JOANNEUM RESEARCH

DIGITAL

Institute of Information and Communication Technologies

Computer Vision Algorithms for Automating HD Post-Production

Hannes Fassold, Jakub Rosner

2010-09-22

SEVENTH FRAMEWORK PROGRAMME

fascinate

2020 3DMedia

Presto PRIME
KEEPING AUDIOVISUAL CONTENTS ALIVE

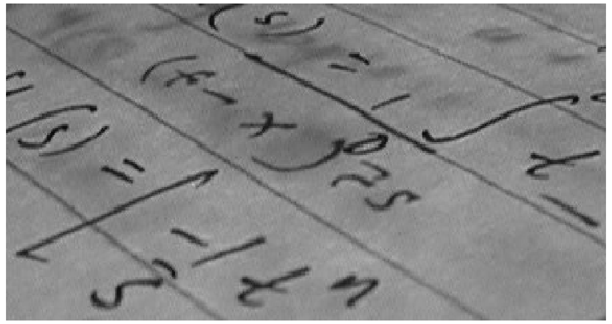
Overview

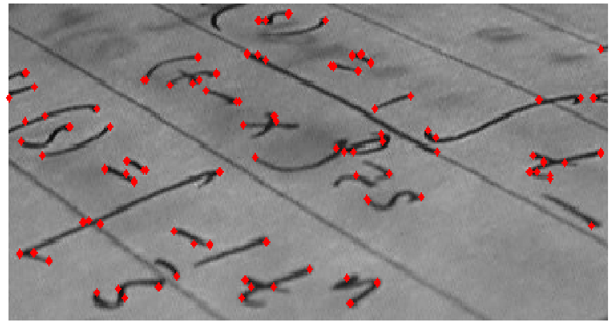
- Harris/KLT feature point detection
- KLT feature point tracking
- Application
  - Real-time HD stabilization

- Image warping
- Image inpainting
- Application
  - Re-Timing (‚Time-Stretching‘)
  - Restoration of damaged / missing frames

Feature point detection
Introduction

- Find ‘reliable’ feature points in image
  - Usage
    - ✓ Camera calibration
    - ✓ Tracking
    - ✓ …
  - Reliable feature points have sufficient structure in their local neighborhood
    - E.g. point within homogeneous area not reliable
  - Reliable feature points typically look corner-like





Feature point detection
Measures

- Structure matrix *G*
  - 2 x 2 matrix *G* encodes structure information for an rectangular area *W(p)* around a point *p*
  - Gradient image → Central Difference, Sobel or Sharr operator

$$G = \sum_{x \in W(p)} \nabla I(x) \cdot \nabla I(x)^T$$

- Cornerness measures
  - ***Harris*** measure: *λ* = det(*G*) – *k* * trace(*G*)^2
  - ***KLT*** measure: *λ* = minimum eigenvalue of *G*
  - *λ* small or zero → homogeneous image area
  - *λ* big → corner, richly textured area

Feature point detection Algorithm

- As in OpenCV routine ‚cvGoodFeaturesToTrack‘
- Algorithm steps
  1. Calculate cornerness λ for all pixels
  2. Calculate maximum cornerness λ_{max} in image
  3. Discard all pixels which λ smaller than a fraction of λ_{max} (e.g. < 5%)
  4. Non-maxima suppression (discard ‚weak‘ local maxima)
  5. Minimum distance enforcement

GPU

CPU

- Minimum distance enforcement
  - Ensures that every feature point has a certain minimum distance to all other points
  - Avoids clumping of most feature points in richly textured image regions
  - Some issues with it which force us to do it on CPU (more later) …

Feature point detection
Steps 1-4, CUDA implementation

- All kernels make extensive usage of shared memory
- Cornerness calculation
  - Three kernels for convolution, structure matrix, cornerness (KLT formula)
- Determine maximum cornerness λ_{max}
  - Is a reduction operation → CUDPP library
- Discard feature points with low cornerness
  - Set a ‚discard flag‘ for each pixel to be discarded
- Non-maxima suppression
  - Kernel is variation of dilate operator
- (Before 5) Transfer non-discarded pixels to CPU
  - Before transfer, all discarded pixels are filtered out by a compaction operation → CUDPP library

Feature point detection
Step 5

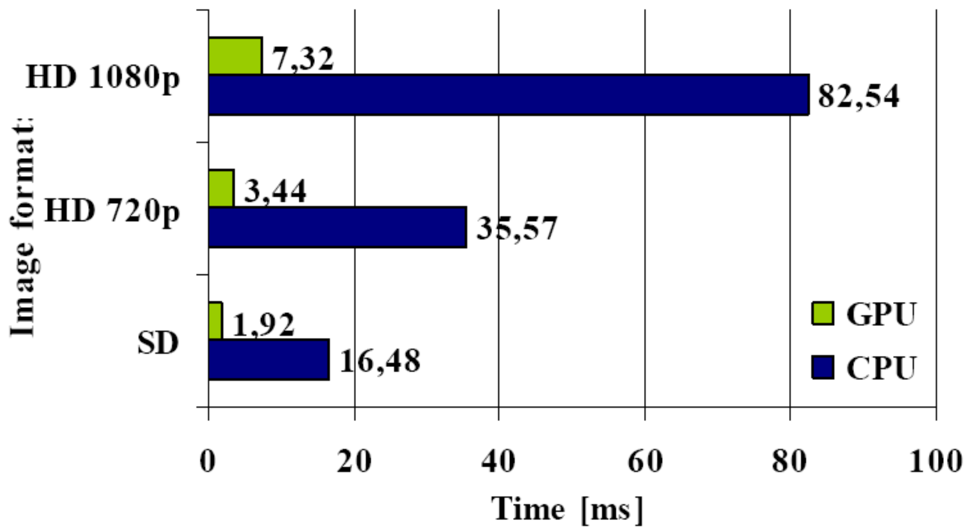
- Minimum distance enforcement
  - Given: ‚candidate list‘ (all pixels which haven‘t been discarded)
  - Iterate through list, starting with candidates with highest cornerness, and add them to output list
  - Before adding a candidate, its distance to all points already in output list is checked
- Issues
  - Process is inherently serial → forces us to do on CPU
  - OpenCV implementation not efficient for several thousand points
    - Developed alternative method
    - Principle: When adding a candidate, the circular area around it is marked as ‚occupied‘.
    - Linear complexity
    - Automatic switching between OpenCV and alternative method

