minimum and maximum value in the image. We implement parallel reduction to solve this problem. The input data consists of the low resolution image. This implies that the input size is small. We did not use the optimizations proposed by Harris et al., but instead wrote our own, simpler implementation. We did this because the problem size is very small. We wanted to avoid the overhead associated with determining the optimal number of threads.

4. After calculating the histogram, we perform iterative histogram equalization. Here for each bin the contrast to its neighbouring bins is calculated. If the difference exceeds a certain contrast ceiling, the counts in that bin are reduced to the ceiling. The ceiling is based on the total count of elements in the histogram. This causes histogram equalization to be an iterative process.

   This step is performed by a single thread on the `GPU` . Since this equalisation is performed in a strictly serial way, the power of a `GPU` is not utilized. Our measurements showed that this step takes approximately 100 micro seconds. We deem this performance good enough for our purposes.

5. In the final step we perform a lookup in the `CDF` of the histogram for each incoming pixel value. Apart from creating the low resolution input, this is the only step that operates on the full resolution input. Despite its simplicity, this step turned out to be among the most time consuming (See Figure 6.11). This is caused by the necessity to perform texture read and surface write for each pixel.

5.2.8 Reinhard

This algorithm is divided into two major steps. One performed during initialization of the pipeline and one on-line.

Initialization An important aspect of this algorithm is that the input is convolved with multiple gauss kernels of different sizes. The gauss kernels are stored as an image with the same resolution as the input. We perform a forward `FFT` on each gauss kernel during initialization and store the resulting complex kernels. In the on-line step these kernels are convolved with the input image in the frequency domain.

Online For each frame of the video we perform the following operations:

1. First, we scale the whole input around a key brightness value. This key brightness determines which value gets mapped to middle gray in the output. All other values of the image are scaled accordingly, so that the key value is in the middle of the range. See Section 2.2.1 for more information on the key-brightness value.

2. Next, we perform a forward `FFT` operation on the Input. For this purpose we use the CUFFT [‡] library provided by Nvidia. The functions provided by the library work on arbitrarily sized input. The best performance can be achieved if the input size is a power of two. We decided not to perform input padding to fulfil that requirement for two reasons:

   (a) We treat the input of multiple cameras at the same exposure as one huge image. This means that our input image of one exposure is 5400×2040 pixels. The next available power of two is $2^{13} = 8192$. An image with size $8192 \cdot 8912$ is roughly 517 % larger than the original input. Although the algorithm performs quicker with an input that fulfils the power of two requirement, the increased input size negates the gain in speed.

   (b) As we will discuss in more detail in Section 6.1, this algorithm already requires a lot of memory. Increasing that memory footprint by 517 % has negative effects. Fewer Gauss kernels would fit into the memory of the `GPU` , thus leading to less noise reduction and likely a less pleasant image.

[‡]`https://developer.nvidia.com/cuFFT`

3. Each complex, transformed pixel is multiplied by its corresponding pixel position in the transformed Gauss kernel. For this purpose we use the complex data-type offered by CUFFT.

4. A reverse `FFT` transform is applied to the resulting image. The result still consists of a real and a complex part. The complex part should be zero, but due to computational inaccuracies it is only close to zero. Still, we discard the complex part and only use the real numbers because the very small complex components are of no interest to us.

5. In the final step we determine the ideal size of gauss kernel for a given input pixel. For this purpose we compare two adjacent sized kernels. This way we can determine contrast edges and pick the smaller kernel that is still within a region of uniform contrast. The result of this kernel is then used to calculate the equation presented in Section 4.2.3.1. The result obtained using the equation is then written to a surface object for further processing. This sequence of steps is performed in parallel for each pixel on the `GPU` .

We let the user decide how many gauss kernels to use. The original authors propose to use eight kernels. In our implementation each gauss kernel is stored in an image with the same size as the input image. Therefore each kernel requires $\sim$ 126 MB of device memory. With our testing set-up(for details see Section 6.1.1) we could not simultaneously achieve more than four kernels before running out of memory.
Only using one kernel equals reading the raw input and not performing any smoothing. Contrast compression is still performed, but no noise reduction or local contrast enhancement are applied to the input.

Chapter 6

Performance Measurement and Study

Contents

6.1 Performance **43**

6.2 Visual Comparison **70**

6.3 Results of subjective study **71**

6.4 Performance vs. Visual Result trade-off **75**

6.5 Final Result **76**

In this chapter we will perform an evaluation of the algorithms that we implemented. This evaluation will cover two areas. First, we compare the execution times of the different implementations. All algorithms had to perform the necessary operations below the real-time threshold of 40ms. Second, we conducted a controlled experiment evaluation the visual quality. For this purpose we had a small group of participants rate videos obtained by executing the different combinations of algorithms.

6.1 Performance

In this section we compare objective metrics of our implementation. We pay special attention to execution times and the achieved parallelism of our implementations on the `GPU` .

6.1.1 Set-Up

The machine we used for execution and evaluation has the following system specifications:

CPU: Intel Core i7-3930K, 3.2 gHz, 6 Cores, 12 MB Cache
RAM: 32GB, DDR3
GPU: Nvidia GTX680, 3GB Memory, Kepler architecture

We capture the panorama video using five cameras, each with a resolution 2040×1080. These cameras are mounted on a custom made rack. This rack enables the cameras to cover the entire stadium and guarantees a slight overlap in the FOV of each camera. This overlap is needed by the stitcher module to generate one panoramic image. Since the large difference in the amount of light available in the two different regions (sun-light and shadow) is our main challenge, we capture two different exposures where one captures the details in shadow region and the other in highlights. These two exposures are captured in turns by the same camera. Hence, to generate one high resolution panorama frame, we have 10 images as the input to the HDR module.

For measuring the performance we use two configurations. In one configuration we run only the HDR module. This provides a good overview of the raw performance of the implementations. In the second configuration we execute the entire pipeline as described in Section 2.4.3. This configuration provides an overview of the real-world performance of the implementation. All modules are executed on a single GPU . Each of the modules contains several kernels. The scheduling of these kernels on the GPU is managed by CUDA. Therefore, it must be noted that the execution times include the scheduling overheads. It can be seen in Figure 6.1 that the execution times of other modules fluctuate when different HDR algorithms are employed. This is due to the fact that CUDA changes the scheduling of the kernels according to various factors. The GPU used in our experiments is capable of executing up to 16 kernels concurrently.

All measurements were captured by taking the average of 3000 input frames. All operations were performed on “cold” systems. This means that no operations were performed before. This way there is no data in memory that could alter execution speed and loading times.

6.1.2 Execution Times

In this section we will present the measured results acquired as described in Section 6.1.1.

As can be seen in Figure 6.1 and Table 6.1, most combinations achieve an execution time below the required real-time threshold. The only exception is the approach proposed by Reinhard et al.. In the following sections we will provide a more fine grained analysis of each individual algorithm.

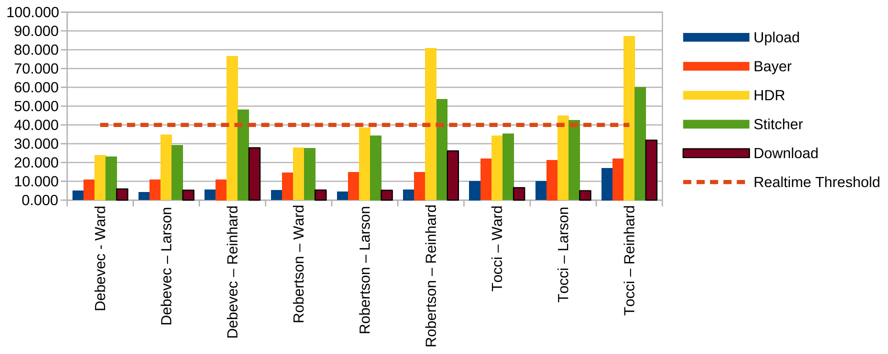


Figure 6.1: Execution times of the entire pipeline with different radiance mapper and tone mapper configurations

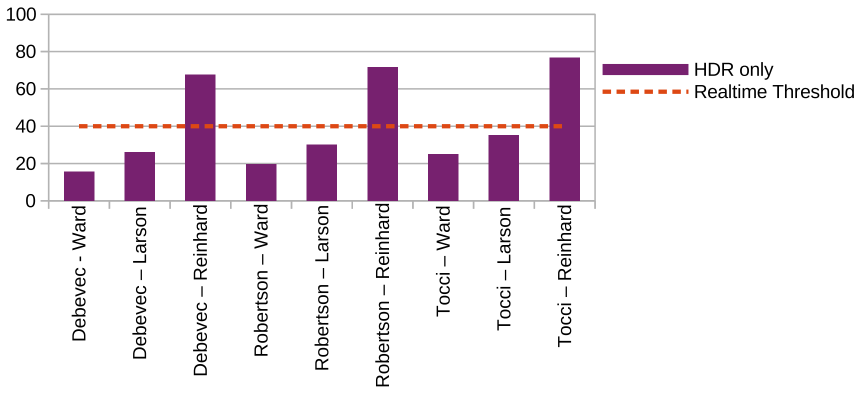


Figure 6.2: Execution times of the HDR module with different radiance mapper and tone mapper configurations

6.1.3 Detailed Evaluation

In this section we perform a detailed evaluation of individual kernels that are executed in our implementations of the algorithms described above.

Algorithm	Upload	Bayer	HDR	Stitcher	Download	HDR only
Debevec - Ward	5.038	10.853	23.814	23.173	5.904	15.564
Debevec - Larson	4.210	10.742	34.729	29.232	5.230	25.998
Debevec - Reinhard	5.520	10.944	76.629	48.210	27.829	67.774
Robertson - Ward	5.247	14.632	27.837	27.529	5.321	19.680
Robertson - Larson	4.385	14.726	38.638	34.340	5.196	30.095
Robertson - Reinhard	5.590	14.916	80.727	53.682	26.164	71.740
Tocci - Ward	10.096	21.992	34.351	35.341	6.576	24.948
Tocci - Larson	10.043	21.346	44.814	42.480	4.964	35.361
Tocci - Reinhard	17.116	22.087	87.291	60.132	31.861	76.697

Table 6.1: Execution times of all modules in different configurations of HDR algorithms

6.1.3.1 Legend

In the following sections we will present graphs obtained by using Nvidia's Visual Profiler. We will explain the legend and notations used by the visual profiler.

The following notations are used in these graphs comparing the utilisation of different modules of the GPU :

- **Load/Store:** Load and store instructions for shared and constant memory.
- **Texture:** Load and store instructions for local, global and texture memory.
- **Single:** Single-precision integer and floating-point arithmetic instructions.
- **Double:** Double-precision floating point arithmetic instructions.
- **Special:** Special arithmetic instructions such as sin, cos, pow, etc.
- **Control-Flow:** Direct and indirect branches, jumps, and calls.

In graphs showing instruction execution counts, the following notation is used:

- **FP32:** 32-Bit floating point instructions
- **FP64:** 64-Bit floating point instructions
- **Integer:** 32-Bit integer instructions
- **Control-Flow:** control flow instructions like if, else, for, etc.
- **Load/Store:** instructions related to loading and storing data
- **Bit-Convert:** Type-casing instructions

- **Comm.:** Communication
- **Misc.:** other operations that did not fit any of the previous categories
- **Inactive:** Thread executions that did not execute any instruction due to divergence

Pie charts that deal with stall reasons use the notation below:

- **Instruction - Fetch:** The next assembly instruction had not yet been fetched
- **Compute:** The compute resource(s) required by the instruction is not yet available
- **Constant:** A constant load is blocked due to a miss in the constant cache
- **Data Request:** A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding.
- **Synchronization:** The warp is blocked at a syncthreads() call.
- **Execution Dependency:** An input required by the instruction is not yet available.
- **Texture:** The texture sub-system is fully utilized or has too many outstanding requests.

6.1.3.2 Debevec

Performing the initial recovery of the response function only needs to be performed once in the lifetime of the camera. Here we did not put emphasis on computation speed but rather on the accuracy of the result. The main influence on the execution time and accuracy is the number of sample points taken. Execution time increases exponentially with the number of points taken. Our experience shows that about 800 sample points are a good middle ground between accuracy and calculation time. The execution time plotted for varying number of sample points can be seen in Figure 6.3.

