PARALLEL ALGORITHMS FOR HISTOGRAM-BASED IMAGE REGISTRATION

Benjamin Guthier, Stephan Kopf, Matthias Wichtlhuber, Wolfgang Effelsberg

{guthier, kopf, effelsberg}@informatik.uni-mannheim.de
mwichtlh@pi4.informatik.uni-mannheim.de
University of Mannheim, Germany

ABSTRACT

Image registration is the process of finding pixel correspondences between images that were taken at different points in time or from different viewpoints. In particular, we are interested in registering exposures captured with varying shutter speed settings for the creation of high dynamic range (HDR) images. For an existing histogram-based image registration technique, we discuss the suitability for parallelization. Modifications to the technique that allow us to run parts of the algorithm in parallel are presented. The parallel parts can then be processed by a graphics processing unit (GPU) for improved performance. Our results show that the most time consuming component of the algorithm can be sped up on an Nvidia GPU by a factor of up to 19 over the original sequential version. The average speed-up is 5.9:1.

Index Terms— Image Registration, HDR Video, GPU

1. INTRODUCTION

In order to capture the entire *dynamic range* of a scene with a regular camera, one exposure may often be insufficient. A scene that consists of a dark indoor area and a window to the bright outside may be partly under- or overexposed, depending on the chosen shutter speed and aperture setting. *High Dynamic Range Imaging* (HDR Imaging) provides techniques to tackle this problem. One of them is temporal exposure bracketing, where multiple exposures of the same scene are captured using varying shutter speeds. These exposures then have to be registered with respect to each other to compensate intermediate camera and scene motion so that pixel correspondences can be established. We argue that a purely translational camera motion model is sufficiently accurate when the exposures are captured with high frame rates (e.g., 208 frames per second for an HDR video). This assumption is supported by [1]. The fully registered exposure set is then merged into a single HDR frame. The last step is tone mapping of the HDR frame to compress its dynamic range to be displayable on a regular computer display. References for merging and tone mapping can be found in [2].

In our work, we aim for the creation of HDR video in real-time. This means that all the steps described here need to be performed within the duration of a single video frame. The first step in achieving this is employing fast algorithms for capturing and image registration [2, 3]. Additionally, we decrease the computation time by an efficient parallel implementation. Current *Graphics Processing Units* (GPUs) can run several hundred threads in parallel. However, the parallelization of sequential algorithms is a non-trivial task. It requires profound knowledge of the algorithm and the hardware being used. There already exist a number of GPU-based image registration techniques, particularly for medical images [4, 5]. To our knowledge, none of them focus on the challenges arising from the large brightness differences and saturation effects among the exposures from which an HDR video is created. This also applies to GPU implementations of registration techniques in libraries like OpenCV.

This paper is based on our previous work on image registration and describes the changes made for a parallel implementation [3]. It is structured as follows. Sections 1.1 and 1.2 give an introduction to our existing registration technique and to the properties of modern GPUs. Section 2 analyzes the components of our image registration method towards suitability for a parallel implementation. It also describes the parallelization of two components. The performance improvements achieved by our modified algorithms are shown in Section 3. Section 4 concludes the paper.

1.1. Histogram-based Image Registration

One HDR video frame is created from a set of low dynamic range (LDR) exposures which were captured with camera motion in between. This section gives an overview of the computation of horizontal and vertical shift between a pair of exposures. Details can be found in the original paper [3].

First, a so-called *Mean Threshold Bitmap* (MTB) is calculated for each exposure [1]. 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 circumstance makes our registration technique particularly suitable for HDR video.

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 1 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.

1.2. Considerations for a GPU Implementation

When designing a parallel algorithm for a GPU implementation, the specific properties of the hardware must be understood. We use Nvidia's *Compute Unified Device Architecture* (CUDA) to create code for Nvidia GPUs. This section introduces the architecture common to all CUDA devices, giving numbers that are specific to our graphics card where applicable. For details, see Nvidia's *CUDA C Programming Guide* [1].

