Fast GPU-based image warping and inpainting for frame interpolation

Jakub Rosner[1]
jakub.rosner@joanneum.at

Hannes Fassold[2]
hannes.fassold@joanneum.at

Peter Schallauer[2]
peter.schallauer@joanneum.at

Werner Bailer[2]
werner.bailer@joanneum.at

ABSTRACT

Frame interpolation (the insertion of artificially generated images in a film sequence) is often used in post production to change the temporal duration of a sequence, e.g. to achieve a slow-motion effect. Most frame interpolation algorithms first calculate the motion field between two neighboring images and scale it appropriately. Afterwards, the images are warped (mapped) with the scaled motion field, and regions to which no source pixel was mapped are filled up (image inpainting). In this paper, we will focus on the latter two steps, the warping of the images and the image inpainting. We present simple and fast algorithms for image warping and inpainting, and discuss their efficient implementation to GPUs, using the NVIDIA CUDA technology. We compare the CPU and corresponding GPU routines and notice a speedup factor of approximately 6 - 10 for image warping and image inpainting. Significantly higher speedups can be expected for the latest NVIDIA GPU generation codenamed *Fermi* due to several architectural improvements (faster atomic operations, L1/L2 cache). When comparing the result images of the CPU and GPU routine visually, practically no difference can be seen.

Keywords

Image warping, image inpainting, frame interpolation, GPU, CUDA, GPGPU

1. INTRODUCTION

Frame interpolation (the insertion of artificially generated frames in a film sequence) is a commonly used method in video and film post production. It can be used for converting a given film sequence to a slow-motion sequence (also known as retiming). Also when doing film restoration, one can replace missing frames, or frames which have been badly damaged, by artificial frames created by frame interpolation.

Typical frame interpolation algorithms operate in the following way (see Figure 1): As a first step, the pixel-wise motion (optical flow) between the two temporally neighboring images of the interpolated image (which is to be calculated) is estimated. A dense motion field is retrieved, whose motion vectors then are scaled linearly according to the desired temporal position of the interpolated image. After that, one neighbor image is warped with the scaled motion field to get the interpolated image. The term *image warping* means that each pixel of the source image is mapped (translated) with its motion vector and written into a destination image. The interpolated image typically has holes, regions in the image to which no source pixel was mapped to. So the last step is to fill these regions with an image inpainting algorithm. One can apply this procedure to both neighbor images and gets two interpolated images, which can be combined to one image e.g. by some sort of blending . In this work, we will focus on the last two steps, image warping and inpainting, and on their efficient implementation on the GPU using the CUDA technology. We will not describe the calculation of the motion field, as there are efficient GPU-based algorithms available (e.g. [Wer09]) which we will take advantage of.

[1] Silesian University of Technology, PhD faculty of Data Mining, Ulica Academicka 16, 44-100 Gliwice, Poland

[2] JOANNEUM RESEARCH, Institute of Information Systems, Steyrergasse 17, 8010 Graz, Austria

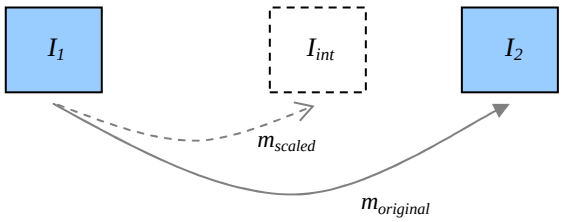

Figure 1: Illustration of the workflow for frame interpolation. I_{int} is the interpolated image, I_1 and I_2 its neighbors, $m_{original}$ the calculated motion field between I_1 and I_2 and m_{scaled} the scaled motion field.

Device Architecture and is a general-purpose GPU programming environment introduced by NVIDIA, allowing the programmer to utilize the massive processing power of current generation GPUs.

In this document, we first give an introduction into GPU programming and CUDA (see next section). In section 3 we discuss shortly previous work on implementing image warping and image inpainting on the GPU. In section 4 and 5, we give an description of the algorithms we developed in our research group for image warping and image inpainting. After that, in section 6 we describe how we ported our CPU algorithms to CUDA. Finally, in section 7 experiments are done to compare the algorithms and their respective GPU implementations in terms of quality and speed.