An analysis of the on-line step can be seen in Figure 6.4. Despite the proposed formula being rather simple, we can see that this operation is bound by computation time, not memory operations(Figure 6.4a). Figure 6.4b shows the utilisation of the different modules of the `GPU` . One can see that the modules are not evenly used. Most operations are performed by the single-precision arithmetic unit and the module for "special" mathematical operations. This uneven load can be attributed to the formula of the on-line step introduced in Section 4.1.1.1. It mostly consists of additions, divisions and one logarithmic operation. Figure 6.4c shows how many operations that were performed by the cores

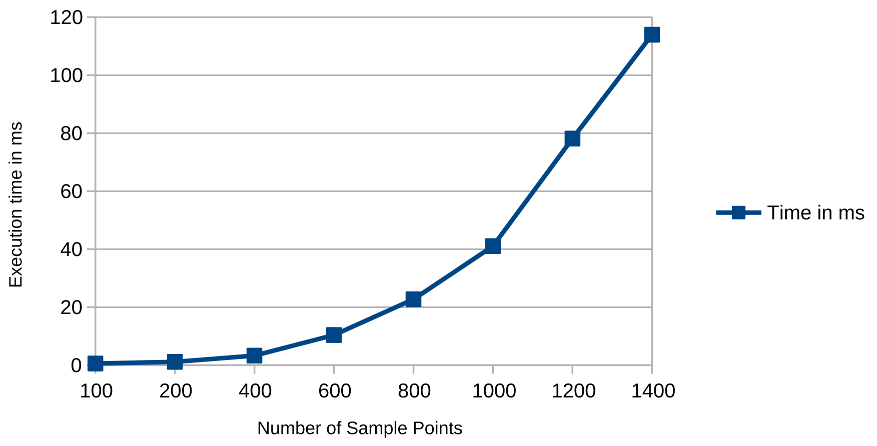


Figure 6.3: Execution times of the off-line calibration step of Debevec and Malik

were idle operations. For these idle operations the cores were not doing any work. This algorithm performs at 90% of theoretical peak efficiency.
Execution Time: 7.737ms

6.1.3.3 Robertson

The on-line step performed for each frame of the video is very similar to the one proposed by Debevec and Malik. This similarity can be observed when comparing Figure 6.4a and Figure 6.5a. Here one can see that the algorithm suggested by Robertson et al. requires only slightly more time for computations.

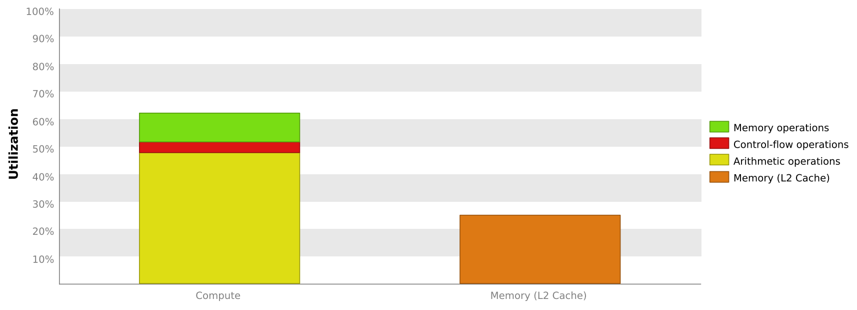
Figure 6.5b shows that the workload on the individual modules of the GPU is not evenly spread. The highest strain is put on the single precision arithmetic unit. On the other hand the other modules are not fully utilized.

Finally, by comparing idle times of the algorithm put forth by Robertson et al. with the algorithm proposed by Debevec and Malik (Figure 6.5c and Figure 6.4c, respectively), we conclude that our implementation of the algorithm proposed by Debevec and Malik is more efficient because it contains fewer idle operations.
Execution time: 11.2ms.

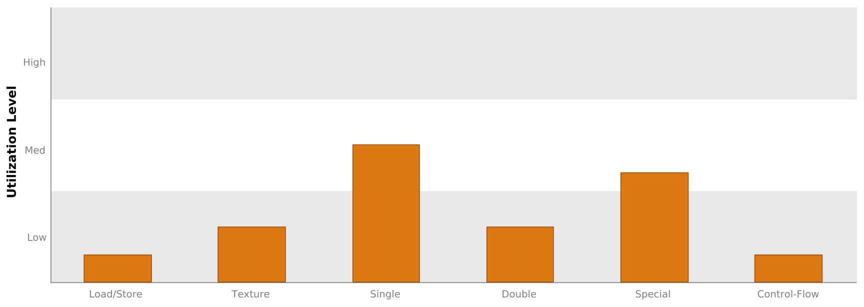
6.1.3.4 Tocci

This algorithm performs multiple operations that take the local neighbourhood into account. This characteristic implies that a lot of memory reads are required for each pixel.

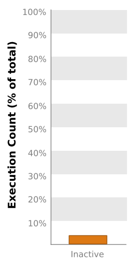
This fact becomes evident when looking at Figure 6.6a. The performance of this algorithm is limited by the available memory bandwidth. One cause of memory performance degradation is caused by register spilling. If the number of local variables exceeds the number of available registers of each core, the variables are moved to L2-cache. The visual profiler reveals that 81% of all memory traffic consists of read/write operations to L2-cache. The cause for this register spilling is the fact that the algorithm proposed by Tocci et al. consists of a single, complex step containing many numerical factors that need consideration. Another memory related problem we have encountered is reading the response function.



(a) comparison of time spent on computations versus memory I/O

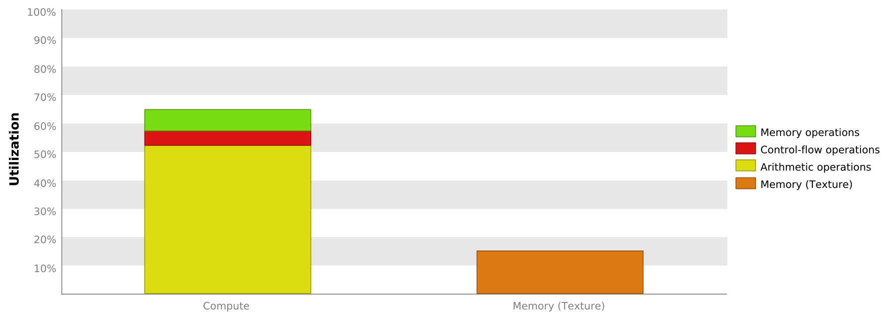


(b) comparison of the utilisation of the different modules on the GPU

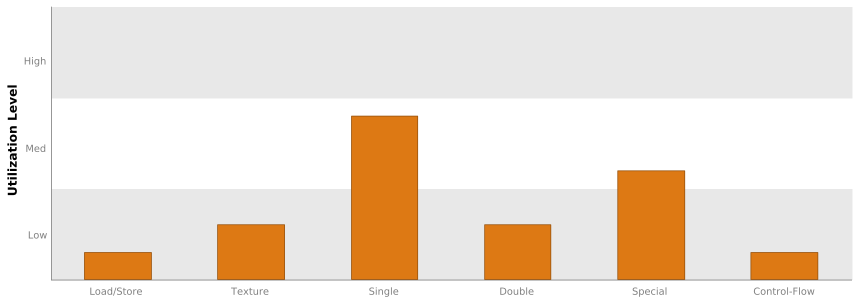


(c) Instruction execution count

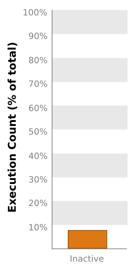
Figure 6.4: Evaluation of our implementation of Debevec and Malik's algorithm



(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the `GPU`



(c) Instruction execution count

Figure 6.5: Evaluation of our implementation of Robertson et al.'s algorithm

This response function is stored in global memory. When reading global memory, having coalesced memory access is desirable. Unfortunately we cannot guarantee this, since each read access of the response function is based on the value of the pixels read. These pixel values vary greatly. This leads to poor performance when trying to read the response function. Despite this fact, Figure 6.6d shows that the throughput to global memory is very high. We achieve this high throughput by coalescing the pixel access between and within threads. This observation is supported by Figure 6.6b. Most of the operations are performed by the texture unit. We use texture objects for accessing the input images, thus causing a high demand on the texture unit. The slow performance during reading of the response function is compensated for by good performance during reading the pixel values.

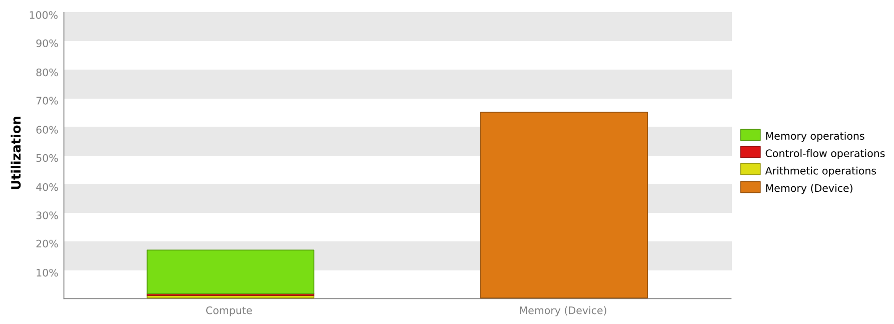
Nvidia's visual profiler provides an overview of the primary causes for stalls. The reasons in this implementation can be seen in Figure 6.7a. One can see that the primary reason is execution dependency. Execution dependency means that instructions cannot be executed because they depend on results of previous operations. As already mentioned, this algorithm consists of many complex operations where some depend on results of the previous operation.

Another problem we face in this algorithm is thread divergence. As mentioned in Section 4.1.3.1, each pixel can have one of four states. We deal with all those states within one kernel, which leads to thread divergence. Most of the pixels of our input fall into one category. This leads to a thread divergence of 7.4% in our evaluation. This is far from optimal, but also not as big a problem as we initially expected.

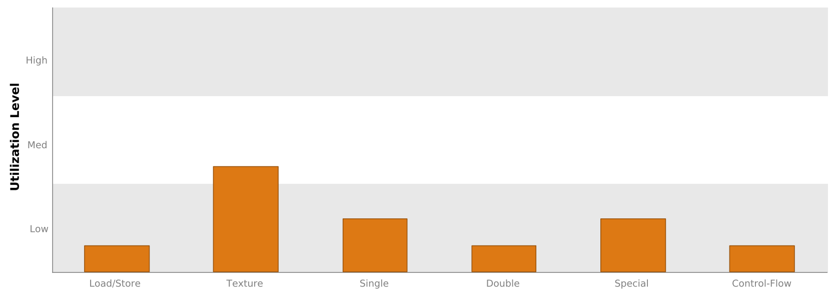
The number of instructions where cores were idle and did not contribute can be seen in Figure 6.6c. Cores were idle about 15% of the time in this algorithm. Compared to the idle times of Debevec and Malik, and Robertson et al. (Figure 6.4c & Figure 6.5c, respectively), the parallelism of our implementation of the algorithm put forth by Tocci et al. performs worse. Whether or not the longer execution time results in a more pleasing image will be discussed in Section 6.5.
Execution time: 30.234 ms.

6.1.3.5 Ward

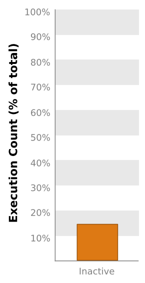
The kernel of this algorithm is very simple. Basically an `HDR` value is read, multiplied with a constant factor, and then written to the output image. What was surprising to us was the fact that this kernel is not limited by memory bandwidth, but instead by arithmetic operations. This can be seen in Figure 6.8a. The reason for this are colour-space conversions. We implemented all algorithms to be agnostic to the underlying colour-space used. This is done by converting the pixel values to the colour-space required by the algorithm when reading an writing. Within the kernel we use the RGB colour-space, while the rest of the pipeline is usually using YUV colour-space. Even though the operations within the kernel are simple, we still have to perform two colour-space conversions for each pixel. These conversions consist of a series of vector multiplications. The observation that colour-space conversions are a major contributor to execution time is supported by Figure 6.8b. Here the highest demand is for the single precision arithmetic unit. This is the one used for colour-space conversions. The high demand for the “special” module seen in the same figure can be explained by the fact that we perform gamma-correction on the output value. This step requires power operations which are performed by the “special” module.



