Parallel Implementation of a Real-Time High Dynamic Range Video System

Benjamin Guthier[a,*], Stephan Kopf[a], Matthias Wichtlhuber[b] and Wolfgang Effelsberg[a]
[a]*Department of Computer Science IV, University of Mannheim, 68131 Mannheim, Germany*
[b]*Peer-to-Peer Systems Engineering, University of Darmstadt, 64283 Darmstadt, Germany*

Abstract. This article describes the use of the parallel processing capabilities of a graphics chip to increase the processing speed of a high dynamic range (HDR) video system. The basis is an existing HDR video system that produces each frame from a sequence of regular images taken in quick succession under varying exposure settings. The image sequence is processed in a pipeline consisting of: shutter speeds selection, capturing, color space conversion, image registration, HDR stitching, and tone mapping. This article identifies bottlenecks in the pipeline and describes modifications to the algorithms that are necessary to enable parallel processing. Time-critical steps are processed on a graphics processing unit (GPU). The resulting processing time is evaluated and compared to the original sequential code. The creation of an HDR video frame is sped up by a factor of 15 on the average.

Keywords: Real-Time, High Dynamic Range Video, Parallel Processing, GPU

1. Introduction

A recurring problem in capturing video is the scene having a range of brightness values that exceed the capabilities of the capturing device. An example would be a video camera in a bright outside area, directed at the entrance of a building (e.g., see Figure 1). Because of the potentially big brightness difference, it may not be possible to capture details of the inside of the building and the outside simultaneously using just one shutter speed setting. This results in under- and overexposed pixels in the video footage. The approach used in this article to overcome this problem is temporal exposure bracketing, i.e., using a set of images captured in quick sequence at different shutter settings. Each image then captures one facet of the scene's brightness range. When fused together, a high dynamic range (HDR) video frame is created that reveals details in dark and bright regions simultaneously.

The process of creating a frame in an HDR video can be thought of as a pipeline where the output of each step is the input to the subsequent one. It begins by capturing a set of regular images using varying shutter speeds. For easier processing, the images are converted into a different color space. Next, they are aligned with respect to each other to compensate for camera motion during capture. The aligned images are then stitched together to create a single HDR frame containing accurate brightness values of the entire scene. As a last step, the HDR frame is tone mapped in order to be displayable on a regular screen with a lower dynamic range.

It is desirable to perform all necessary steps from capturing of the low dynamic range (LDR) images to displaying of the HDR frame in real-time. The result is a live HDR video that can be viewed instantaneously. Example scenarios for such a video are a surveillance camera monitoring the entrance to a building or an advanced driver assistance system that works even in difficult lighting situations like the exit of a tunnel. If the goal of the real-time HDR video system is to achieve a rate of *25* frames per second, all the steps of the HDR pipeline must be completed within *40 ms*. This requirement necessitates a high frame-rate camera, fast algorithms for processing of the frames, and a fast implementation.

[*]Corresponding author. E-mail: guthier@informatik.uni-mannheim.de

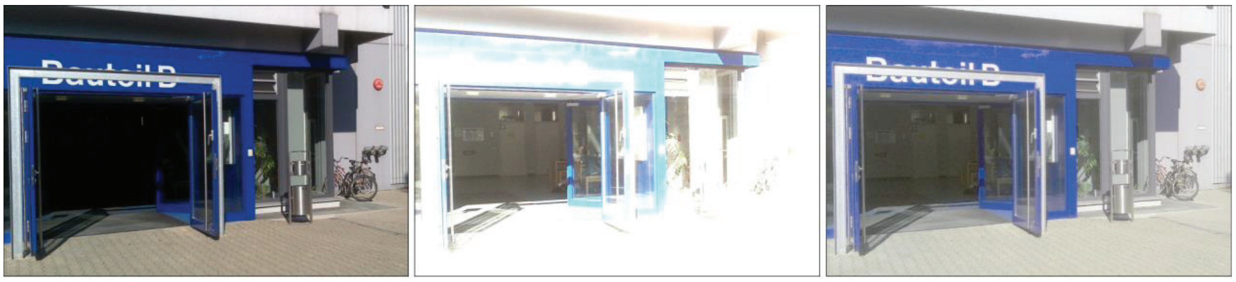

Fig. 1. The inside of the building is much darker than the outside. There is no shutter speed setting that exposes both correctly at the same time. A shorter shutter underexposes the inside of the building (left) while a longer shutter overexposes the outside (center). A solution to this problem is merging the two frames into one HDR frame (right).

The basis for the work described in this article is an existing HDR video system [12]. It implements all steps of the HDR video pipeline. The employed algorithms are optimized for reduced capturing costs and fast processing speed. They are implemented as strictly sequential code running on a CPU. This article focuses on a parallel implementation of the steps of the pipeline. It makes use of the parallel processing capabilities of a modern graphics processing unit (GPU). The article begins with an introduction to GPU programming in Section 3. A basic understanding of the underlying concepts is necessary to comprehend the design decisions made in the later sections. Section 4 gives an overview of the HDR video system to be optimized, outlining the steps of the HDR pipeline and their major subtasks. Because implementing an algorithm to run on a GPU can be difficult, it is preferable to focus on the subtasks that benefit from parallelization the most. Considerations regarding the necessity and feasibility of a parallel implementation are given in Section 5. It also describes the modifications to the identified subtasks that were necessary to process them in a parallel way. The performance of the parallel implementation of the HDR video system is evaluated in a realistic scenario and compared to the existing CPU implementation in Section 6. The section also discusses the system's run-time behavior when changing the image size or the number of captured LDR exposures.

2. Related work

The fast increase of the computational power of personal computers has made the development of real-time multimedia systems possible that even allow complex video analysis and video processing tasks [1]. Parallelization of processing tasks is especially useful in the case of low-level image processing algorithms [2].

The goal of our system is to achieve a rate for the resulting HDR video of 25 frames per second. We defined an upper limit of eight low dynamic range exposures per HDR frame. This means that our system must be capable of processing 200 LDR frames per second in the worst case. Such high frame rates necessitate the use of efficient implementations. Many of the necessary computations are inherently parallel. Oftentimes a simple arithmetic operation must be applied to a large number of pixels. Contrary to a CPU, which is optimized for executing a low number of complex tasks, GPUs are capable of processing large numbers of comparably simple tasks, which makes them suitable for image and video processing tasks.