The computational problem is first divided into independent blocks which share as little data as possible. They are later processed on separate multiprocessors on the GPU. In our scenario, this is done by parting the image into rectangular areas (e.g., 32 by 32 pixels). The main difficulty lies in achieving data independence between the blocks. Data is shared more easily within a block.

A block is processed by starting multiple threads on a multiprocessor. Every thread runs the same code on a different part of a block. Sets of 32 threads are processed in parallel on one multiprocessor, sharing the same instruction counter. Conditional jumps that take different paths in parallel threads can thus only be handled inefficiently.

The data to be processed (i.e., the LDR exposures) must be copied from the host computer's memory to the memory of the graphics card. The GPU distinguishes (among other types) between *global* and *shared* memory. They differ in visibility, size and access time. Global memory is accessible by all blocks and is typically several gigabytes in size, but accessing it can take up to 800 clock cycles. Shared memory is a user-managed cache that is shared only among the threads of one block. Its size is 48 kilobytes on our GPU, and it can be read or written without latency, similar to a CPU register. Its main purposes are fast temporary data storage and communication between the threads of a block. 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 shared memory in parallel as long as the requested words lie on 32 different memory banks. When two concurrent threads access different words on the same memory bank at the same time, they can only be read sequentially, and the performance gain of parallel processing is lost. This needs to be considered when designing algorithms for CUDA.

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

Operation	Cost	P	AI	Impl.
Brightness Histogram	high	med.	low	GPU
Median Computation	low	med.	low	CPU
Row / Column Histogram	high	high	low	GPU
Cross Correlation	low	high	high	GPU
Kalman Filtering	low	low	high	CPU

Table 1. Relative computational cost, amount of parallelism (P), arithmetic intensity (AI), and type of implementation of the registration steps. "high" entries indicate factors that suggest a GPU implementation.

2. PARALLEL IMAGE REGISTRATION

Redesigning an algorithm for a parallel implementation takes considerable effort. It is also more difficult to assure correctness and to maintain such an implementation. We thus first analyze the individual steps of image registration with respect to their computational cost and their suitability for parallelization. The former is measured 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. These criteria are summarized in Table 1.

Brightness histogram: During the creation of a histogram with 64 bins, data must be written to the same set of 64 memory addresses for all of the pixels. This induces a strong data dependence. Additionally, there is very little computational work between the memory accesses. Despite these facts, we considered it for a GPU implementation because it is a costly operation.

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 the bins of a histogram is too small to justify the effort.

Row and column histograms: The creation of an MTB can be viewed as an intermediate step to the computation of row and column histograms. Both operations have low arithmetic