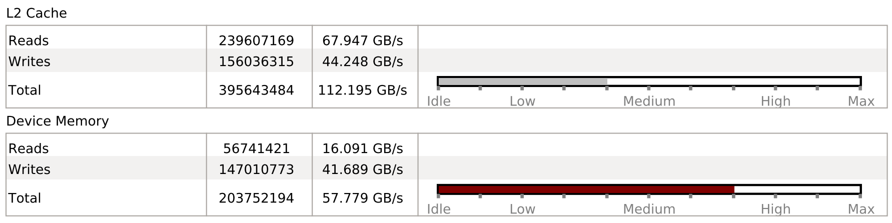
(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the `GPU`



(c) Instruction execution count

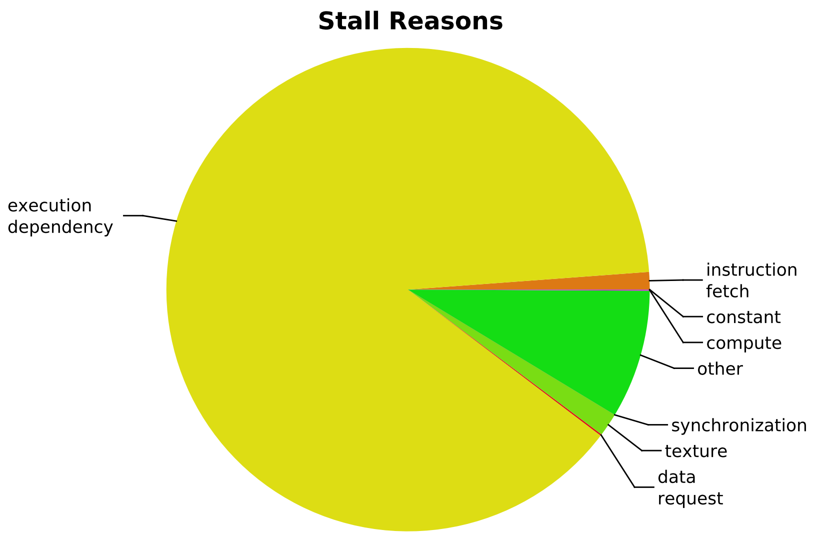

L2 Cache

Reads	239607169	67.947 GB/s	
Writes	156036315	44.248 GB/s	
Total	395643484	112.195 GB/s	Idle Low Medium High Max

Device Memory

Reads	56741421	16.091 GB/s	
Writes	147010773	41.689 GB/s	
Total	203752194	57.779 GB/s	Idle Low Medium High Max

(d) Memory bandwidth occupancy



(a) Computation stall reasons

Figure 6.7: Evaluation of our implementation of Tocci et al.'s algorithm

Figure 6.8c shows the number of idle operations which illustrate how well this algorithm performs in parallel. One can observe that less than 10% of operations were idle. We deem this performance good enough for our purposes.
Execution time: 7.758 ms.

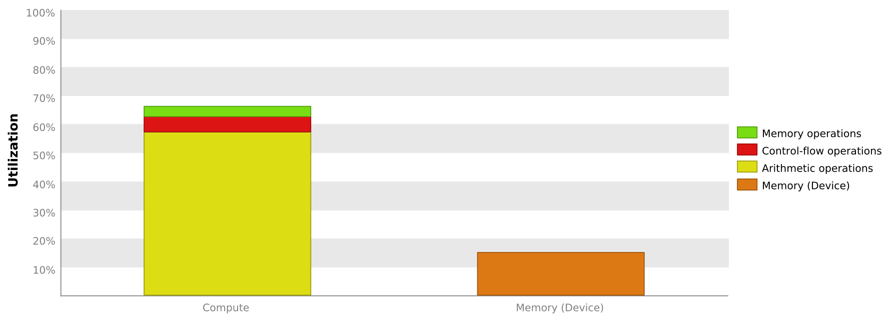
6.1.3.6 Larson

This algorithm consists of multiple kernels that are executed sequentially. We will briefly describe each kernel and perform a full performance analysis for each one of them.

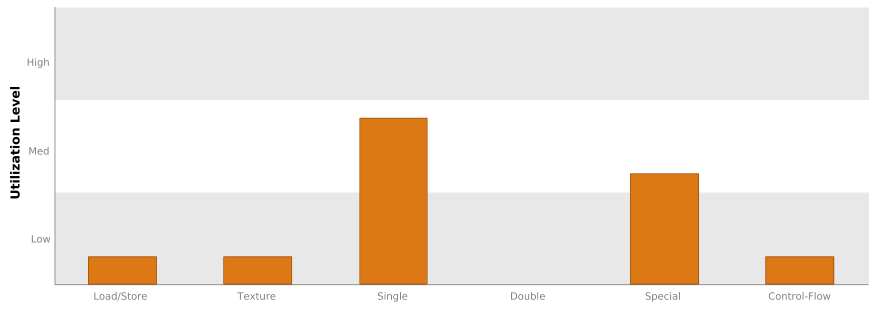
Create foveal image: The first kernel is responsible for creating the foveal image introduced in Section 4.2.2.1. We use a simple box filter to merge multiple pixels into a single pixel of the foveal image.

Figure 6.9a shows that the performance of this kernel is limited by memory bandwidth. This is due to the fact that each kernel has to read multiple pixels in order to eventually merge them.

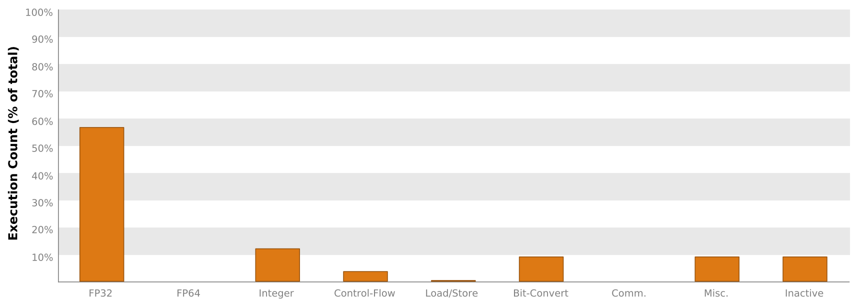
The utilisation of different modules seen in Figure 6.9b reveals that the "special" module is used the most by this kernel. This is caused by a conversion operation. We store HDR pixels as "real pixels" (explained in Section 5.2.2). We perform a conversion from the real-pixel-format to three floating point values to manipulate them. This conversion



(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the `GPU`



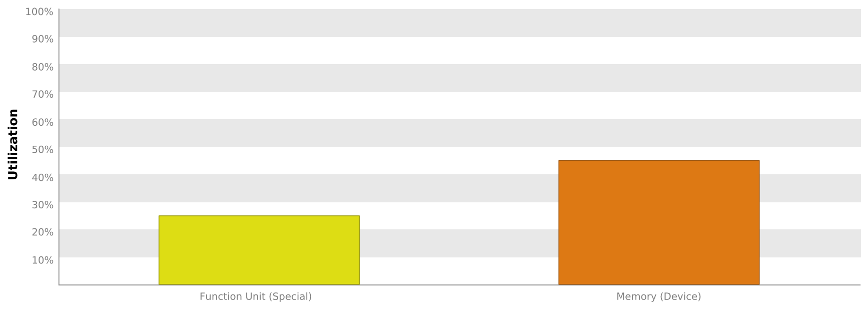
(c) Instruction execution count

Figure 6.8: Evaluation of our implementation of Ward's algorithm

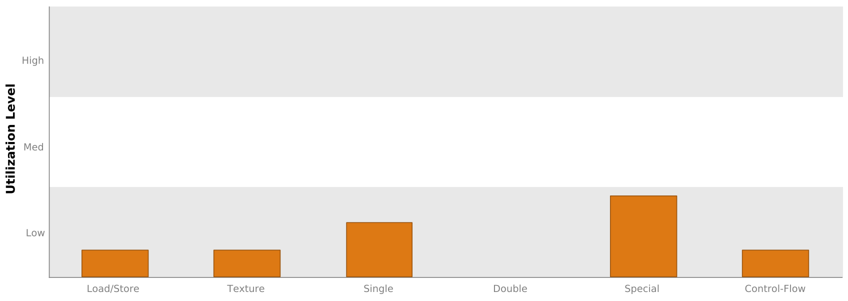
includes exponential operations which cause a high load on the "special" module.

When examining idle operations (Figure 6.9c), one can observe a high percentage of about 18%. The reasons for stalling, which causes the undesirable performance, can be found in Figure 6.9e. It is obvious that the primary cause is execution dependency. This is most likely due to stalls caused by waiting for operations responsible for reading pixel-values. These read operations can take a long time in which all following operations that depend on the read values can not be executed. Furthermore, Figure 6.9c shows that a lot of bit convert operations are performed. The reason for this can be found in image reading operations. These reading operations perform multiple type casting operations when converting from “real-pixels” to floating point values.

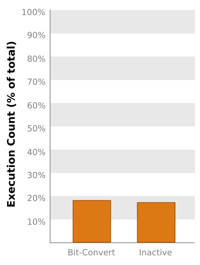
Figure 6.9d shows the memory throughput that we achieve with this kernel. This high occupancy is accomplished by coalescing the pixel access within each kernel. This way we can fully utilize the spatial caching provided by CUDA texture objects.
Execution time: 4.459 ms.



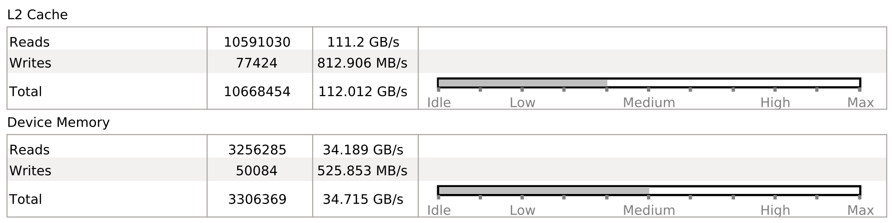
(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the GPU



(c) Instruction execution count



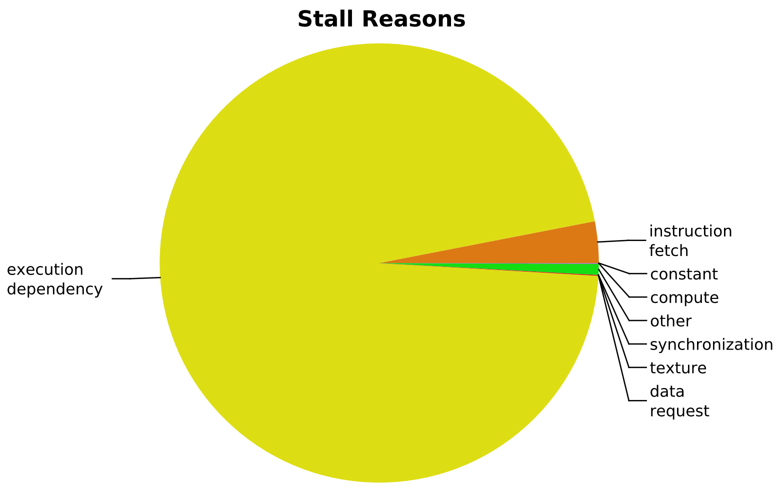
L2 Cache

Reads	10591030	111.2 GB/s	
Writes	77424	812.906 MB/s	
Total	10668454	112.012 GB/s	Idle Low Medium High Max

Device Memory

Reads	3256285	34.189 GB/s	
Writes	50084	525.853 MB/s	
Total	3306369	34.715 GB/s	Idle Low Medium High Max

(d) Memory bandwidth occupancy



(e) Computation stall reasons

Figure 6.9: Evaluation of our implementation of the sub step compute-foveal-image in Larson et al.'s algorithm

Histogram equalisation: The next kernel we observe will be referred to as calculate-histogram. This kernel performs histogram adjustment as introduced in Section 4. Since this operation depends on the entire image and is iterative, we implemented it to be executed by a single thread.

Our performance evaluation mostly focuses on the efficiency of parallel execution. This fact leads to poor results of this kernel in the performance domain. When comparing the utilization of the `GPU` (Figure 6.10a) with the same graphs of other algorithms this becomes immediately apparent. Both bars are very small, therefore indicating a poor utilization of the capabilities offered by the `GPU` .

Looking at the occupancy of different modules on the `GPU` (Figure 6.10b) shows that they are used evenly. There is no single bottleneck module that slows down overall performance.

Most of the instructions executed for a single warp are idle operations due to the non-existent parallelism in this implementation. This can be seen in Figure 6.10c. About 97%

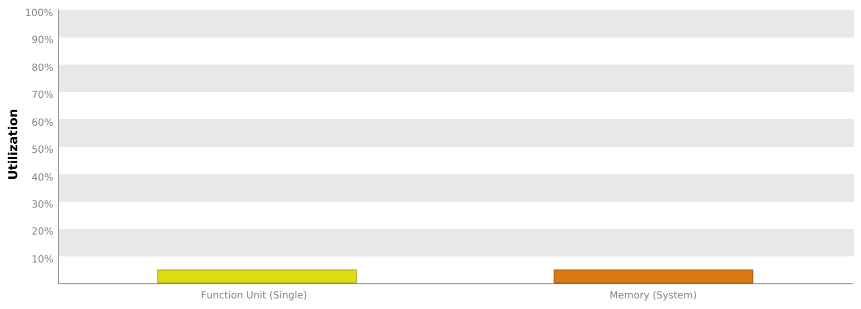
of the time warps were idle. There are 32 Threads in a warp. Of those threads only a single one was actually active. Each thread contributes 3.125% to the overall theoretical occupancy of 100% in a warp. This means that the single thread that was active actually was active most of the time and only performed few idle operations. All the other idle operations are caused by entirely idle threads.