Examples of works in the area of GPU implementations are [6, 17, 18]. Using a GPU, it is even possible to run complex algorithms like graph cuts in real-time as presented by Lattari et al. [17].

Riego et al. [20] have developed a virtual 3D interface which computes the motion analysis on a GPU. They use a parallelized hierarchical Lucas-Kanade (HLK) algorithm to calculate the optical flow. In contrast to our approach, the authors only consider one computation step (the estimation of the optical flow) without considering the other steps in detail. This approach is applicable, because their required 30 frames per second are significantly lower compared to the requirements of our system.

Van den Braak et al. [5] demonstrate how GPUs can be used to speed up voting algorithms like the computation of histograms or the Hough transform. These algorithms are difficult to parallelize due to their memory access pattern. Our implementation of the histogram computation is very similar to their work.

An efficient implementation of Bayer demosaic filtering on GPUs was published in [18]. The presented OpenGL implementation of the Malvar-He-Cutler filter is two to three times faster than a straightforward GPU implementation.

Some recent work has been published in the area of HDR video. Creating HDR videos typically consist of four steps: capturing [7, 13, 21], LDR image registration [8, 15], merging LDR frames into an HDR frame [23], and tone mapping [3, 4, 9, 16, 19].

A popular technique to create HDR images is using a set of LDR images captured in quick sequence at different exposure settings. The most challenging problem is the estimation of the inverse camera response function to map pixel values onto scene radiance [7, 21].

Image registration may be avoided if specific hardware is used as presented in [23]. The authors have developed a system for capturing HDR video in cinema quality. They focus on an optical sensor that allows the capturing of three LDR frames with different exposure settings simultaneously. Major limitations of their approach are the fixed number of exposures and the high hardware cost for the specialized sensors.

Several techniques have been proposed to determine suitable exposure settings. Hasinoff et al. present an approach to determine noise-optimal exposure settings by using varying gain levels [13]. For a given sum of exposure times, increasing gain also increases the SNR. The proposed computation of the exposure settings is too expensive to be used in a real-time scenario.

Only few image registration techniques are able to handle the lighting differences of LDR frames with different exposure settings. Kang et al. propose a method for estimating camera and scene motion, but its computational cost is too high to be used in real-time [15]. In previous work, we have proposed a fast registration algorithm based on threshold images [8].

The tone mapping step converts radiance values back to suitable 8-bit pixel values for display or storage. Benoit et al. [3] propose a model based on properties of the human retina. HDR video content is enhanced by a non-separable spatio-temporal filter with added temporal constancy. A general model for temporal luminance adaptation was proposed by Krawczyk et al. [16]. In accordance with the human visual system that reacts to temporal changes in luminance conditions, a time constant for the speed of the adaptation is introduced. Guthier et al. have developed a tone mapping technique for videos which removes flicker in a post processing step and is applicable to all tone mapping operators [9].

To the best of our knowledge, there is no low cost system available that allows the creation of HDR video in real-time.

3. Considerations for a GPU implementation

To allow the creation of high dynamic range video in real-time, time-critical parts of the HDR pipeline are processed on a graphics processing unit. As opposed to a CPU with a small number of high-performance processor cores, optimized for sequential programs, a GPU has many simple cores that complete a large number of simple tasks in parallel. Implementing algorithms for a GPU requires a higher degree of understanding of the underlying platform than a serial CPU implementation. This article thus begins by giving an introduction to Nvidia's *Compute Unified Device Architecture* (CUDA) which is used in the described work. Details can be found in Nvidia's CUDA C Programming Guide[1] or in [22], which discusses processor and memory organization and generally applicable optimization strategies.

When designing a parallel algorithm for a GPU implementation, the specific properties of the hardware must be understood. CUDA classifies GPUs into categories with similar compute capabilities, allowing independence from hardware details. Between the GPU classes, the differences are often just a matter of parameter adjustment, and all code is backward compatible so far. This section introduces the architecture common to all CUDA devices, giving numbers that are specific to the graphics card used where applicable.

The CUDA programming model is a C99 dialect with a minimum set of language extensions. At its core there are three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. This model requires partitioning of the problem into many small sub problems that can be solved independently or by cooperation of the threads within a block. More precisely, for a problem to be shifted to the GPU, it must be expressible as a data-parallel algorithm. Data parallelism describes a programming paradigm that suggests the subdivision of a problem into smaller sub problems such that the same program (kernel) can be executed by a large number of threads working on many data elements in parallel.

[1] http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation

The sum of all threads launched by a GPU within one kernel call is called a grid. A grid consists of several blocks of threads in a two dimensional structure. The elements contained in the blocks are lightweight threads executed by the GPU. In image processing, there is often a one-to-one correspondence between image blocks (e.g., 32 × 32 pixels) and blocks of threads. This means that the image is divided into pixel blocks which are then each processed by one thread block.

The conceptual structure of grids, blocks, and threads is called thread hierarchy. Thread blocks are required to be executable independently, in any sequential order or in parallel. This requirement allows threads to be scheduled in arbitrary order to a flexible number of cores, as one block is always executed by one core. The graphics card used for this work contains 15 multiprocessors of which each can process 32 threads at once.

The data to be processed (e.g., the images) must be copied from the host computer's memory to the memory of the graphics card. The GPU distinguishes between global, local, and shared memory. They differ in visibility, size, and access time. *Global memory* is accessible by all threads running on all multiprocessors. It is typically several gigabytes in size, but accessing it can take up to 800 clock cycles. If data is read only and the access pattern exhibits locality, this is sped up by level-1 and level-2 caches. The access to *local memory* is restricted to a single thread and is very fast. It is comparable to the access to CPU registers. *Shared memory* is a user-managed cache that is shared among the threads of one block and invisible to all other blocks. Its size is 48 kilobytes on the GPU in this work, and it can be read or written without latency, similar to a low-level cache or registers. Its main purposes are fast temporary data storage and communication between the threads of a block. For the sake of performance, it should be avoided to use the slow global memory wherever possible by keeping intermediate results in local or shared memory.