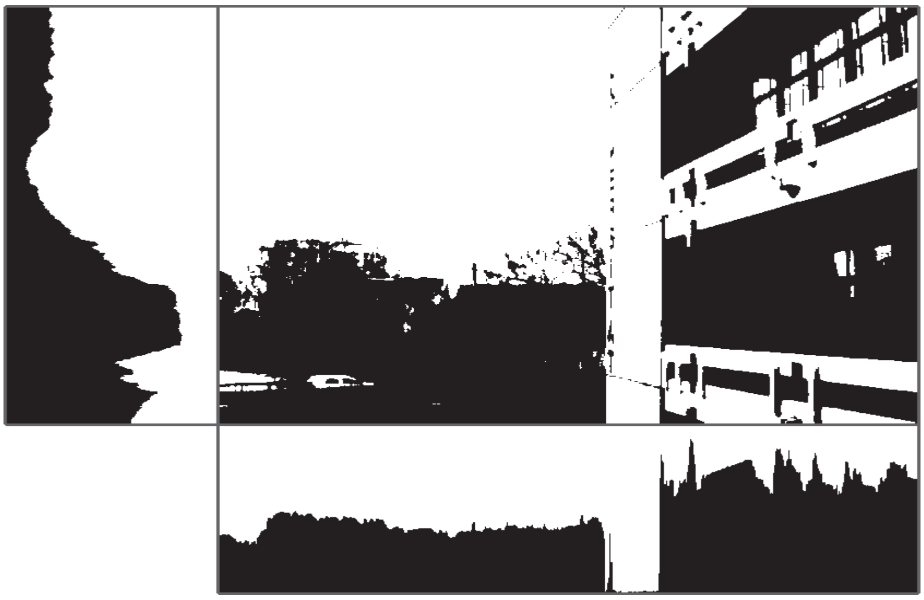

Fig. 1. A mean threshold bitmap. The row and column histograms to the left and below respectively count the number of black pixels in the corresponding line.

intensity. MTB creation 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, though: An entire row/column accesses the same histogram bin. However, in this case, this access is predictable and can be optimized. Since this is the most time consuming step of the entire image registration, it is also done by the GPU.

Normalized cross correlation: The NCCs for all possible shift values of the search range can be calculated independently of each other. A high cache hit rate is expected because the same histogram bins are *read* repeatedly. Additionally, a high arithmetic intensity makes NCC computation well-suited for parallelization. On the other hand, it is a rather cheap operation overall. We implemented it on a GPU but left it out of this paper because of space limitations.

Kalman filtering: Filtering only takes about 16μs on a CPU, so its computational effort is negligible.

2.1. Brightness Histogram Computation

The image over which the histogram is computed is first subdivided into rectangular areas of size $M \times N$ pixels (32×64 in our case). To calculate a histogram, each block has to perform $M \cdot N$ 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, we apply a paradigm called *parallel reduction*. That is, we first create histograms for small image areas and then successively merge them into one final histogram over the entire image.

M 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. 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 2.

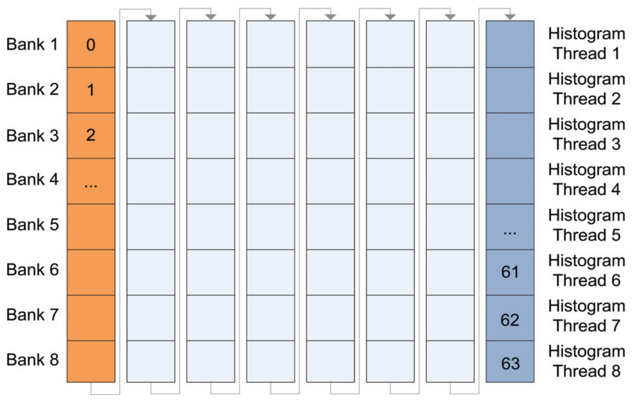

Fig. 2. Simplified illustration of shared memory with 8 banks and 64 addresses. Consecutive addresses lie on consecutive banks. The histograms are interleaved such that each lies on its own bank. Eight threads can write concurrently.

Next, the M 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 M threads must be re-used for summation. Each thread is assigned two bins of the total histogram. It loops through the M

histograms (vertically in Figure 2) 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 M 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, we reduce the number of expensive write operations to global memory from $M \cdot N$ to 64 (the number of histogram bins) per block.

2.2. Row and Column Histogram Computation

This section describes the creation of the horizontal and vertical projection profiles, called row and column histograms, on a GPU. For the registration of an image pair, a total of eight such histograms are created (see Section 1.1). All histograms are calculated separately by similarly implemented method calls. On a GPU, this is more efficient than running one parametrized method with code branching. For simplicity, we restrict our description to the creation of a column histogram counting black pixels for every column of one image.

We subdivide the image into blocks of $M \times N$ (here: 32×32) pixels. This time, one thread is started for each pixel in the block. Each thread computes the MTB of its respective pixel, that is, the thread checks if its pixel is darker than the threshold and writes a 1 into shared memory. The MTB is thus created in shared memory only. Care is taken that the set of threads that are executed concurrently on a multiprocessor write the bit to separate memory banks.