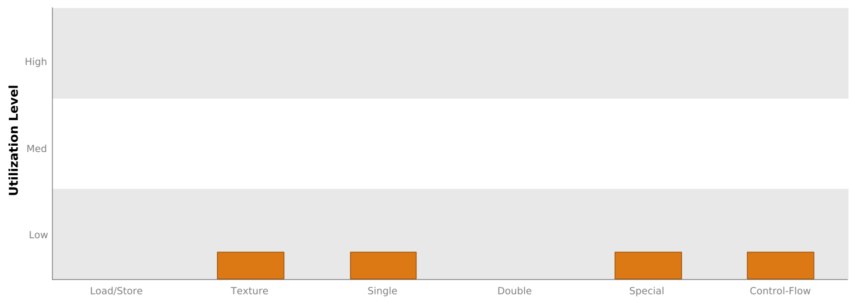
In Figure 6.10d we can see that we do not fully utilize the number of warps and threads. Again, this is due to the fact that we only use a single thread.

The previous observations for this kernel may lead to the conclusion that this step is implemented inefficiently, but our observations reveal that this kernel only takes 109 micro-seconds. This is less than 2% of the overall execution time of the algorithm. Therefore we conclude that this performance is "good enough" and does not warrant further optimisation because the gains in execution time are negligible.

Execution time: 0.109 ms.

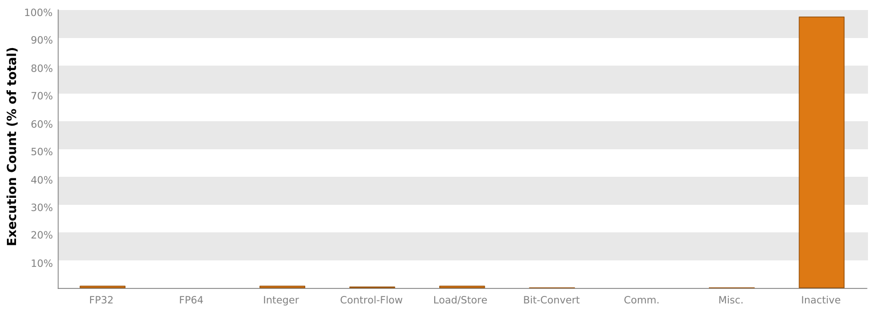
(a) comparison of time spent on computations versus memory I/O



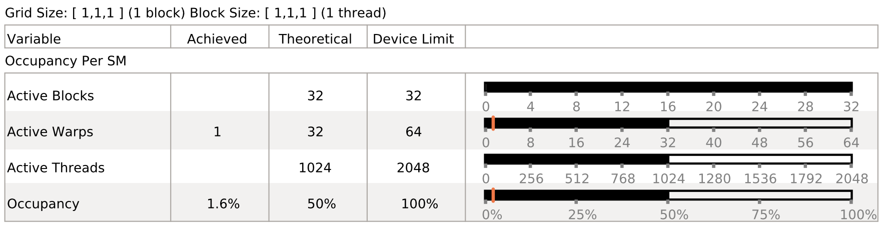


(b) comparison of the utilisation of the different modules on the GPU

Apply histogram: The last kernel of this algorithm that we will evaluate is called apply-histogram. Here the actual contrast compression is carried out. For each incoming HDR pixel-value we perform a lookup in the previously calculated CDF function and scale the incoming pixel accordingly.



(c) Instruction execution count


Grid Size: [1,1,1] (1 block) Block Size: [1,1,1] (1 thread)

Variable	Achieved	Theoretical	Device Limit	
Occupancy Per SM				
Active Blocks		32	32	0 4 8 12 16 20 24 28 32
Active Warps	1	32	64	0 8 16 24 32 40 48 56 64
Active Threads		1024	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	1.6%	50%	100%	0% 25% 50% 75% 100%

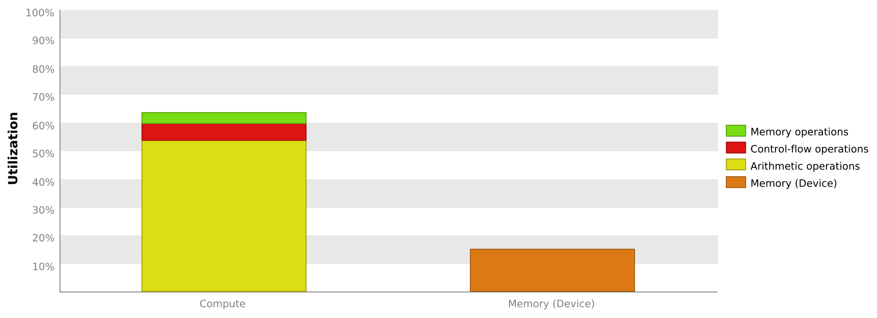
(d) Memory bandwidth occupancy

Figure 6.10: Evaluation of our implementation of the sub step calculate-histogram in Larson et al.'s algorithm

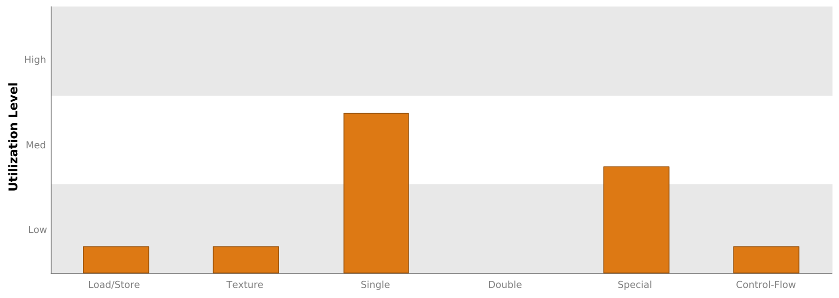
Figure 6.11a shows that our implementation is bound by the number of arithmetic operations that are performed. This is most likely due to the fact that we perform both a colour-space conversion as well as converting from “real pixels” (See Section 5.2.2) to three-floating point values. Furthermore, multiple computationally expensive multiplications and divisions are performed within the kernel. These operations seem to greatly outweigh the cost associated with reading `HDR` pixel values and writing `LDR` values.

When examining the utilization of the different modules of the `GPU` , as shown in Figure 6.11b, one can see that the load is not evenly distributed. There is a high demand placed on the single precision arithmetic unit and a medium load on the “special” arithmetic unit. Most of the other units remain mostly inactive. How this uneven distribution affects the amount of idle operations performed can be seen in Figure 6.11c. Less than 10% of instructions were idle. This is in a similar range to the other algorithms and we deem this performance good enough for our purposes.

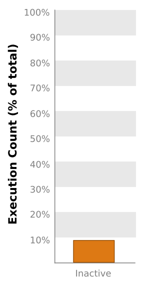
Execution time: 9.69 ms.



(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the `GPU`



(c) Instruction execution count

Figure 6.11: Evaluation of our implementation of the sub step apply-histogram in Larson et al.'s algorithm

6.1.3.7 Reinhard

As can be seen in Figure 6.1, this algorithm exceeds the real-time threshold in every possible combination with different radiance-mappers. We decided to do further testing by varying the number of Gauss-kernels used by the algorithm. This affects memory consumption and run-time. After convolving the image with the different kernels a reverse `FFT` transform is applied. The number of Gauss-kernels determines the size of the input for this operation. Fewer kernels result in quicker computation. This step is computationally the most demanding step that needs to be performed for each frame of the video. The execution time with different number of kernels can be seen in Figure 6.12. This figure shows the execution times with the whole pipeline active. In Figure 6.13 only the `HDR` module is used. Detailed execution times can be found in Table 6.2.

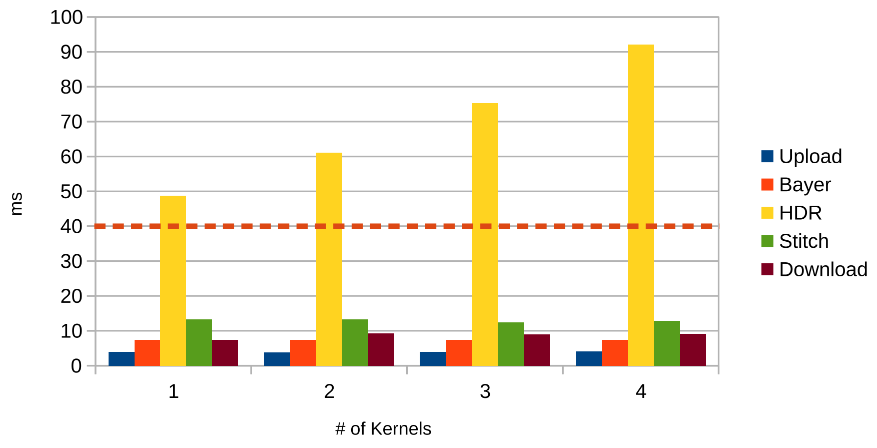


Figure 6.12: Execution times of the entire pipeline with different number of Gauss-kernels used in Reinhard et al.'s algorithm

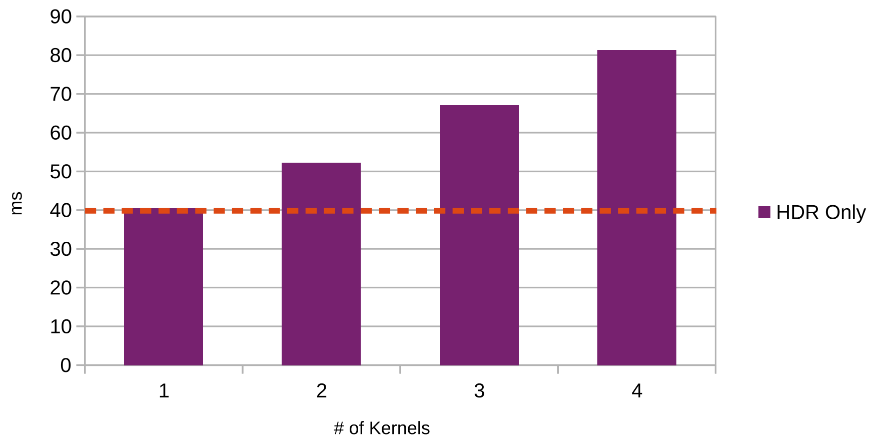


Figure 6.13: Execution times of the HDR module with different number of Gauss-kernels used in Reinhard et al.'s algorithm

Even with just one Gauss kernel the execution time exceeds the real-time threshold of 40ms. We think that there are two ways to achieve better performance:

1. **Reduce resolution of input:** Unfortunately this is not an option since one of the

goals of the pipeline is creating high resolution panoramic videos.

2. **Rework memory allocation:** We decided to favour image quality over speed. Therefore we perform multiple memory allocations and deallocations each frame to enable as many Gauss kernels as possible. Fewer kernels result in worse image quality because the neighbourhood observed is smaller. This leads to more image noise.

# of Kernels	Upload	Bayer	HDR	Stitcher	Download	HDR only
1	3.882	7.388	48.676	13.263	7.448	40.492
2	3.849	7.431	61.034	13.259	9.299	52.261
3	3.949	7.433	75.290	12.414	8.900	67.104
4	4.023	7.441	92.047	12.857	9.102	81.284

Table 6.2: Execution times of all modules with different number of Gauss Kernels used by the radiance-mapper proposed by Reinhard et al. in ms

In the following section we will perform a more detailed evaluation of the performance of multiple kernels that are part of this algorithm. We will not explain all of them, but the ones that have the biggest impact on performance.