Special care must be taken to prevent race conditions when multiple threads write to the same address of the shared memory concurrently. The entire memory range is split up into 32 interleaved memory banks such that successive 32-bit words are assigned to successive banks. This means that the 32 threads of a block that are processed concurrently can all access the shared memory in parallel as long as the requested words lie in 32 different memory banks. When two concurrent threads access different words in the same memory bank at the same time, they can only be read sequentially, and the performance gain of parallel processing is lost. This must be considered when designing algorithms for CUDA.

There exist two additional read-only caches called constant memory and texture memory. *Constant memory* is used for broadcasting read-only values quickly to requesting threads. *Texture memory* is interesting in the given scenario as it is an optimized cache for 2D access. When a thread accesses the texture cache, the hardware prefetches values from global memory that are close to the fetched value in 2D (e.g., neighboring pixels). This decreases cache misses in image processing applications, leading to a large performance gain. Additionally, the texture cache offers addressing modes like linear interpolation of values in hardware. Consequently, these operations are very fast: fetching a linear interpolated value does not take any longer than fetching a non-interpolated one.

Multiprocessors schedule and execute threads in groups of 32 parallel threads called *warps*. Warps each have their own instruction counter and registers. They always execute one common instruction at a time. If threads diverge due to data-dependent branching, the warp serializes which means that threads following the branch are executed together while all other threads are idle. When all threads are on the same path again, execution is merged for the whole warp. Such divergent branching thus slows down execution speed and is to be avoided in the code.

4. Overview over the HDR pipeline

4.1. Calculation of optimal shutters

Each frame in the HDR video is created from a sequence of differently exposed LDR images. Before capturing such an image sequence, suitable shutter speeds must be determined. The algorithm for finding an optimal shutter speed sequence is described in detail in [10]. It makes use of the existing radiance histogram which is a by-product of tone mapping the previous HDR frame. Shutter speeds are chosen in a way such that radiance values that occur frequently in the scene are well-exposed in at least one of the captured images.

The "well-exposedness" of a certain range of radiance values is expressed by a so-called *contribution function*. It is derived from pixel weighting functions that are often found in the literature as parts of HDR creation techniques [7]. They judge a pixel's usefulness for estimating an accurate radiance value from it and are generally used during the stitching of images into an HDR frame. In the context of shutter speed calculation, weighting functions are used to construct a *combined contribution function* for any given sequence of shutter values. It indicates how well each radiance value can be reconstructed from an exposure sequence captured using the given shutter speed sequence. In other words, the combined contribution function judges the well-exposedness of a certain scene brightness range in a sequence of differently exposed images.

The cross correlation between the radiance histogram and the combined contribution function can now be calculated, resulting in a *total coverage* value. The coverage value is high when peaks in the histogram correspond well with peaks in the combined contribution function. This is equivalent to saying that frequently occurring radiance values (peaks in the histogram) are well-exposed by a certain shutter speed sequence (peaks in the contribution function). The algorithm uses this metric to decide which shutter speeds to add to the sequence next. Once an optimal shutter speed sequence is determined, it is transmitted to the camera which then starts capturing.

It should be noted that the number of shutter speeds required to cover a given scene is not known in advance. The algorithm stops once the desired total coverage is achieved. Furthermore, the combined contribution function changes with each shutter speed that is added to the sequence. This must be kept in mind for the GPU implementation described later.

4.2. Color conversion

Digital cameras usually use a Bayer color filter array to capture color images. Each sensor pixel then only records a specific range of the color spectrum – either red, green, or blue. In order to obtain an RGB value for each pixel, the two missing components must be interpolated from the neighboring pixels. See Figure 4 for an exemplary arrangement of red, green, and blue pixels. In the example, a red pixel interpolates its blue component from its diagonal neighbors. Depending on the position in the array, four cases of interpolation exist: red, blue, and two cases of green pixels.

Most processes in the HDR pipeline only operate on the brightness of an image and leave color unchanged. It is thus desirable to separate the brightness from the color information. This is done by converting the image from RGB into the Yxy color space. A pixel is then represented by the brightness component Y and two color components x and y. The first step of the conversion is a matrix multiplication that converts the RGB vector to XYZ. Conversion matrices for this step can be found in the literature [14]. The Y component is then used directly while the color components x and y are derived from the XYZ vector in an operation similar to normalization.

At the end of the HDR pipeline, the created HDR frame must be converted back to RGB for display. This is done analogously.

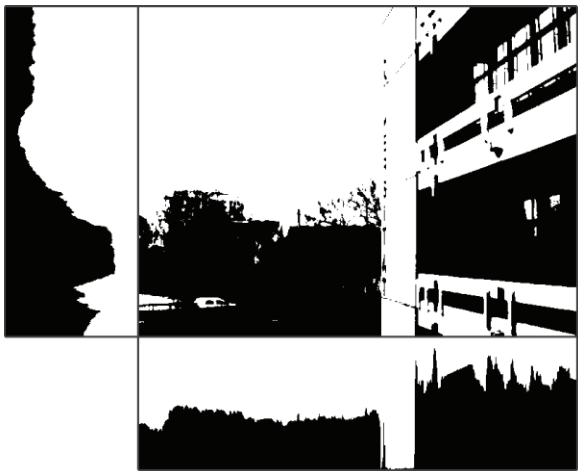
Fig. 2. A mean threshold bitmap. The row and column histograms to the left and below respectively count the number of black pixels in the corresponding line.

4.3. Histogram-based image registration

The set of low dynamic range (LDR) exposures was captured with camera motion in between. Before merging the images into an HDR frame, the horizontal and vertical shift between each pair of exposures must be estimated and compensated. Details can be found in [8] and [11].

First, a so-called *Mean Threshold Bitmap* (MTB) is calculated for each exposure [25]. It is a black and white version of the original image with a threshold chosen such that 50% of the pixels are black and 50% white. The threshold is set by first creating a brightness histogram and finding its median. The advantage of MTBs is that they are – to a certain degree – invariant to exposure change. This is a desirable property for the registration of exposure sequences.

Once an MTB is created, its pixels are summed up horizontally and vertically to establish row and column histograms. It is necessary to calculate separate histograms counting black and white pixels, because pixels near the threshold are ignored. This leads to a total of four histograms per exposure and eight histograms for the registration of an exposure pair. See Figure 2 for an example.

Next, the *Normalized Cross Correlation* (NCC) between corresponding histograms of the two exposures is calculated to estimate the intermediate shift. In the example of horizontal shifts, the NCC between the column histograms of both images is calculated for each possible shift value within a predefined search range. The shift value leading to the best correlation value is assumed to be the correct one.