2. GPU PROGRAMMING & CUDA

In the last years, GPUs have gained significant importance in computer vision and other scientific fields. A number of basic computer vision algorithms has already been implemented efficiently on GPUs, be it optical flow calculation [Wer09], feature point tracking [Fas09] or SIFT features [Sin06]. Typically they provide a speedup of an order of magnitude with respect to a reference CPU implementation, depending on the algorithm's ability to be executed in a massively parallel way. Most GPU implementations use CUDA as it is currently the best supported programming environment.

A CUDA program is typically composed of a control routine, which calls a couple of CUDA *kernels*. A *kernel* is similar to a function, but is executed on the GPU in parallel by a larger number of threads (typically thousands). Groups of 32 consecutive threads are organized into *warps*. Furthermore, sets of up to 512 threads are grouped into *thread blocks*, which then form a *grid*.

An important property of NVIDIA GPUs is *shared memory*, which is a small, but very fast cache which has to be managed by the user. There are also other important memory types with different properties, e.g. texture memory (read-only, cached), constant memory and global memory (read-write, high latency). Also atomic functions are very helpful when different threads try to access the same memory location.

For a more detailed description we refer to the publications [Fas09] and [Ryo08] where GPU/CUDA programming is explained more in depth and guidelines are given for porting algorithms efficiently to CUDA.

3. RELATED WORK

Although the literature for image inpainting algorithms is huge (e.g. see [Ber00][Bor07][Che10] [Cri03][Fid08]), there are not many algorithms which have been reported to run on the GPU. This might be because a significant amount of them have a rather complicated workflow or an implicit serial nature which can not be easily mapped to a GPU. In fact, to our knowledge only for one algorithm [Har01] a corresponding GPU implementation has been described[5]. It is implemented in shading language[6], having the disadvantage that the algorithm has to be adapted to fit to the computer-graphics oriented render pipeline. This adaption typically leads to a more complicated implementation and performance degradation. Regarding image warping, a survey of various warping methods can be found in [Wol90].

4. IMAGE WARPING

Algorithm

Image warping is a fundamental task in image processing. Given an source image I and a dense motion field, one wishes to generate a warped image I_{warped} where all the pixels in I have been translated by their corresponding motion vector.

Note that, depending on the motion field, multiple pixels of the source image may map to the same place in the warped image. On the other hand, there may be areas in the warped image to which no source pixel was mapped to, leading to holes in the warped image. Filling up those areas will be described later in this document in section 5.

The algorithm we propose for image warping needs an additional accumulator image and a weight image. Both are floating point (fixed-point is also possible) and are initially set to zero. Now for each source pixel its destination position is calculated, using the mapping defined by the dense motion field. As we

[3] http://www.nvidia.com/object/cuda_home_new.html

[5] http://www.eecs.harvard.edu/~hchong/goodies/inpaint.pdf

[6] http://www.opengl.org/documentation/glsl/

can not write directly to the destination position (it typically has non-integer coordinates), we instead 'write' into the four surrounding pixels of the destination position (one can imagine this as sort of 'bilinear writing'). For that, we *increment* the four surrounding pixels in the accumulator image and also in the weight image. The amount of increment depends on the distance of the destination position to the specific pixel neighbor.

The usage of an accumulator image solves the problem that multiple source pixels possibly map to the same destination pixel. The resultant intensity in the warped image will be a weighted combination of the source pixels intensities.

Finally, the intensity values of the warped image I_{warped} is calculated by dividing the accumulator image pixel-wise by the weight image. Areas to which no source pixel was mapped (holes) are identified by having a zero value in the weight image. A hole mask is generated which is needed for the inpainting process, which is described in the next section. Note that the proposed image warping algorithm is quite fast as it has has linear complexity with respect to the number of image pixels.

5. IMAGE INPAINTING

Algorithm

The input for the image inpainting algorithm is an intensity image *I* and a hole mask *H* which defines the areas of then intensity image, which should be inpainted. In the following, we give an outline of our proposed inpainting method. It needs an additional floating-point accumulator image *A* and weight image *W*. Both are initially set to zero. For multi-channel images, each channel is calculated separately.