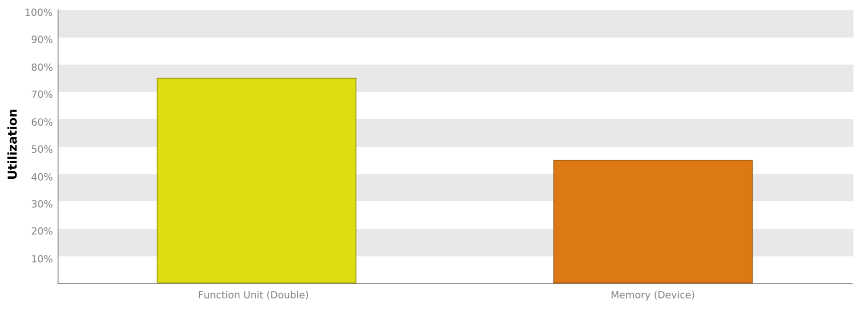
Create-radiance-image: The first kernel we discuss is called create-radiance-image. We perform two operations. First we convert the incoming `HDR` pixel values from “real pixels” (see Section 5.2.2) into floating point values. These values are then converted to a format used by Nvidia’s CUFFT which handles complex numbers. These conversions are necessary because in a later step we perform an `FFT` on this input that will result in complex values. The second operation performed by this kernel is scaling the brightness of the incoming pixels around a user determined key-brightness value. How this key brightness is affecting the image is explained in Section 2.2.1.

As can be seen in Figure 6.14a, this kernel is bound by the number of compute operations, not memory bandwidth. It is caused by the second step described above. For each pixel a number of arithmetic operations has to be performed in order to perform the scaling around the key-brightness value in a uniform manner.

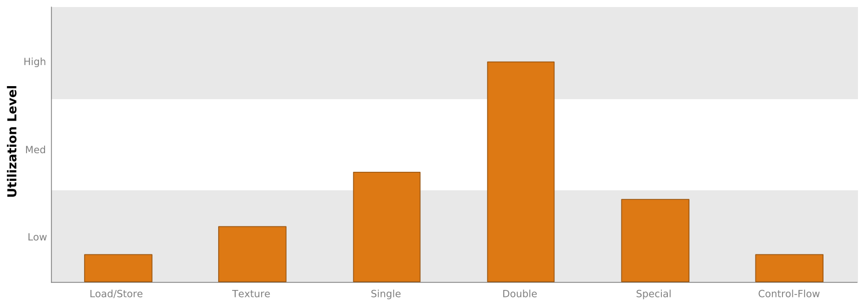
Figure 6.14b shows that the modules of the `GPU` are not used evenly. Most of the load is imposed on the module responsible for double precision operations. This is caused by the second step in the kernel. We perform the scaling of the pixels in double precision. This is done because the scaling often includes a form of compression. We use double precision instead of single precision floating point values in order not to lose the brightness difference between pixels that are very similar in brightness.

The efficiency of our implementation can be seen in Figure 6.14c. Only about 5% of the operations executed were idle instructions. Most of the time the `GPU` was performing as intended. This observation is further supported by Figure 6.14d. It shows that we achieved good memory throughput in this operation.

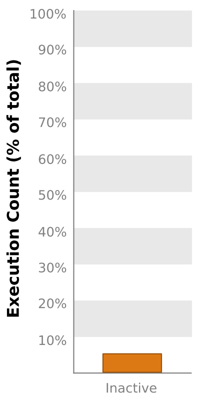
Execution time: 4.732 ms.



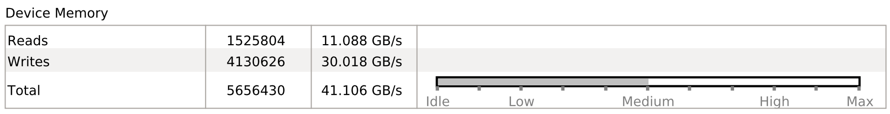
(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the `GPU`



(c) Instruction execution count



Device Memory

Reads	1525804	11.088 GB/s	
Writes	4130626	30.018 GB/s	
Total	5656430	41.106 GB/s	Idle Low Medium High Max

(d) Memory bandwidth occupancy

Figure 6.14: Evaluation of our implementation of the sub step create-radiance-image in Reinhard et al.'s algorithm

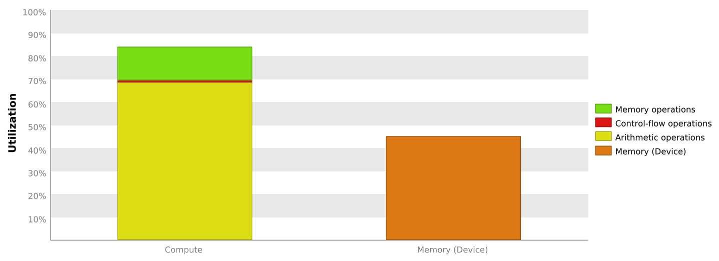
FFT foward: The next kernel we are going to examine was not implemented by us. Instead it is an operation offered by CUFFT. This algorithm requires multiple forward and inverse Fourier transforms. Here we picked Nvidia's implementation of a forward `FFT` to examine. Figure 6.15a shows that the performance of this operations is limited by arithmetic instruction throughput.

When examining the load on the different modules of the `GPU` (Figure 6.15b), one can see that it is not evenly distributed. By far the highest demand is put on the single precision floating point unit. Because the load is very high the implementation is very efficient without saturating the floating point unit. This indicates a constant, high throughput by that module.

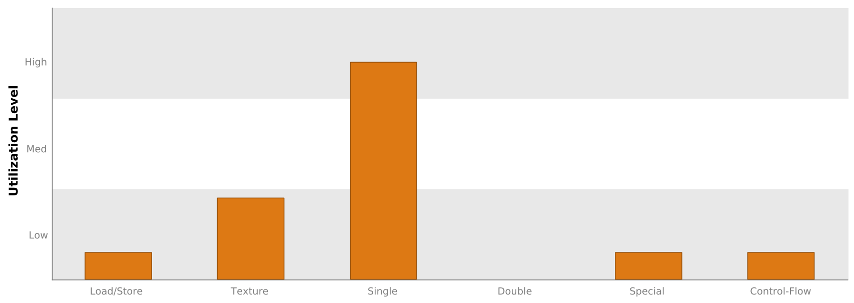
This observation is further supported by Figure 6.15c. Approximately 5% of executed instructions were idle instructions. This means that sufficient parallelism was achieved for this operation. This figure furthermore shows that a large number of operations are integer operations. On a modern `GPU` , integer operations are slower than floating point operations. This is due to the fact that there are more `FPU` (floating point unit) than `ALU` (arithmetic logic unit) units on a `GPU` .

Figure 6.15d shows the memory bandwidth utilisation. A high throughput to L1-cache can be observed. This indicates a high amount of data sharing between threads of the same warp using shared memory.

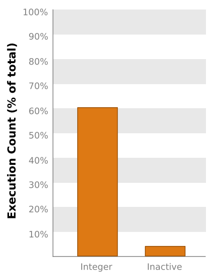
Execution time: 4.297 ms.



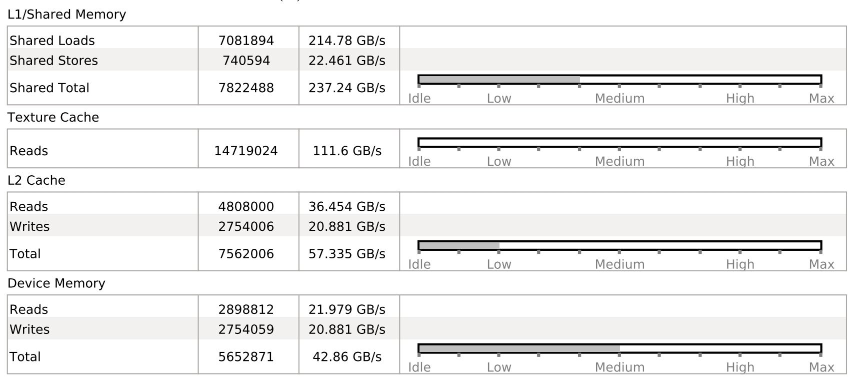
(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the `GPU`



(c) Instruction execution count



L1/Shared Memory

Shared Loads	7081894	214.78 GB/s	
Shared Stores	740594	22.461 GB/s	
Shared Total	7822488	237.24 GB/s	Idle Low Medium High Max

Texture Cache

Reads	14719024	111.6 GB/s	Idle Low Medium High Max

L2 Cache

Reads	4808000	36.454 GB/s	
Writes	2754006	20.881 GB/s	
Total	7562006	57.335 GB/s	Idle Low Medium High Max

Device Memory

Reads	2898812	21.979 GB/s	
Writes	2754059	20.881 GB/s	
Total	5652871	42.86 GB/s	Idle Low Medium High Max

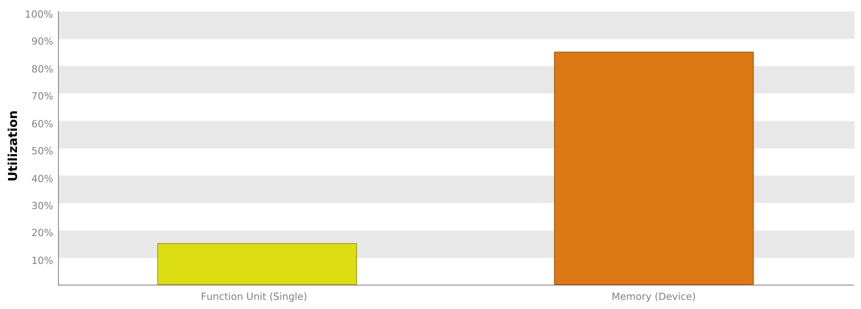
(d) Memory bandwidth occupancy

Figure 6.15: Evaluation of our implementation of the sub step fft-forward in Reinhard et al.'s algorithm

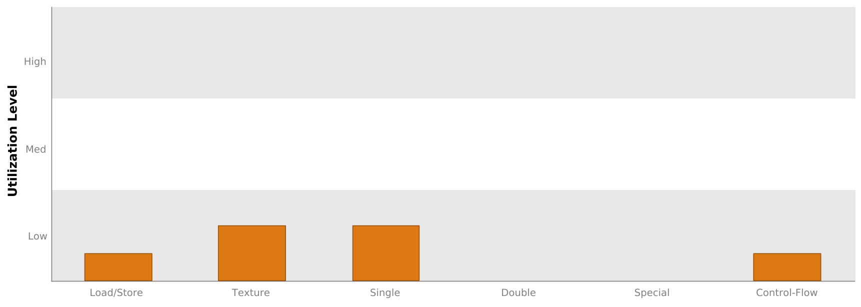
Process complex numbers: The next kernel we want to analyse deals with multiplying complex numbers. This kernel is used to convolve the input image with the different Gauss kernels. For this, we performed a forward `FFT` transformation on both the incoming image and the Gauss kernels. Then for each pixel position the input is multiplied with the Gauss kernel.

Figure 6.16a shows that this kernel is severely limited by memory bandwidth. This is caused by the fact that for each pixel two read operations and one write operation have to be performed. Compared to this, the arithmetic operations performed are very simple and consist only of additions and multiplications. These operations can be performed very efficiently on a `GPU` .

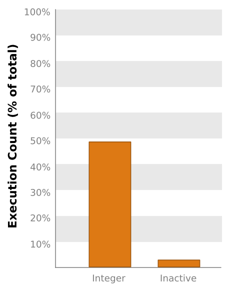
This fact is further supported by two figures. Firstly, Figure 6.16b, showing the utilisation of the different modules on the `GPU` . The load is evenly spread on different modules and not very high on any of them. In contrast to this, Figure 6.16d shows a very high demand on global device memory. Figure 6.16e shows that the major cause of stalling is instruction dependency. This further supports the notion that reading the required data is the bottleneck of these operations. Many operations had to wait for data that was being read in the previous operation. Despite this one sided utilisation of the `GPU` , we achieve high parallelism. This can be seen in Figure 6.16c. Only 3% of commands executed were idle instructions. Another observation in this figure is the fact that a lot of operations are integer operations. As mentioned in the previous kernel analysis, integer operations are comparably slow operations on a `GPU` . Operations related to calculating addresses in a linear array are the cause for the high number of integer operations in this kernel.
Execution time: 3.568 ms.