As a last step, all resulting shift vectors are validated using a *Kalman filter* to incorporate knowledge of the motion in previous frames into the estimation. Based on a certainty criterion, the shift vector is used directly or interpolated from values obtained in preceding frames.

4.4. HDR stitching

The registered image sequence is merged into an HDR frame in a process called HDR stitching. A detailed explanation of it can be found in [7].

During image capture, the radiance emitted from a point in the scene is measured and recorded as a device-specific pixel value. The goal of HDR imaging is to recover the physical radiance again that gave rise to a pixel value. A pixel in an HDR frame represents the radiance at one point in the scene. HDR stitching is thus the inverse of the capturing process: Estimating radiance from pixel values. This is done by applying the inverse camera response function to all pixels of all LDR exposures and dividing them by their respective shutter speed. Like this, one approximation of the radiance map is obtained from each of the exposures. A weighted average over the estimated radiance maps then yields the real radiance. The weighting function used here is identical to the one used to determine optimal shutter values.

4.5. Tone mapping with flicker reduction

In order to be displayable on a regular screen, the large radiance ranges of an HDR frame need to be compressed to 8-bit values. Preferably, the compression is done in a way that maintains as much of the gained HDR information as possible. This process is called *tone mapping*. The tone mapper used in this work is described in [24]. It is augmented by flicker reduction as detailed in [9].

The tone mapper in use is a global operator that applies the same tone reproduction function to all pixels in the HDR frame. This function is derived from the cumulative histogram over log radiance values in the scene. After normalization and clipping of the histogram bins, the cumulative histogram is used directly as the mapping function. This is similar to histogram equalization.

For tone mapping, the following steps are necessary. The highest and lowest radiance values in the HDR frame must be determined first to set the range of histogram bins. A log radiance histogram can then be calculated. It is used later to determine the shutter speeds for the next frame. Summing up the bins results in a cumulative histogram. The tone mapping function derived from it is then applied to each pixel in the HDR frame.

The described operator was designed with still images in mind. It is used on each frame of the HDR video individually. When doing so, temporal changes of the minimum or maximum scene radiance lead to rapid changes of the mapping function from one frame to the next. This shows up as flicker in the tone mapped video. Flicker is thus detected and removed in a post-processing step. The average image brightness of each tone mapped frame is calculated. Large variations of the average over a short amount of time indicate flicker. When flicker is detected, the average brightness is adjusted to remove the flicker effect. Adjusting the image brightness is done by multiplying the pixels by a certain factor.

5. Parallel Implementation of the subtasks

Redesigning an algorithm for a parallel implementation takes considerable effort. It is also more difficult to assure correctness and to maintain such an implementation. The individual steps of the HDR pipeline are thus first analyzed with respect to the computation time they require and their suitability for parallelization. The former is measured easily from the existing sequential code. The latter is judged by the amount of parallelism a problem exhibits and its arithmetic intensity: Parallelism is the percentage of instructions that can be executed concurrently; *arithmetic intensity* can be defined as the ratio between mathematical operations and memory access, where a higher arithmetic intensity is preferable for a GPU realization. Both criteria are somewhat vague but still sufficient for assessing the suitability for a

Table 1

Overview of the subtasks of the HDR pipeline. Shown are the relative computational cost, the amount of parallelism (P), and the arithmetic intensity (AI). "high" entries indicate factors that suggest a GPU implementation. Our decision for the type of implementation is given in the rightmost column.

Pipeline Step	Operators	Cost	P	AI	GPU/ CPU
Optimal Shutter Seq.	Cross Correl.	low	med.	high	CPU
Bayer Pattern Interpolation	-	high	high	med.	GPU
Color Space Conversion	-	high	high	high	GPU
Image Registration	Brightness Hist.	high	med.	low	GPU
	Median	low	med.	low	CPU
	Row/Col. Hist.	high	high	low	GPU
	NCC	low	high	high	GPU
	Kalman Filter	low	low	high	CPU
HDR Stitching	Weighted Avg.	high	high	high	GPU
Tone Mapping	Min / Max	high	med.	low	GPU
	Brightness Hist.	high	med.	low	GPU
	Hist. Cumul.	low	med.	low	CPU
	TM Operator	high	high	high	GPU
	Avg. Brightness	high	med.	low	GPU
	Normalization	high	high	med.	GPU
Color Back Conversion	-	high	high	high	GPU

parallel implementation. For a more detailed discussion of arithmetic throughput and global memory latency see [22].

The complete HDR video system described here consists of the following parts: Calculation of optimal shutters, capturing images, color conversion, histogram-based registration, HDR stitching, histogram adjustment tone mapping with flicker reduction, and color back conversion. These parts can be further divided into their computationally expensive subtasks.

The most expensive step of *determining shutter sequences* is repeatedly calculating the cross correlation between the (existing) brightness histogram and the contribution vector.

Capturing is done by the camera and cannot be sped up. The Bayer pattern in the captured LDR images is first interpolated to full RGB and then converted into Yxy.

For *registration*, a brightness histogram must be calculated for each LDR image and its median must be found. Row and column histograms are then created from a temporary MTB and the normalized cross correlation (NCC) between them is calculated repeatedly. The resulting shift vector is Kalman filtered.

HDR *stitching* consists of computing each HDR pixel from a weighted average over the corresponding LDR pixels.

Tone mapping requires the computation of a cumulative log radiance histogram, which consists of finding the minimum and maximum radiance, calculating a log radiance histogram and cumulating the bins. Each pixel is then tone mapped from radiance to pixel values. To reduce flicker, the average brightness of the tone mapped result must be calculated, and the image must be normalized iteratively.

In the end, the tone mapped image is *converted* back to RGB.

In the following, all subtasks are analyzed with respect to necessity and feasibility of a parallel implementation. The results of the analysis are summarized in Table 1. Refer to Section 4 for details. For those subtasks that are chosen for a parallel implementation, the modifications to the algorithms that are necessary to run them on a GPU are described as well. Subtasks that are similar to each other are discussed only once.