First, the set of border pixels of all holes are determined. Now for *each* border pixel, its intensity is *propagated* into the hole in a fixed set of directions (typically 16, equally distributed over the 360 degree range). See Figure 3 for an illustration of the process. The propagation is done in the following way: For a specific border pixel and a specific direction, a line-tracing using the Bresenham algorithm [Bre65] is performed, starting at the border pixel and ending when the line hits the opposite side of the hole. The Bresenham algorithm is slightly modified so that during line-tracing it updates also the approximate distance d_{curr} from the current pixel to the start border pixel. Now, for each visited pixel *p* during line-tracing, its corresponding accumulator image value *A(p)* and weight image value *W(p)* are incremented according to $A(p) = A(p) + \frac{1}{d_{curr}} g_b$ and

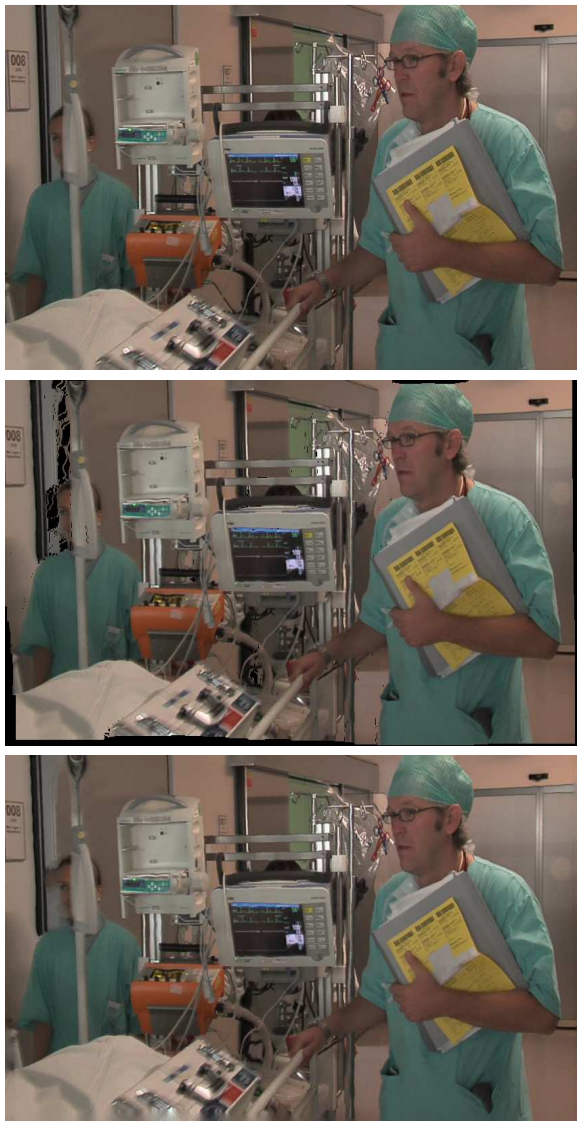

Figure 2: From top to bottom: input image, warped image, final interpolated image where holes have been inpainted.

value of the start border pixel. One can see from this that border pixels which are nearer to a given hole pixel have a higher contribution to its intensity value, as the increment in the accumulator image will be higher for them.

After having done the propagation for all hole border pixels and all directions, the intensities values for the regions to be inpainted can be calculated simply by dividing the accumulator image pixel-wise by the weight image.

A problem of the proposed method is that due to using a fixed set of directions, it can introduce star-shaped artefacts into the inpainted regions. In order to reduce these artefacts, we enforce an additional post-processing step.

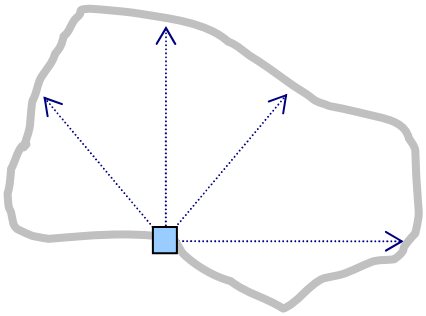

Figure 3: A specific hole border pixel is propagated into the hole in a fixed set of directions.