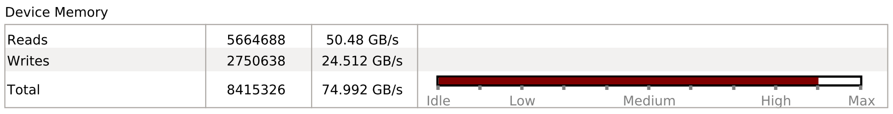
(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the `GPU`



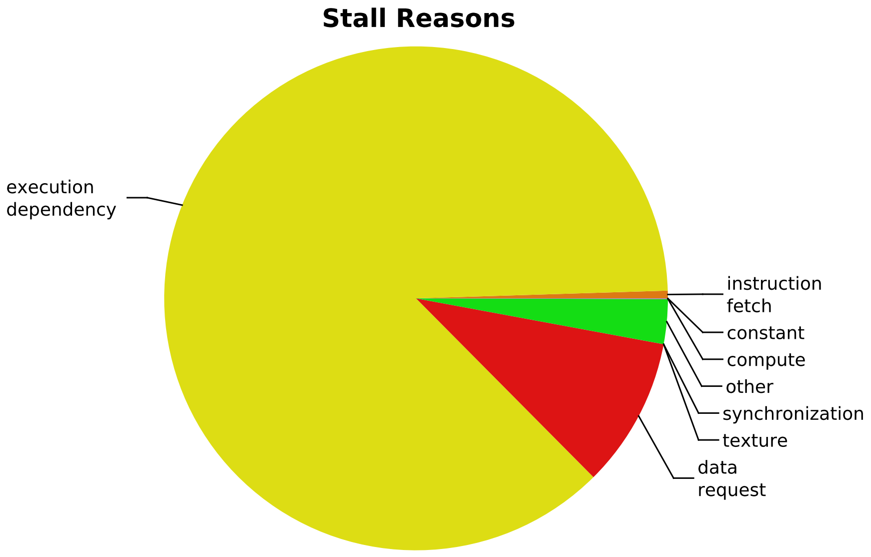
(c) Instruction execution count



Device Memory

Reads	5664688	50.48 GB/s	
Writes	2750638	24.512 GB/s	
Total	8415326	74.992 GB/s	Idle Low Medium High Max

(d) Memory bandwidth occupancy



(e) Computation stall reasons

Figure 6.16: Evaluation of our implementation of the sub step about processing complex numbers in Reinhard et al.'s algorithm

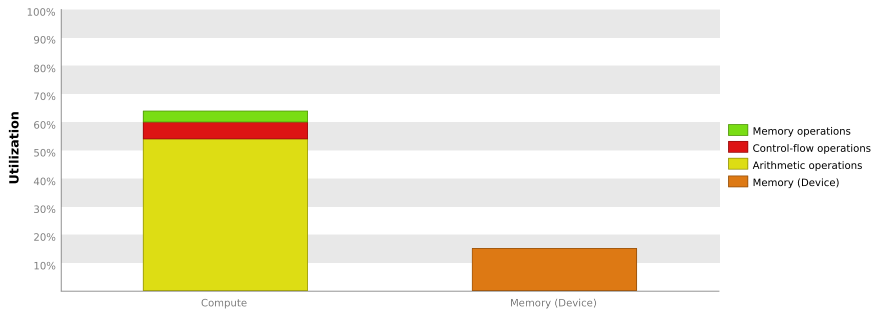
Tone mapper: The last kernel of this algorithm we want to examine is the final step in the algorithm where the actual dynamic compression is performed. This is done by first picking the correct size of Gauss kernel for each pixel and then calculating the equation introduced in Section 4.2.3.1.

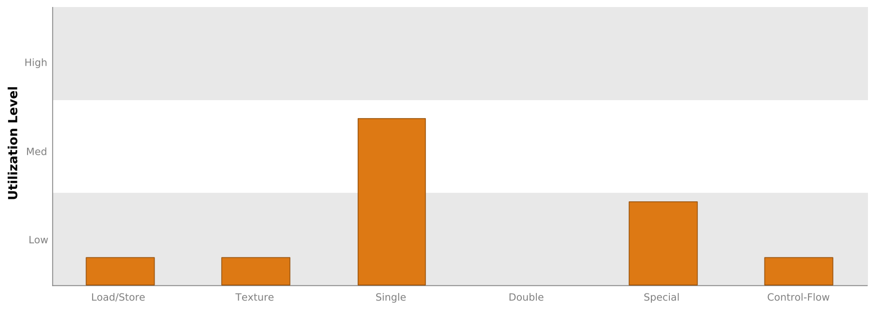
Figure 6.17a reveals that the performance of this algorithm is limited by the number of arithmetic operations performed. The reason for this can be seen in Figure 6.17b. It shows that the load on the different modules of the GPU is not evenly distributed. Most of the load is put on the single precision arithmetic unit. This can lead to a bottleneck that degrades overall performance.

In Figure 6.17c we can see that about 10% of the instructions executed were idle operations. This is most likely caused by the one sided utilisation of the GPU .

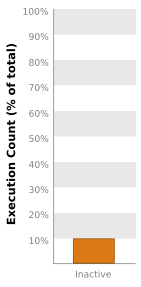
Execution time: 8.677 ms.

(a) comparison of time spent on computations versus memory I/O





(b) comparison of the utilisation of the different modules on the GPU



(c) Instruction execution count

Figure 6.17: Evaluation of our implementation of the sub step applying the formula introduced in Section 4.2.3.1 in Reinhard et al.'s algorithm

6.1.3.8 Common

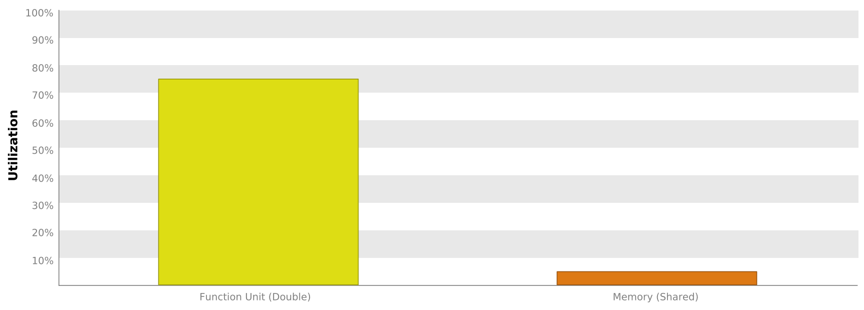
In the following section we will perform the detailed analysis of a helping kernel that is used by multiple algorithms. This kernel calculates the average brightness of an incoming image. We use this kernel in the algorithms proposed by Ward, Larson et al., and Reinhard et al.. For this purpose we employed a heavily optimized version of parallel reduction proposed by Harris et al. in [15]. We calculate the logarithm of the average brightness using the equation introduced in Section 4.2.1.1.

Figure 6.18a shows that the performance of this algorithm is limited by the number of arithmetic operations performed. The reason for this is twofold. First, the logarithm of sums needs to be calculated in each iteration . This is computationally rather expensive. Memory consumption on the other hand is reduced by using local memory for all operations apart from the initial read. The combination of these two factors leads to high demand for arithmetic operations coupled with low memory consumption.

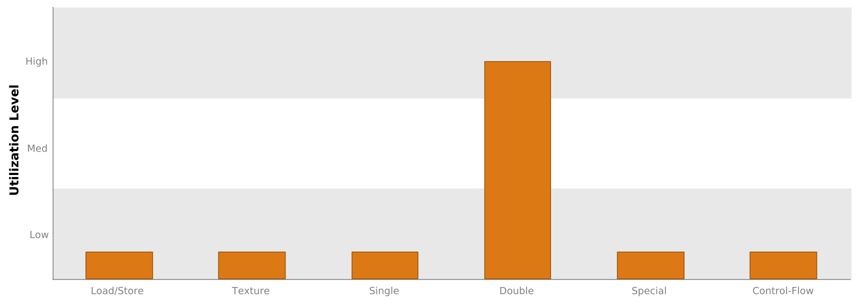
According to Figure 6.18b, the highest strain is put on the double precision floating point unit. We use double precision during this operation. This is done because the sum during this operation tends to get very large. We only perform the division by the number of pixels in the end after adding everything up. In order not to lose precision for large numbers we elected to use double precision.

In Figure 6.18c we can observe that the amount of idle operations lies around 8%. Possible reasons for this can be seen in Figure 6.18d. The primary reason for this is execution dependency. This is caused by the nature of parallel reduction. The more iterations have been performed, the fewer threads are active. For example in the last iteration only a single thread is adding up the last two remaining elements.

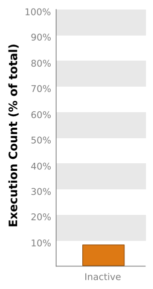
Execution time: 12.365 ms.



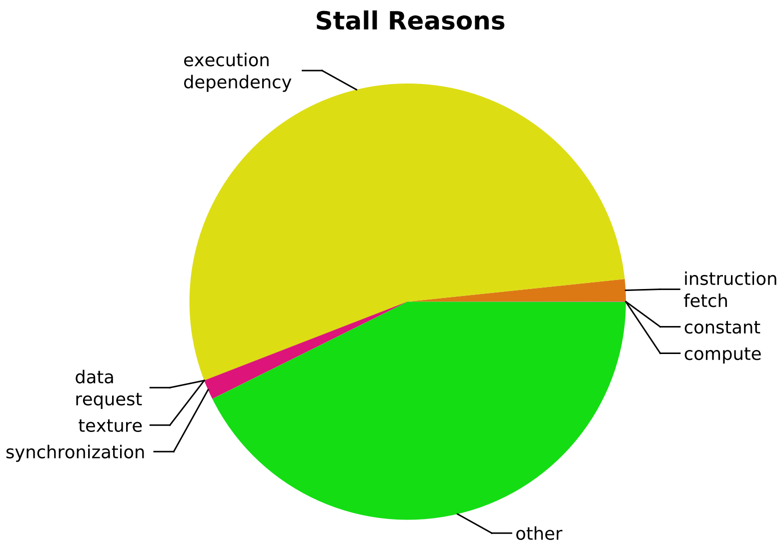
(a) comparison of time spent on computations versus memory I/O



(b) comparison of the utilisation of the different modules on the GPU



(c) Instruction execution count



(d) Stall reasons

Figure 6.18: Evaluation of our implementation of parallel reduction to recover log-average brightness

6.1.3.9 Overview

An overview of the execution times of the kernels mentioned above can be seen in Table 6.3.

Kernel	Execution Time
Debevec	7.737 ms
Robertson - radiance mapper	11.209 ms
Tocci - radiance mapper	30.234 ms
Ward - tone mapper	7.758 ms
Larson - compute foveal image	4.459 ms
Larson - calibrate histogram	0.109 ms
Larson - apply histogram	9.69 ms
Reinhard create-radiance-image	4.732 ms
Reinhard `FFT` -forward	4.297 ms
Reinhard process-complex-numbers	3.568 ms
Reinhard tone mapper	8.677 ms
Common - log average luminance	12.365 ms

Table 6.3: Execution times of all kernels in ms

6.2 Visual Comparison

We performed a subjective visual comparison of the output of our algorithms. This was done in order to find algorithms that are pleasing to the eye of the viewer. Many studies on the quality of tone-mappers have been performed ([25], [6], [30], [35]), but at our best knowledge no study paired up different radiance mappers with tone mappers. For a more complete list of comparisons of tone mappers we refer to Petit and Mantiuk [25]. The output images of the different combinations of algorithms can be seen in Figure 6.20.

6.2.1 Set-Up

We perform the evaluation of quality without a real world reference. This is because we wanted the content shown to resemble the real use-case. For this reason we were showing recordings of the football stadium in Tromsø. We had no `HDR` monitor available that could substitute for a real world reference either. We used a standard 24" TFT-monitor that was set up in a dark room with controlled environmental luminance levels that exhibited no direct light.

6.2.2 Proceedings

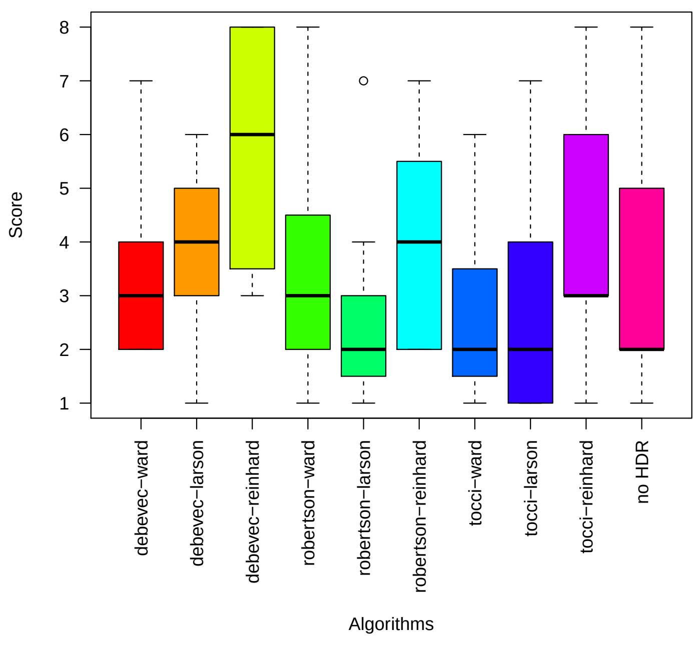
The different video sequences were shown to 11 participants individually. We based our proceedings on the methodology proposed by Petit and Mantiuk in [25]. Before reviewing the sequences, participants were told to pick the video they "liked the best" and "found the most visually pleasing". They furthermore were told not to look at only a single feature, but rate the overall impression. Nine thumbnails of the different combinations were shown on the screen. We also included a tenth video that was captured without performing HDR processing. This was used as an LDR video to be compared with the others. The participants were not aware of the fact that one of the videos was captured using traditional LDR techniques. Upon clicking on one of the thumbnails the corresponding video clip was shown in full screen using VLC media player *. The resolution of the panoramic video did not match the resolution of the monitor used, therefore downscaling was performed by VLC. The participants then had to rate each video in the following categories: Colour reproduction, contrast, brightness, detail reproduction, and overall pleasantness. We adopted those categories from [6]. In order to ensure that each video has been seen, the users could only rate a video after it has been watched. In the following section we will perform an analysis of the results obtained by this study.