5.1. Normalized cross correlation

The normalized cross correlation between all row or column histograms for all possible shift values can be calculated independently of each other. A high cache hit rate is expected, because the same row/column histogram bins are read repeatedly and never changed in between. This allows them to be bound to the texture cache. Additionally, a high arithmetic intensity makes cross correlation well-suited for parallelization. On the other hand, it is a rather cheap operation overall. Cross correlation was implemented on the GPU only for image registration and not for shutter calculation: The row and column histograms were created on the GPU and thus already reside in the graphic card's main memory. Furthermore, they are constant during the entire cross correlation, enabling efficient caching and data independence. This is not the case when determining optimal shutters. The combined contribution, which is repeat-

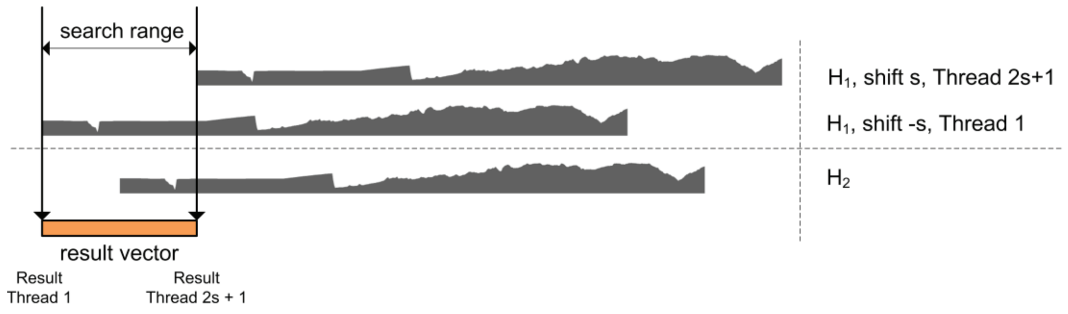

Fig. 3. Normalized cross correlation between two column histograms H_1 and H_2 to determine the horizontal shift s between two images. A thread is started for each value in the result vector to be calculated. The values represent the correlation for a specific shift.

edly correlated with the brightness histogram, changes after each determined shutter speed. This adds a sequential dependence to the calculation, making it less suitable for parallelization. Since it is a cheap operation, the existing sequential code was kept.

To compute the NCC between two row or column histograms, one thread is started for each shift *s* in the search range. Each thread calculates the normalized cross correlation for its shift and writes the result of the calculation to its corresponding position in the result vector. In the end, the result vector is downloaded into the host memory. A sequential search on the CPU finds the position of the highest correlation value. Figure 3 illustrates the process.

5.2. Bayer pattern interpolation

For bilinearly interpolating a pixel's RGB value from its neighbors, four cases need to be differentiated based on the pixel's location on the color filter array. This means that without further modification, this method leads to massive branching in the kernel. On the other hand, an interpolation kernel can benefit from texture caching, because the access pattern is highly local in 2D. It is also a highly parallel problem. Its arithmetic intensity varies with the pixel position with an average of 0.65 arithmetic operations per memory access.

A naive implementation iterates through the image and interpolates the missing pixel values differently depending on the four possible locations in the Bayer grid. In order to avoid branching, a thread relocation mechanism was implemented which is illustrated in Figure 4. It changes the relationship between a pixel and the thread which does the interpolation. Normally, a thread would be responsible for interpolating the RGB values for a pixel matching the thread's position in the 2D grid. Neighboring threads would then be executed at the same time sharing the same instruction counter. In this situation, the different branching of the threads would lead to a serial execution and low performance. Relocating the threads so that those corresponding to pixels with matching location on the color filter array are executed simultaneously avoids branching. For example, in the Figure, every thread in block 4 can now calculate its blue component from its left and right neighbors.

5.3. Color conversion, tone mapping, normalization

Color space conversion, as well as applying the tone mapping operator and image normalization, are ideally suited for a GPU implementation since they fully comply with the data parallelism paradigm [5]. No branching takes place as each element is treated in the same way. Each image pixel is read and written exactly once. Additionally, all three operations have a high arithmetic intensity caused by the multiplication and addition of pixel values.

We limit our description here to the parallel implementation of color conversion. The same also applies to using a global tone mapping operator and to image normalization. The implementation of the kernel for color conversion from RGB to XYZ is the translation of a color conversion matrix into code. Again, one thread is started per pixel. The RGB values for the conversion are read from the pixel corresponding to the thread. These values are multiplied by the color transformation matrix. The resulting XYZ vector is then normalized to obtain the chromaticity *xy* and the brightness *Y*. These values are written back to the three channels of the pixel in the output image. Afterwards, the result can either be passed on to the next kernel (e.g., image registration), or it can be displayed in the case of the final back conversion to RGB.

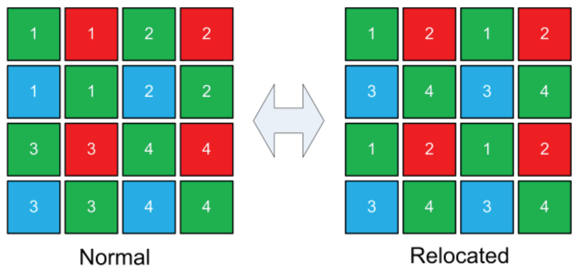

Fig. 4. Each pixel is assigned one thread to interpolate its RGB values from the surrounding. The threads with the same number belong to the same block and are executed at the same time. This leads to threads with different colors in their neighborhood running simultaneously, and branching becomes necessary. After relocating the threads, each thread in the block can interpolate in the same way.

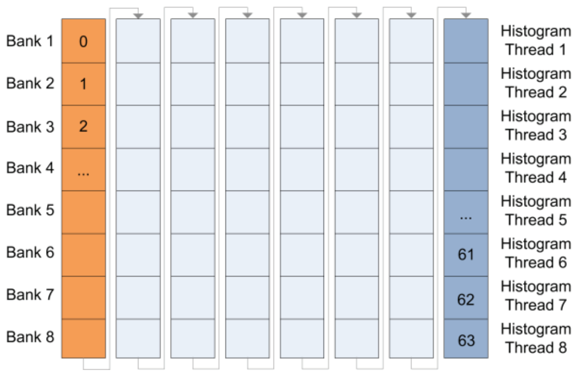

Fig. 5. Simplified illustration of shared memory with *8* memory banks and *64* addresses. Consecutive addresses lie on consecutive banks. The histograms are interleaved such that each lies on its own bank. Eight threads can write concurrently.