For this purpose, we first calculate a distance map for the hole regions, which gives for each hole pixel approximately its distance to the *nearest* hole border pixel. Note that fast algorithms for calculating the distance map are available [Bor86]. Now each hole pixel is blurred with a distance-adaptive box kernel, with kernel sizes ranging from 3 (for hole pixels near the border) to 9 (inner hole pixels).

One can increase the quality of the inpainted regions by using more directions in the propagation step. On the other hand, also the runtime increases linearly with the number of directions. According to experiments, a value of 16 seems to be a good compromise between quality and runtime.

In Figure 4 the results of proposed image inpainting algorithm (using 16 directions) for some commonly used test images[8] can be seen. As the algorithm is solely diffusion-based, blurring can be observed in the inpainted regions. Note that the proposed algorithm shares some loose similarities with inpainting methods using radial basis functions (RBF) [Uhl06].

6. CUDA IMPLEMENTATION

The following section describes some CUDA - specific issues resolved while implementing the image warping and inpainting algorithms for GPU.

The first step is to transfer the source data from CPU memory to GPU memory, unless it already resides in GPU memory. E.g. when the optical flow is calculated with a GPU-based method, then the motion field is already on the GPU and doesn't have to be transferred.

In order to minimize allocation and deallocations of GPU memory, we use a context object which holds the necessary temporary data buffers throughout the whole sequence. The context object can be used for both algorithms (warping and inpainting).

Image warping

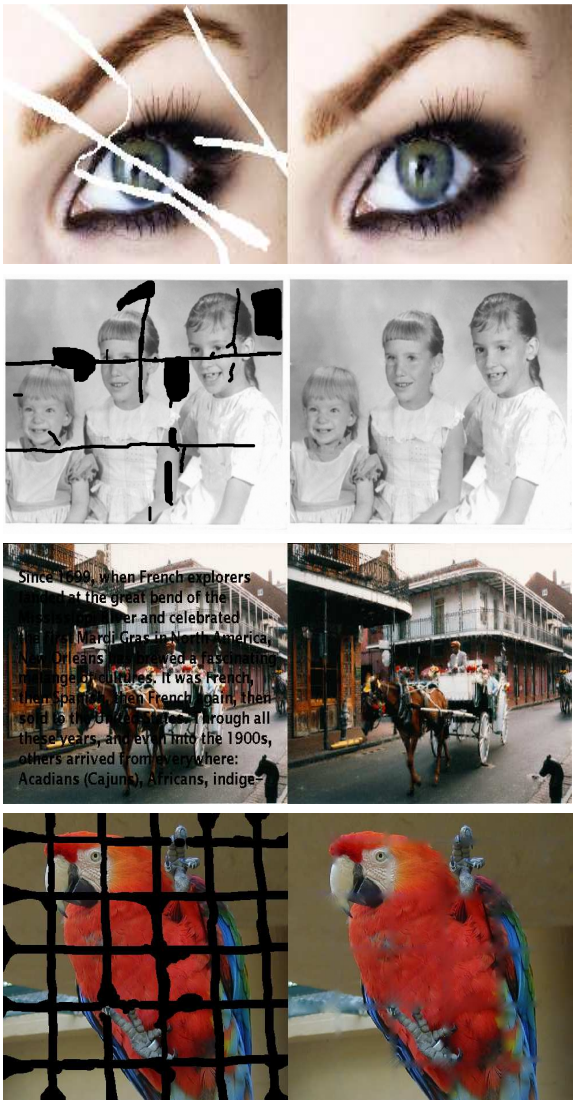

Figure 4: Image inpainting results for the images Eye, Girls, New Orleans and Parrot.

The image warping algorithm has been implemented in CUDA in two steps. The first one calculates the accumulator and weight images and the second one then calculates the warped image.

The first step of this algorithm, while being quite straightforward to implement efficiently on the CPU, turns out to be problematic to optimize on graphics processor. The reason is that, multiple GPU threads possibly try to increment the same value in the weight or accumulator images simultanously, leading to read-write hazards. To handle this we have to use *atomic* increment operations which serialize the workflow, but at a large cost in performance.