6.3 Results of subjective study

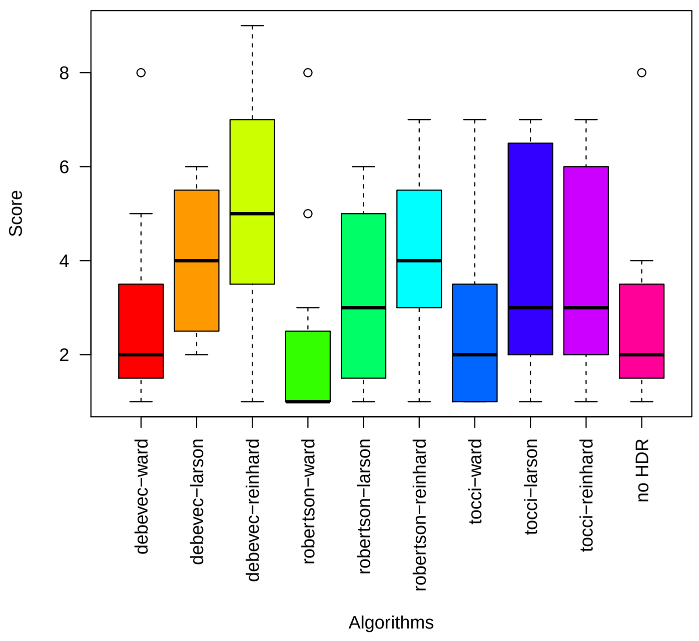
By comparing the charts in Figure 6.19e, we can observe several emerging patterns. The first one is that the tone-mapper proposed by Reinhard et al. outperforms the other tone-mappers we implemented. For each possible combination with the implemented radiance-mappers it scored the most points in all categories. A similar trend can be observed for the two remaining tone-mapping operators. In most of the cases the one proposed by Larson et al. outperforms the tone-mapper suggested by Ward. Only when paired up with the radiance-mapper proposed by Robertson et al., the ordering between the two tone-mappers was reversed. Another trend can be observed when comparing the performance of the implemented radiance-mappers. The radiance-mapper proposed by Debevec and Malik scored best among the participants of the study. The scores given to this radiance-mapper outperform the other radiance-mappers when paired with either the tone-mapper proposed by Larson et al., or Robertson et al.. Only when using the tone-mapper proposed by Ward, the radiance-mapper proposed by Tocci et al. scored more points. To determine whether HDR processing helps improve the quality of the captured video, we compare the scores of our various radiance- and tone-mapper configurations with an LDR video. Here we see that most of the algorithms perform on par, or better than the LDR video. When only looking at the scores in the overall pleasantness category (Figure 6.19e), we can observe

*http://www.videolan.org/vlc/index.html

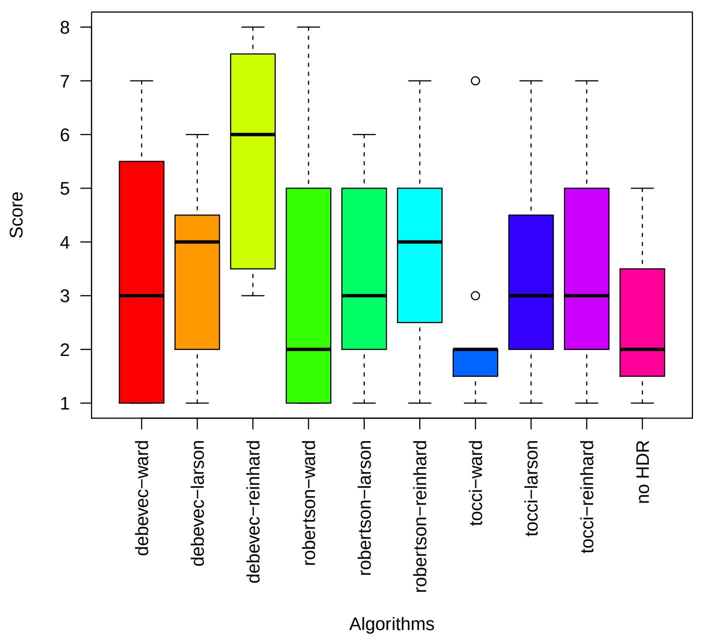
that only a single combination scored worse than the LDR video. This is when pairing up the radiance-mapper proposed by Robertson et al. with the tone-mapper suggested by Ward. We think this is because the formula used in the radiance-mapper emphasises long exposures. This leads to a higher brightness of the HDR output image. The mostly linear compression performed by the tone-mapper suggested by Ward leads to undesirable results with the increased dynamic range of this HDR input. One further observation is concerned with how the scores differ between the categories. The average score is lowest in the contrast category. Here the participants were asked to judge the existing contrast in the picture. Low scores were to be given if the contrast was either too high or too low. We see two possible reasons for the poor results of our HDR processing algorithms in this domain. The first reason is related to the fact that most tone-mapping operators try to reduce contrast in order to minimize the overall dynamic range. We believe that the resulting low contrast image appeared too "bland" to the participants. The second reason lies in the nature of the tone-mapping algorithm proposed by Ward. Nearly linear dynamic range compression is performed here in order to preserve the relative contrast of the image. If the dynamic range of the incoming HDR image is very high, this leads to only the very dark and bright parts being visible. This results in an image where large parts are either mostly over- or under-saturated. This causes the contrast to be perceived as too high by our participants.



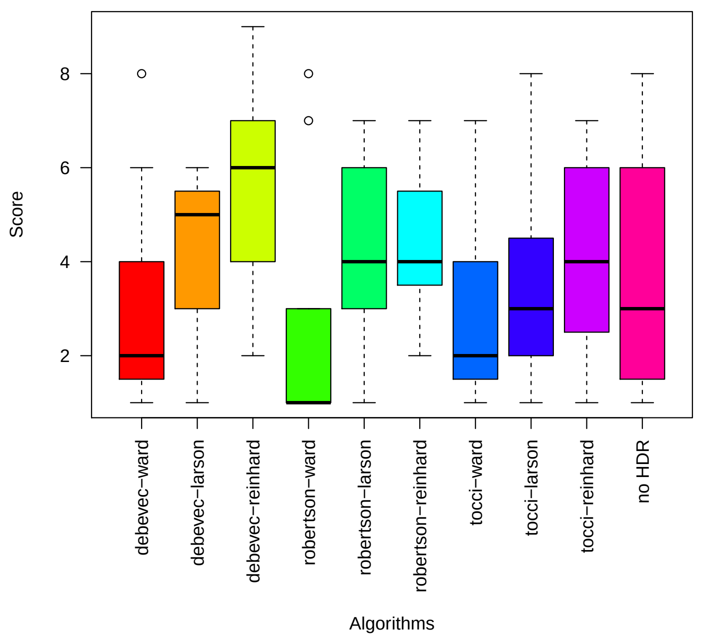
(a) Result for the category: Colour



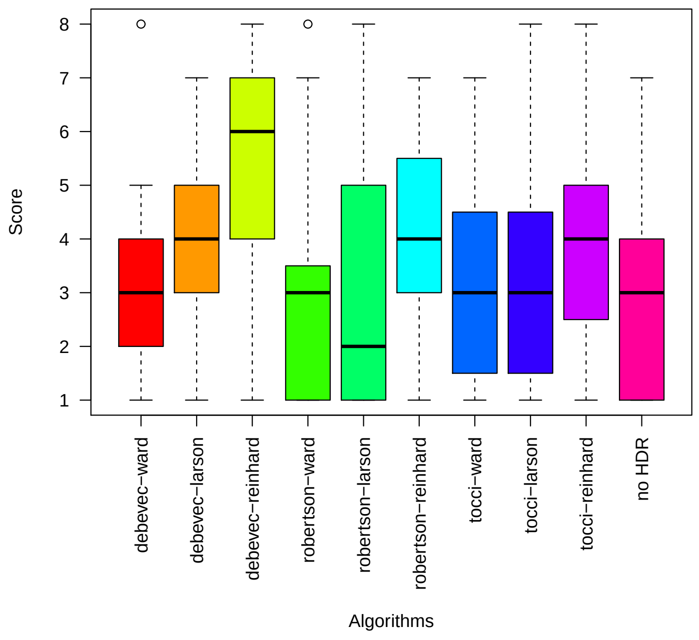
(b) Result for the category: Contrast



(c) Result for the category: Brightness



(d) Result for the category: Detail reproduction



(e) Result for the category: Overall pleasantness

Figure 6.19: Results of the subjective user study

6.4 Performance vs. Visual Result trade-off

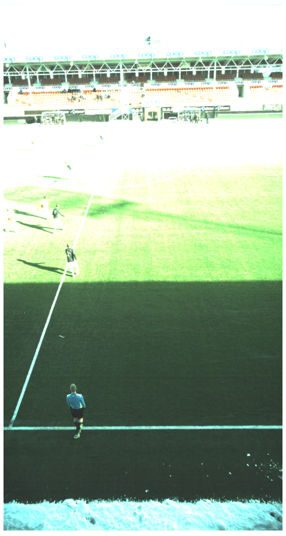
When only looking at the results of the visual study the ideal combination of algorithms is clear. When using the radiance-mapper proposed by Debevec and Malik in conjunction with the tone-mapper suggested by Reinhard et al., the participants of the study gave an average score that was 50% higher than the next best combination. When looking at the execution times of the algorithms such a clear verdict can not be given. The tone-mapper introduced by Reinhard et al. barely passes the real time threshold when using only a single Gauss kernel. When the whole pipeline is active the algorithm is too slow, even when only using a single Gauss kernel. The next most visually pleasing combination uses the same radiance mapper, but uses the tone-mapper proposed by Larson et al.. When using this combination the execution times are well below the real-time threshold.

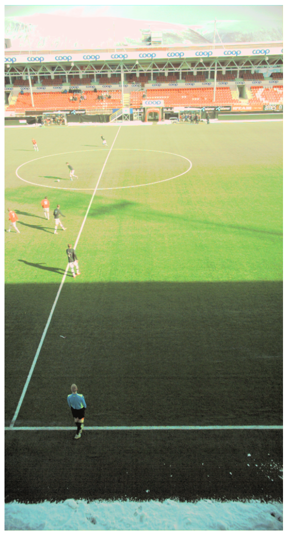
We want to answer the question if the higher visual quality of Reinhard et al.'s tone-mapper can outweigh the disadvantage of a longer execution time. To accomplish this we perform a naïve analysis to compare the two tone-mapping operators. By taking execution time and visual study score into account we can determine a ratio between score and execution time. When only using a single Gauss kernel the tone-mapper by Reinhard et al. achieves a ratio of 1:8. This means that 8ms of execution time equal one point scored in the visual study. The tone-mapper suggested by Larson et al. accomplishes a ratio of 1:8.5. Using this simple metric, we come to the conclusion that the tone-mapper proposed by Larson et al.is better suited for our application.