5.4. Brightness histogram

During the creation of a brightness histogram, data must be written to a small set of memory addresses (the histogram bins) for all of the pixels. This induces a certain data dependence and leads to thread collisions and sequential writing. Additionally, there is very little computational work between the memory accesses. Despite these difficulties, it was considered for a GPU implementation because it is a costly operation overall. Our implementation is similar to the one proposed in [5].

The image over which the histogram is computed is first subdivided into rectangular areas of size *32* × *64* pixels. To calculate a histogram, each block has to perform *2048* read and write operations on the histogram bins in global memory. This would take many clock cycles, and the operations would be strictly serial. Instead, the paradigm of parallel reduction is applied. That is, histograms for small image areas are created first and then successively merged into one final histogram over the entire image. *32* threads are started per block. Each thread computes a separate histogram with *64* bins over one row of the block which is written to the fast shared memory. The histogram has only *64* bins for efficiency purposes. Interleaving of the histograms in shared memory such that a whole histogram resides on the same memory bank allows for conflict-free memory access by the threads, and true parallelism can be achieved. The banks of shared memory and the way the histograms are stored is illustrated in Figure 5.

Next, the *32* histograms of the block are summed up into a single histogram for the block. Since the content of the shared memory expires when the threads terminate, the same *32* threads must be reused for summation. Each thread is assigned two bins of the total histogram. It loops through the *32* histograms (vertically in Figure 5) and maintains two sums. It must be noted that each histogram resides on its own memory bank. If all threads started with the first histogram, the 32 read operations to the same bank would be serialized, leading to bad performance. Instead, summation loops start with a different histogram for each thread (shifted by one relative to its predecessor thread). Like this, all summations can be done in parallel without bank conflicts. The final sums (i.e., the histogram for the block) are then written to global memory by an atomic add function provided by CUDA.

With this approach, the number of write operations to global memory is reduced from *2048* to *64* per block. Pixel data is read from global memory as a texture which allows for efficient caching.

5.5. Median computation

There exist parallel sorting algorithms, so the problem of finding the median of a histogram can be parallelized. However, the problem size of searching through a number of histogram bins is too small to justify the effort.

5.6. Row and column histograms

The creation of a mean threshold bitmap can be viewed as an intermediate step to the computation of row and column histograms. Both operations have a low arithmetic intensity. The creation of a threshold bitmap has high parallelism as each pixel can be converted separately. Computation of row and column histograms brings up a similar issue as brightness histogram computation: An entire row/column accesses the same histogram bin. However, in this case, this access is predictable and can be optimized.

For the registration of an image pair, a total of eight row or column histograms are created. All histograms are calculated separately by similarly implemented method calls. On a GPU, this is more efficient than running one parameterized method with code branching. For simplicity, only the creation of a column histogram counting black pixels for every column of an image is described here.

The image is subdivided into blocks of *32 × 32* pixels. This time, one thread is started for each pixel in the block. Each thread computes the thresholded value of its respective pixel, that is, the thread checks if its pixel is darker than the threshold and writes a *1* into shared memory if it is. The mean threshold bitmap is thus created in shared memory only. Care is taken that the set of *32* threads that are executed concurrently on a multiprocessor write the bit to 32 separate memory banks. Pixel data is again read from global memory as a texture. *32* of the threads are then re-used to count the black pixels of each column. Each thread is assigned one of the columns of the block. The thread loops through all rows to count the *1*s in shared memory. An entire row resides in the same memory bank (see Figure 5). So, in order to prevent bank conflicts, the threads each start counting from a different row so that all *32* read operations can be done in parallel. The sum of black pixels in a column is then added to the column histogram in global memory using an atomic add operation.

5.7. Kalman filtering

Filtering only takes about 16 µs on a CPU, so its computational effort is negligible and thus left on the CPU.

5.8. HDR stitching

HDR stitching complies well with the data parallelism paradigm. The radiance value of each HDR pixel can be obtained independently without a need for synchronization. The arithmetic intensity of this operator is high due to the addition and multiplication of pixel values and the evaluation of the weighting function.

HDR stitching cannot use the GPU's texture cache to access the LDR exposures to be stitched. This is because the CUDA compiler needs to know the number of texture cache bindings in advance before compilation; however, the number of LDR exposures is dynamically chosen. Depending on the parameters of the shutter speed selection algorithm [10], a large and unbounded number of images may need to be captured. Furthermore, due to the way we capture exposures [12], their size may vary in each frame.

The LDR sequence can be viewed as a 3D stack of images with varying size. This stack of images resides in the global memory of the GPU. One thread is started for each HDR pixel which iterates through the corresponding pixels in the stack of exposures and calculates the weighted average. During one iteration, the current radiance and chrominance values and the cumulated weight of all pixels must be held in local memory (registers). The radiance and the two chrominance values are then written to the output image in global memory as floating point values. Contrary to the CPU version, the GPU implementation does not precalculate the weighting function or stores it in memory. Experiments showed that calculating only the required weights on the fly is cheaper than accessing the precalculated array from all threads.

5.9. Minimum, maximum, and average

Calculating a cumulative log radiance histogram for tone mapping starts by finding the minimum and maximum log radiance in the HDR image. This is similar to calculating the average image brightness for flicker reduction. By employing so-called parallel reduction, these processing steps can be made parallelizable to some extent. The arithmetic intensity is low, because very little computational work is done between the memory accesses. We implemented it for the GPU, because it is a costly operation overall.

To illustrate parallel reduction, we describe the calculation of the maximum of a one-dimensional array here. This technique is applied to the calculation of the minimum, maximum and average of the pixels in an image.

Parallel reduction is an iterative divide-and-conquer approach. In the first iteration, one thread is started for each element in the array. Each thread calculates the (trivial) maximum of the element, which is simply the element itself, and writes it to shared memory. Every other thread is then discarded. In the next iteration, the remaining threads calculate the maximum of their element and its neighbor and again write it to shared memory. In each iteration, the number of threads is halved, and maxima are combined with their neighbors. This is repeated until only one global maximum is left which can then either be written to global memory or copied back to the host system.

5.10. Cumulative log radiance histogram

The same considerations as for general brightness histograms described above apply to the computation of the cumulative log radiance histogram. The only exception is the summing up of the histogram bins. It has negligible computational costs and does not require a GPU implementation.