To reduce this penalty, each thread block (usually 16x16 threads) determines an approximate region in the destination image where its threads will likely be mapped to. As the runtime for executing operations using shared memory is much lower than executing

[8] www-m3.ma.tum.de/bornemann/InpaintingCodeAndData.zip

them using global memory, each thread atomically increments the four pixels around the destination position in global memory only if this position falls outside the approximated region. Otherwise it increments the appropriate values in shared memory, and after all the threads are completed, the whole region is copied into the destination image. Note that the more 'regular' the motion field is, the the more atomic operations in fast shared memory are done.

The final step of the warping algorithm, unlike the first one, is pretty straightforward. It should be noted that for performance reasons, computing the destination value from accumulator and weight image is performed in shared memory.

Image inpainting

As first step of the image inpainting process, we have to determine the position of all hole border pixels. For this, we first do a 3x3 dilation operation followed by subtraction of the original mask. In the resultant image only hole border pixels have non-zero value. To get a list of their positions, we apply a *compaction* operation which filters out zero-valued pixels. The first two operations are relatively easy to implement on a GPU and a highly efficient compaction algorithm for GPU which was used by us can be found in the CUDA performance primitives (CUDPP) library[9] .

As the next step, the line tracing, involves propagation in different directions, we would encounter cases where multiple threads try to modify the same value at the same time, which would demand the usage of slow atomic functions.In order to avoid this, we split the algorithm into separate kernels, each tracing lines in exactly one direction from each border pixel and therefore prevent multiple-way access hazards. Note that in the line tracing process, each hole border pixel is assigned to one CUDA thread.

To calculate the intensity value for each hole pixel from the accumulator and weight image we use a similar kernel to the one used in image warping, but modify it slightly to compute only the hole regions.

In the last step, the distance-adaptive blurring of the hole regions, for every hole pixel the neighbor pixels for the maximum possible box kernel window size (9x9) are read in, to avoid divergent threads. After that, only the actual needed neighbor pixels (according to the window size for the hole pixel) are used for calculating the result value of the box filter.

7. EXPERIMENTS AND RESULTS

In this section we will describe the results from comparing our CUDA implementation against a optimized CPU implementation. The runtime measurements were done on a 3.0 GHz Intel Xeon Quad-Core machine, equipped with a NVIDIA GeForce GTX 285 GPU. The tests have been performed for two commonly used resolutions: Standard Definition (SD) with 720x576 pixels and High Definition 1080p (HD) with 1920x1080 pixels. Note that all the test images are 3-channel color images with 8 bit per channel.

Quality test

The quality results show that our CUDA implementation of image warping provides the same results in term of quality as the corresponding CPU routine, yet there are some minor differences in the image inpainting. Those differences however are visually indistinguishable and occur only for a small fraction of pixels (on average 8 pixels for SD and 50 pixels for HD have a difference which is higher than a few gray values).

Runtime test

For doing the runtime comparison, we simulate the frame interpolation scenario. For that, we calculate the motion field between neighboring frames of a short video sequence (10 frames) with the method described in [Wer09] and then do the image warping and inpainting.

In the warped image, on average 1.4 % of the pixels are to be inpainted. For the image inpainting, 16 directions are used. The allocation and deallocation of the context object altogether takes approximately 0.3 milliseconds for SD and 0.6 milliseconds for HD. These times and the time needed for transferring the input image to GPU memory are not included in the given runtime of the GPU implementations. Note that in our application, the input images are already on the GPU as the first step (the calculation of the motion field) was also done on the GPU.

The average speedup (see Figure 5 and Figure 6) which is achieved by the GPU implementations of the algorithms is up to an order of magnitude, clearly demonstrating how advantageous the usage of GPUs can be for sufficiently parallizable algorithms. All runtime numbers are given in milliseconds.