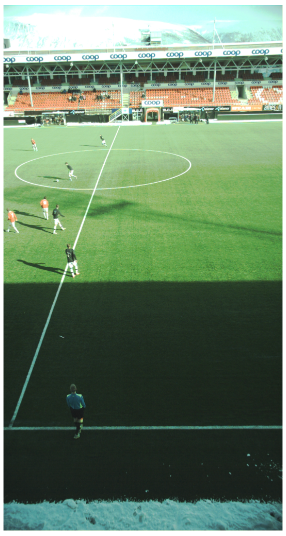
6.5 Final Result

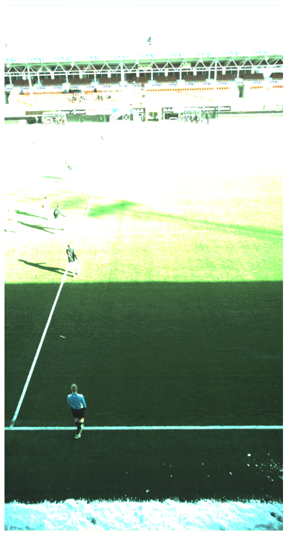
We will briefly explain the final combination we settled on to use in the pipeline in the future. As a radiance-mapper we selected the one proposed by Debevec and Malik. From a performance standpoint, this algorithm proved to be the fastest. Furthermore, the results also turned out to be the most visually pleasing. Therefore we conclude that this radiance mapper is the ideal solution for our needs. Picking a corresponding tone-mapper proved to be more difficult. The one proposed by Larson et al. exhibits good performance characteristics. Visually it performs well in respect to brightness compression. But image noise is greatly boosted by performing histogram equalisation. This is a major downside of this algorithm. The visual results of the tone mapper introduced by Reinhard et al. were also very pleasing. Brightness was not compressed as much, but the resulting image was perceived as more natural by our test participants. The downside of this algorithm is its runtime. When using only a single Gauss kernel we can barely accomplish the real-time threshold as can be seen in Figure 6.12. Using only one kernel turns this tone mapper from a local to a global tone mapper. In our opinion, this defeats the purpose of the tone-mapper by forgoing “dodging & burning”. For this reason we favour the tone mapper suggested by Larson et al.. In the future, when more powerful hardware is available we will re-evaluate using the tone mapper proposed by Reinhard et al.. If computing power doubles, this algorithm will probably prove to be a good choice.

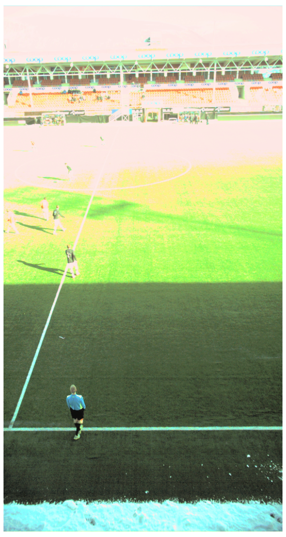
Eventually we settled on the combination of the radiance-mapper by Debevec and Malik in conjunction with the tone-mapper put forward by Larson et al..

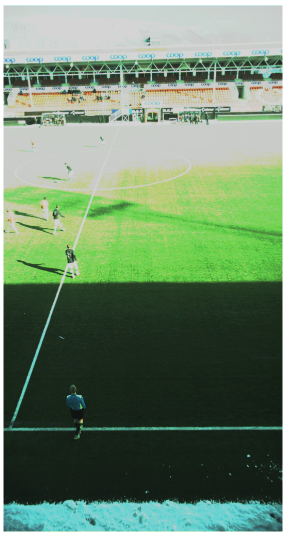

(a) Debevec - Ward

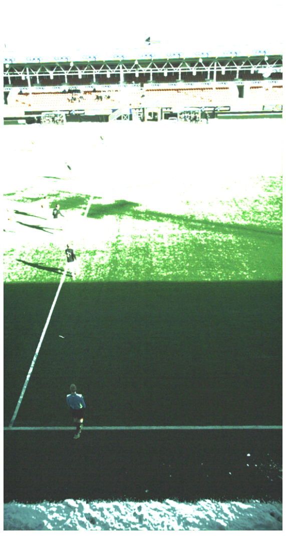

(b) Debevec - Larson

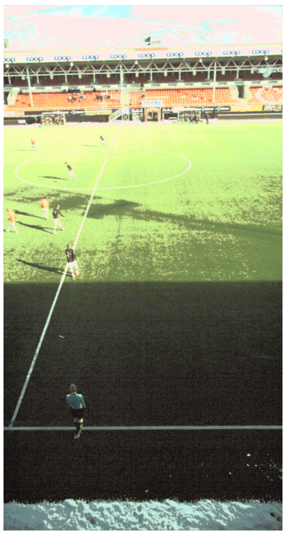

(c) Debevec - Reinhard

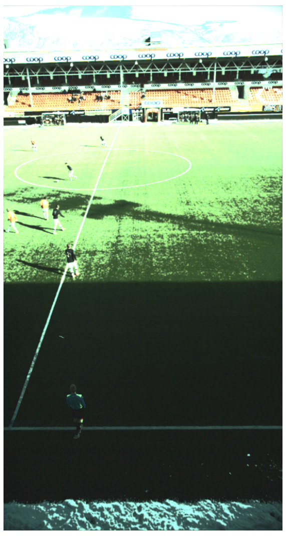

(d) Robertson - Ward


(e) Robertson - Larson


(f) Robertson - Reinhard


(g) Tocci - Ward


(h) Tocci - Larson



(i) Tocci - Reinhard

Figure 6.20: Output images acquired using different combinations of algorithms

Chapter 7

Conclusion

7.1 Future Work

After this initial foray into real-time HDR processing in the context of high resolution panoramic videos, we want to refine our work in several areas. During our work we mostly focused on implementing as many algorithms as possible. This means that due to time constraints some possible optimisations could not be performed. After selecting the best combination of algorithms in Section 6.5, we want to perform an additional optimization pass on those selected algorithms. Furthermore, we want to investigate ways to improve our implementation of the algorithm put forth by Reinhard et al.. This algorithm performed rather well in our subjective visual study, but our implementation could not perform the necessary operations quick enough. Even in the fastest configuration we could not pass the real-time threshold. After bringing this work to a close, we became aware of a more effective implementation. This optimisation warrants further investigation in the future.

Another important step towards a more autonomous video capturing system is automatic determination of an optimal scene key brightness value. In the current system the scene key brightness is set manually at the beginning of a recording. We want to investigate ways of implementing a feedback loop that enables us to set the optimal scene key brightness based on the output of our pipeline.

7.2 Concluding Remarks

The goal of this work was to determine whether high-resolution, real-time HDR processing is possible with the current hardware available. For this purpose we implemented several algorithms in Nvidia's CUDA. We extended an existing real-time panoramic video pipeline with an HDR module. Within this module we experimented with various combinations of our implemented algorithms. We performed extensive performance analysis on our HDR processing module to determine how well the implemented algorithms are suited for massive parallel execution as it is performed on a GPU . We also perform a user study to determine which algorithms lead to visually pleasing results. We conclude the work with finding the most preferable combination of algorithms and thus proving the feasibility of our approach.

Appendix A

Acronyms and Symbols

List of Acronyms

`HDR`	high dynamic range
`LDR`	low dynamic range
`FOV`	field of view
`CDF`	cumulative distribution function
`FFT`	fast Fourier transform
`SVD`	singular value decomposition
`ALU`	arithmetic logic unit
`FPU`	floating point unit

Bibliography

[1] Adams, A. (1980). The camera, the ansel adams photography series. *Little, Brown and Company.*

[2] Adams, A. and Baker, R. (1981). *The negative.* New York Graphic Society.

[3] Adams, A. and Baker, R. (1983). *The print.* Little, Brown.

[4] Bayer, B. E. (1976). Color imaging array. US Patent 3,971,065.

[5] Benoit, A., Alleysson, D., Herault, J., and Le Callet, P. (2009). Spatio-temporal tone mapping operator based on a retina model. In *Computational Color Imaging*, pages 12–22. Springer.

[6] Čadík, M., Wimmer, M., Neumann, L., and Artusi, A. (2006). Image attributes and quality for evaluation of tone mapping operators. In *National Taiwan University*. Citeseer.

[7] Debevec, P. E. and Malik, J. (2008). Recovering high dynamic range radiance maps from photographs. In *ACM SIGGRAPH 2008 classes*, page 31. ACM.

[8] Drago, F., Myszkowski, K., Annen, T., and Chiba, N. (2003). Adaptive logarithmic mapping for displaying high contrast scenes. In *Computer Graphics Forum*, volume 22, pages 419–426. Wiley Online Library.

[9] Fattal, R., Lischinski, D., and Werman, M. (2002). Gradient domain high dynamic range compression. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 249–256. ACM.

[10] Gaddam, V. R., Langseth, R., Ljødal, S., Gurdjos, P., Charvillat, V., Griwodz, C., and Halvorsen, P. (2014). Interactive zoom and panning from live panoramic video. In *NOSSDAV*, page 19.

[11] Gauss, C., Schering, E., Brendel, M., and der Wissenschaften in Göttingen, A. (1903). *Carl Friedrich Gauss Werke ...* Carl Friedrich Gauss Werke. Gedruckt in der Dieterichschen Universitätsdruckerei (W.F. Kaestner).

[12] Granados, M., Ajdin, B., Wand, M., Theobalt, C., Seidel, H.-P., and Lensch, H. (2010). Optimal hdr reconstruction with linear digital cameras. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 215–222. IEEE.

[13] Guthier, B., Kopf, S., Eble, M., and Effelsberg, W. (2011). Flicker reduction in tone mapped high dynamic range video. In *IS&T/SPIE Electronic Imaging*, pages 78660C–78660C. International Society for Optics and Photonics.

[14] Guthier, B., Kopf, S., and Effelsberg, W. (2013). Algorithms for a real-time hdr video system. *Pattern Recognition Letters*, 34(1):25–33.

[15] Harris, M. et al. (2007). Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2:45.

[16] Jacobi, C. (1846). Ueber ein leichtes verfahren, die in der theorie der saekularstoerungen vorkommenden gleichungen numerisch aufzuloesen. *Journal f$\tilde{A}\frac{1}{4}$r reine und angewandte Mathematik*, 30:51–95.

[17] Krawczyk, G., Myszkowski, K., and Seidel, H.-P. (2005). Perceptual effects in real-time tone mapping. In *Proceedings of the 21st spring conference on Computer graphics*, pages 195–202. ACM.

[18] Larson, G. W., Rushmeier, H., and Piatko, C. (1997). A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Trans. on Visualization and Computer Graphics*, 3(4):291–306.

[19] Madden, B. C. (1993). Extended intensity range imaging. *Technical Reports (CIS)*, page 248.

[20] Mann, S. (2000). Comparametric equations with practical applications in quantigraphic image processing. *Image Processing, IEEE Transactions on*, 9(8):1389–1406.

[21] Mann, S. and Mann, R. (2001). Quantigraphic imaging: Estimating the camera response and exposures from differently exposed images. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–842. IEEE.

[22] Mann, S. and Picard, R. (1994). *Being undigital with digital cameras.* MIT Media Lab Perceptual.

[23] Mantiuk, R., Daly, S., and Kerofsky, L. (2008). Display adaptive tone mapping. In *ACM Transactions on Graphics (TOG)*, volume 27, page 68. ACM.

[24] Mitsunaga, T. and Nayar, S. K. (1999). Radiometric self calibration. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 1. IEEE.

[25] Petit, J. and Mantiuk, R. K. (2013). Assessment of video tone-mapping: Are cameras s-shaped tone-curves good enough? *Journal of Visual Communication and Image Representation*, 24(7):1020–1030.

[26] Reinhard, E., Stark, M., Shirley, P., and Ferwerda, J. (2002). Photographic tone reproduction for digital images. *ACM Trans. on Graphics*, 21(3):267–276.

[27] Robertson, M. A., Borman, S., and Stevenson, R. L. (2003). Estimation-theoretic approach to dynamic range enhancement using multiple exposures. *Journal of Electronic Imaging*, 12(2):219–228.

[28] Schubert, E. F., Gessmann, T., and Kim, J. K. (2005). *Light emitting diodes*. Wiley Online Library.

[29] Tocci, M. D., Kiser, C., Tocci, N., and Sen, P. (2011). A versatile hdr video production system. 30(4):41.

[30] Villa, C. and Labayrade, R. (2010). Psychovisual assessment of tone-mapping operators for global appearance and colour reproduction. *Proc. of Colour in Graphics Imaging and Vision*, 2.

[31] Ward, G. (1991). Real pixels. *Graphics Gems II*, pages 80–83.

[32] Ward, G. (1994). A contrast-based scalefactor for luminance display. *Graphics gems IV*, pages 415–421.

[33] Xiong, Y. and Pulli, K. (2010). Fast panorama stitching for high-quality panoramic images on mobile phones. *Consumer Electronics, IEEE Transactions on*, 56(2):298–306.

[34] Yamada, K., Nakano, T., Yamamoto, S., Akutsu, E., and Aoki, I. (1994). Wide dynamic range vision sensor for vehicles. In *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pages 405–408. IEEE.

[35] Yoshida, A., Blanz, V., Myszkowski, K., and Seidel, H.-P. (2005). Perceptual evaluation of tone mapping operators with real-world scenes. In *Electronic Imaging 2005*, pages 192–203. International Society for Optics and Photonics.