6. Experimental results

The following performance aspects of the HDR video system were assessed:

- processing times of the subtasks of the HDR pipeline when changing the image size,
- processing times when changing the number of exposures,
- average capturing and processing times in a *30* seconds HDR video under realistic conditions,
- comparison of CPU and GPU processing times.

The individual subtasks were grouped together as seen fit for the analysis. Displaying the processed video frames means copying them into the memory of the graphics card. Since the last step in the pipeline – color conversion from Yxy back to RGB – is performed on the GPU anyway, displaying the result is a free operation and thus ignored in the following.

For all experiments, a desktop PC was used which is equipped with an AMD Athlon II X2 250 64-bit CPU with two cores running at 3 GHz and a total of 4 GB of RAM. The installed graphics card is an Nvidia GeForce GTX 480 with 15 multicores running at a clock rate of 1.4 GHz and 1.5 GB of dedicated memory. Each multicore can process *32* threads at once. The used camera is an AVT Pike F-032C FireWire camera capable of capturing *208* frames per second in VGA resolution. It uses a Bayer color filter array to acquire color images.

The computation time of most of the steps in the HDR pipeline depends on the size of the images. In the experiment described here, the relationship between processing time and image size is analyzed. Most parts of the GPU implementation are optimized specifically for the image width of *640* pixels. Changing this in the implementation would bias the results of this test. However, different image heights were considered in the implementation to accommodate the outputs of the employed capturing method. This fact is used to investigate the relationship between image size and processing time. The images in this experiment all had the full width of *640* pixels and a height varying from *100* to *480* pixels.

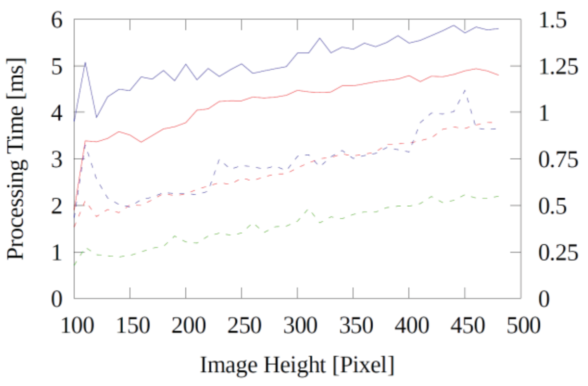

Fig. 6. Processing time versus image height for a fixed number of *5* exposures. The two solid lines are image registration (blue) and color conversion (red). They both use the left scale ranging from *0* to *6* ms. The dashed lines use the right scale. They are tone mapping (blue), HDR stitching (red) and color back conversion (green).

For each size, a sequence of five exposures was captured once and processed *20* times by the entire HDR pipeline to obtain stable average processing times. The steps from Bayer pattern interpolation to the computation of row and column histograms were thus performed five times in each iteration, cross correlation and filtering was done four times, and HDR stitching needs to take five exposures into account. All the subsequent steps work on just one HDR image.

The content of the images has no significant influence on the processing times. The shutter speeds were thus set to an arbitrary value that exposes the recorded indoor scene well. Most steps of the pipeline depend on the image size in an obvious way as it influences the number of pixels or blocks to process. Only the processing steps of shutter speed computation and Kalman filtering are completely unaffected. Figure 6 shows the measured processing times versus image height. Debayering and the initial color conversion are grouped together. Excluding apparent measurement noise, the computation times of the pipeline steps grow linearly in the number of pixels, as expected.

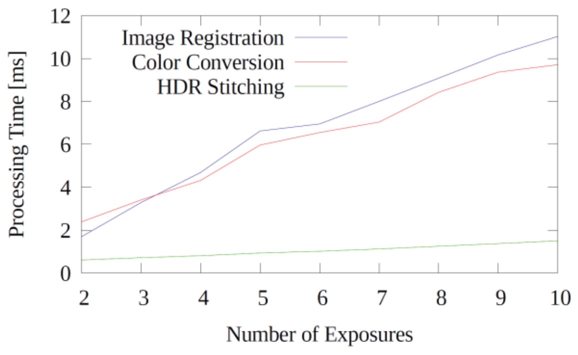

Fig. 7. Processing time versus number of exposures for full images. Only those steps that process multiple LDR exposures are considered.

Now, the image size is kept at its maximum of *640* × *480* pixels and the number of LDR exposures is varied instead. The only steps that need to process a varying number of exposures are color conversion from RGB to Yxy, Bayer pattern interpolation, image registration, and HDR stitching. The initial color conversion and debayering are again grouped together. In order to perform registration, at least two images must be present in the sequence. The number of exposures thus varies from two to ten and, again, the measurement was repeated *20* times on the same sequence for a stable average. The shutters were chosen to match the given scene. Figure 7 shows the measured processing times versus the number of exposures. Again, the dependency is linear, as expected.

For the final test, *30* seconds of HDR video material was captured in a realistic scenario using the HDR video system. The camera was situated inside a room illuminated only by sunlight shining through a window on a sunny day. During the *30* seconds, the camera pans from the bright window towards the darker room and eventually towards a door leading to an even darker hallway. The video thus includes very bright, very dark, and mixed lighting conditions. It consists of *733* HDR frames. Averaged over the entire video, an HDR frame was created from *3.62* LDR exposures. On the average, *29.8* ms per frame were spent for capturing and *13.6* ms for processing. This results in a total time per frame of *43.4* ms and an average frame rate of *23* fps.

The time to create an HDR frame from beginning to end is more closely inspected in Figure 8. The top left chart shows the fractions of the computation time of the steps of the HDR pipeline. Color conversion from RGB to Yxy, back to RGB and debayering is grouped together. The other sub-figures show how

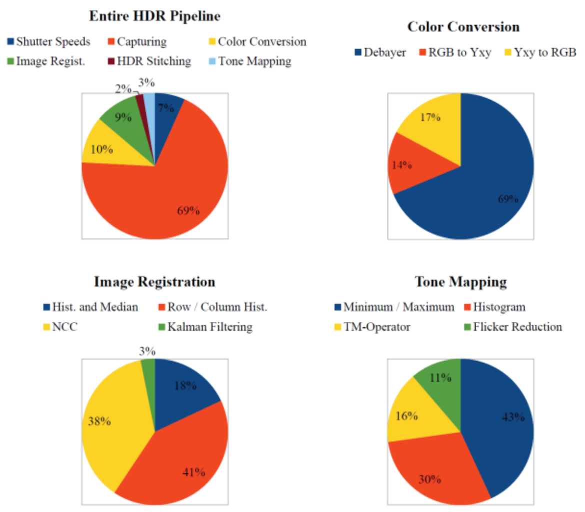

Fig. 8. Percentage of the time taken to perform the steps of the HDR pipeline in the test video. The steps of color conversion, image registration, and tone mapping are further subdivided into their individual tasks.

computation time is further divided among the subtasks for color conversion, image registration, and tone mapping.