[9] http://www.gpgpu.org/developer/cudpp

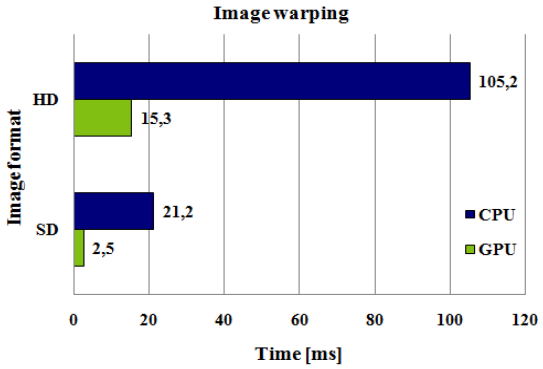

Figure 5: Runtime of the image warping.

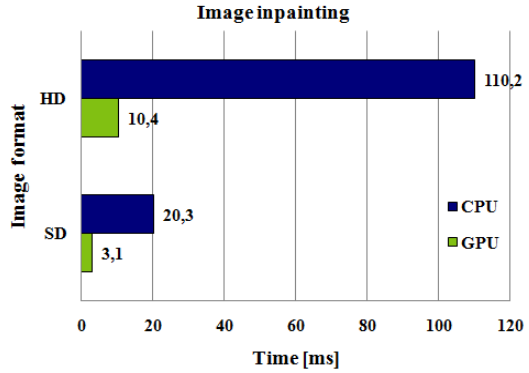

Figure 6: Runtime of the image inpainting.

Note that the speedup for image warping is larger for smaller formats, which is counter-intuitive as usually algorithms running on the GPU are more effective for larger sets of data. The reason is that for smaller images the part of image covered by the thread block's shared memory window is relatively larger.

8. CONCLUSION

Simple and fast algorithms for image warping and image inpainting for usage in frame interpolation have been presented, and their efficient implementation to the GPU was described. Experiments were done which show an significant speedup factor of 6 – 10 for the GPU implementations of image warping and inpainting. It is expected that in the future this factor keeps the same or even increases as currently GPU generation cycles are shorter than CPU generation cycles.

9. ACKNOWLEDGMENTS

This work has been funded partially under the 7th Framework program of the European Union within the project "PrestoPRIME" (FP7-ICT-231161). Furthermore, the work of Jakub Rosner was partially supported by the European Social Fund.

10. REFERENCES

[Ber00] M. Bertalmio, G. Sapiro, V. Caselles, C. Ballester, Image inpainting, International Conference on Computer Graphics and Interactive Techniques, 2000

[Bre65] J. E. Bresenham, Algorithm for computer control of a digital plotter, IBM Systems Journal 4, 1965

[Bor86] G. Borgefors, Distance transformations in digital images, Computer Vision, Graphics, and Image Processing, Volume 34, 1986

[Bor07] F. Bornemann, T. März, Fast image inpainting based on coherence transport, Journal of Mathematical Imaging and Vision, Volume 28, 2007

[Che10] X. Chen, F. Xu, Automatic image inpainting by heuristic texture and structure completion, 16th International Multimedia Modeling Conference, 2010

[Cri03] A. Criminisi, P. Perez, K. Toyama, Region filling and object removal by exemplar-based inpainting, IEEE Transactions on Image Processing, Volume 28, No. 9, 2004

[Fas09] H. Fassold, J. Rosner, P. Schallauer, W. Bailer, Realtime KLT Feature Point Tracking for High Definition Video, GravisMa workshop, Plzen, 2009

[Fid08] I. Fidaner, A survey on variational image inpainting, texture synthesis and image completion, Bogazici University, 2008

[Har01] P. Harrison, A non-hierarchical procedure for re-synthesis of complex textures, Proceedings of WSCG, Plzen, 2001

[Ryo08] S. Ryoo, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008

[Sin06] S. Sinha, J. Frahm, M. Pollefeys, Y. Genc, GPU-Based Video Feature Tracking and Matching, EDGE workshop, 2006

[Uhl06] K. Uhlir, V. Skala, Radial basis function use for the restoration of damaged images, Computational Imaging and Vision, Volume 32, 2006

[Wer09] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers, H. Bischof, Anisotropic Huber-L1 Optical Flow, Proceedings of the British Machine Vision Conference, London, UK, 2009

[Wol90] G. Wolberg, Digital Image Warping, IEEE Computer Society Press, 1990