N of the threads are then re-used to count the black pixels of each column. A thread loops through all rows of its column to count the 1s in shared memory. An entire row resides on the same memory bank (see Figure 2). So, in order to prevent bank conflicts, the threads each start counting from a different row so that all N 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.

3. EXPERIMENTAL RESULTS

In our experiments, we compared our new parallel version of the algorithms running on a GPU to the original sequential one running on a CPU. The algorithms are executed on a static test set consisting of 1000 captured images for a stable average. The images have VGA resolution, which is the resolution of the 208 fps FireWire camera we employ. Note that none of the processing times depends on the actual contents of the images.

Our evaluation system has an Intel Core 2 Duo CPU with two cores running at 2.13 GHz. The sequential image registration is strictly running in one thread though, so no performance gain due to the second CPU core is to be expected. The graphics device used is a GeForce GTX 480 from Nvidia. It has 15 multiprocessors running at 1.4 GHz. Each of them can process 32 parallel threads running the same code. This leads to a total of 480 concurrent threads. The shared memory is divided into 32 banks.

Operation	GPU		CPU	
	μ [ms]	σ [ms]	μ [ms]	σ [ms]
Brightness Hist.	0.076	0.009	0.503	0.061
Row / Col. Hist.	0.278	0.010	5.312	0.676
Cross Correlation	0.245	0.013	1.639	0.070

Table 2. Mean (μ) and standard deviation (σ) of the computation time taken by GPU (parallel) and CPU (sequential) for the considered parts of the algorithm. All measurements were performed on 1000 VGA images.

Table 2 shows the mean processing times taken by GPU and CPU for the components of our algorithm we modified. The components are executed as many times as necessary for the registration of an image pair. That is, we create one brightness histogram, eight row and column histograms and perform one NCC. The table shows that the creation of row and column histograms can be sped up by a factor of 19 while the other two components are 7 times faster on the GPU. In the CUDA architecture, processing time scales linearly with the number of blocks to process. The computation time of the histograms is thus approximately linear in the number of image pixels. This behavior is identical for both implementations and has been verified in our experiments. The computation time of the NCC mainly depends on the search range of the shift which is a constant. If more than one pair of images is registered, the entire process is repeated for each pair.

Note that the numbers presented here only contain raw processing time of the parallelized parts. In a realistic scenario, image data and intermediate results have to be transferred between the machine's RAM and the graphics card as some steps are calculated on the GPU and some are left on the CPU. This decreases the overall performance gain. All considered, registering a pair of exposures takes 7.69 ms on the CPU and 1.30 ms on the GPU. This means a decrease of overall processing time by a factor of 5.9, leaving enough time per HDR frame available for color space conversions, frame merging and tone mapping. The resulting HDR frame rate also depends on these processes.

4. CONCLUSIONS

We presented our optimization for image registration of low dynamic range exposures. We analyzed the steps of the existing method to detect the most time-critical components and to assess their suitability for implementation on a GPU. Performance was improved by a factor varying from 7 to 19 with an average overall speed-up of 5.9:1 for a pair of registered exposures.

5. REFERENCES

[1] 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, pp. 17–30, 2003.

[2] B. Guthier, S. Kopf, and W. Effelsberg, “Capturing high dynamic range images with partial re-exposures,” in *Proceedings of the IEEE 10th Workshop on Multimedia Signal Processing (MMSP)*, 2008, pp. 241–246.

[3] 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)*, Sept. 2010, pp. 145–148.

[4] R. Strzodka, M. Droske, and M. Rumpf, “Image registration by a regularized gradient flow. A streaming implementation in DX9 graphics hardware,” *Computing*, vol. 73, no. 4, pp. 373–389, 2004.

[5] A. Köhn, J. Drexl, F. Ritter, M. König, and H. Peitgen, “GPU accelerated image registration in two and three dimensions,” *Bildverarbeitung für die Medizin 2006*, pp. 261–265, 2006.