For comparison, the entire video was processed again using a fully optimized CPU implementation. The computation times were now *56* ms for color conversion, *24* ms for image registration, *43* ms for HDR stitching and *80* ms for tone mapping. This leads to an average processing time of *203* ms per frame. The GPU implementation is thus faster by a factor of *15* on the average.

7. Conclusions

This article presented the GPU implementation of a high dynamic range video system. It gave an overview of the system and outlined the major steps from capturing to display of an HDR frame. Before parallelizing the HDR pipeline, it was analyzed for necessity and feasibility of a parallel implementation. The main decision criteria were computational cost, arithmetic intensity and the amount of data parallelism of the subtasks. The HDR algorithms had to undergo modifications to make them suitable for execution on a GPU.

The performance of the GPU implementation was evaluated using 30 seconds of HDR video captured in real-time under realistic conditions. The system achieved a frame rate of 23 frames per second. It is thus fast enough for real-time HDR video. Compared to the CPU implementation, a speedup by a factor of 15 was achieved.

The experiments showed that after parallelization, the bottleneck now lies in capturing of the exposure sequence. 69% of the time taken to create an HDR frame was spent for capturing. In order to increase the frame rate further, it is sensible to focus on image capturing next. Possible improvements could be using a bigger lens to allow for shorter shutter speeds, stronger decoupling of capturing and processing or employing different capturing hardware.

References

[1] M. Amamiyaa, H. Tomiyasua, R. Taniguchia, P. Kacsukb and Z. Nemethb, Multithreaded architecture for multimedia processing, Integrated Computer-Aided Engineering 7(1), 2000.

[2] F. Amiot and E. Pissaloux, Towards vision application adaptable parallel computer, Integrated Computer-Aided Engineering 8(4), 2001.

[3] A. Benoit, D. Alleysson, J. Herault, P. Callet, Spatio-temporal Tone Mapping Operator Based on a Retina Model, Springer, 2009.

[4] T. B. Borchartt, et al., on the reproduction of cloud influence in natural and aerial images, IWSSIP, 2011.

[5] G. J. van den Braak, C. Nugteren, B. Mesman, H.Corporaal, GPU-Vode: a framework for accelerating voting algorithms on GPU, Euro-Par Parallel Processing, 2012.

[6] L. Carro-Calvo, S. Salcedo-Sanz, E. G. Ortiz-García, A. Portilla-Figueras, An incremental-encoding evolutionary algorithm for color reduction in images, Integrated Computer-Aided Engineering 17(3), 2010.

[7] P.E. Debevec and J. Malik, Recovering high dynamic range radiance maps from photographs, in Proc. of the 24th Conference on Computer Graphics and Interactive Techniques, 1997.

[8] B. Guthier, S. Kopf, and W. Effelsberg, Histogram-based image registration for real-time high dynamic range videos, in Proc. of IEEE International Conference on Image Processing (ICIP2010), 2010.

[9] B. Guthier, S. Kopf, M. Eble, and W. Effelsberg. Flicker reduction in tone mapped high dynamic range video. In Proc. of IS&T/SPIE Electronic Imaging (EI) on Color Imaging XVI: Displaying, Processing, Hardcopy, and Applications, volume 7866, 2011.

[10] B. Guthier, S. Kopf, W. Effelsberg, Optimal Shutter Speed Sequences for Real-Time HDR Video, Int. Conf. on Imaging Systems and Techniques (IST), 2012.

[11] B. Guthier, S. Kopf, W. Effelsberg, Parallel Algorithms for Histogram-based Image Registration, IWSSIP, 2012.

[12] B. Guthier, S. Kopf and W. Effelsberg, Algorithms for a real-time HDR video system, Pattern Recognition Letters, 2012.

[13] S. Hasinoff, F. Durand, W. Freeman, Noise-Optimal Capture for High Dynamic Range Photography, in: Proc. of the 23rd IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2010.

[14] International Telecommunication Union (ITU), Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange, ITU-R Recommendation BT.709, 1990.

[15] S. B. Kang, M. Uyttendaele, S. Winder, R. Szeliski, High dynamic range video, ACM Transactions on Graphics (TOG) 22 (3), 2003.

[16] G. Krawczyk, K. Myszkowski, D. Brosch, HDR Tone Mapping, Springer Series in Advanced Microelectronics (26), Springer, 2007.

[17] L. Lattari, A. Montenegro, A. Conci, E. Clua, V. Mota, M. Bernardes Vieira and G. Lizarraga, Using graph cuts in GPUs for color based human skin segmentation, Integrated Computer-Aided Engineering 18(1), 2011.

[18] M. McGuire, Efficient, high-quality bayer demosaic filtering on GPUs, Journal of Graphics, GPU and Game Tools, vol. 13 (4), 2008.

[19] A. Oppenheim, R. Schafer, and T. Stockham, Nonlinear filtering of multiplied and convolved signals, Proc. IEEE vol. 56, pp. 1264-1291, 1968.

[20] R. del Riego, J. Otero and J. Ranilla, A low-cost 3D human interface device using GPU-based optical flow algorithms, Integrated Computer-Aided Engineering 18(4), 2011.

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

[22] S. Ryoo et al., Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, Prof. of the PPoPP, pp. 73-82, 2008.

[23] M. D. Tocci, C. Kiser, N. Tocci, and P. Sen, A versatile HDR video production system. ACM Trans. Graph. 30 (4), Article 41, 2011.

[24] G. Ward, H. Rushmeier, and C. Piatko, A visibility matching tone reproduction operator for high dynamic range scenes. IEEE Transactions on Visualization and Computer Graphics, 3(4), 1997.

[25] G. Ward, Fast, robust image registration for compositing high dynamic range photographs from hand-held exposures, Journal of Graphics Tools: JGT, vol. 8, no. 2, 2003.