Feature point detection
Results

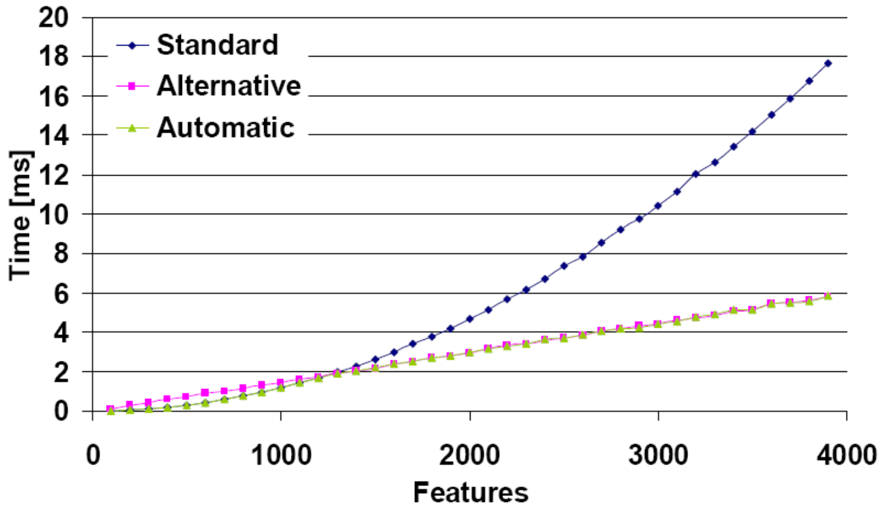
Feature point detection
Results

Feature point detection
Runtime comparison

- GPU impl.: CUDA, GTX 280
- CPU impl.: OpenCV (using IPP), 2.4 Ghz Xeon Quad-Core
- Window size = 5 x 5, Maximum # of features = 10000



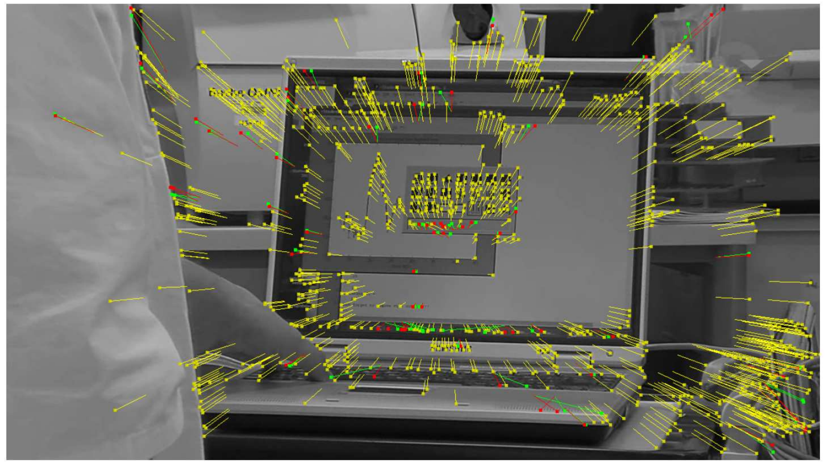
Runtime for the steps 1 – 4 (feature point detection without minimum distance enforcement)



Runtime for step 5 (minimum distance enforcement)

Feature point tracking
Introduction

- Feature point tracking
  - Given a sparse set of feature points in current image *I* (e.g. found by feature point detection), find their position in subsequent image *J*
- Important low-level task in computer vision
  - Used for object tracking, camera motion estimation, structure from motion, …
- KLT algorithm (Kanade, Lucas, Tomasi)
  - Very popular method
    - Reasonably fast, fully automatic, sufficient quality
  - For each frame *I* in sequence
    - Detect new features in *I* and add them to already existing ones
    - Track all features from *I* to subsequent image *J*



Feature point tracking
Algorithm principle

- Dissimilarity function $\varepsilon(v) = \sum_{x \in W(p)} (J(x+v) - I(x))^2$
  - *p* … point, *v* .. motion vector,
    W(*p*) .. *n* x *n* window centered at *p*
- For each point *p*, find motion *v* that minimizes ε(*v*)
- Minimization of ε(*v*)
  - Gradient descent method
    (iterative method, Gauss-Newton type)
  - Gradient descent methods need ‚good‘ initial value v_0
    - Create multi-resolution image pyramid
    - Do minimization on each level of pyramid
    - Solution of level *m* + *1* is used as initialization for level *m*

Feature point tracking Algorithm

- Pseudo-Code
  - For one Pyramid Level, for one point
  - Typically: W(p) = 5 x 5 pixel, maxIter = 10, eps = 0.03

1. Set initial motion vector $v_1 = (0,0)^T$
2. Spatial image gradient $\nabla I = \partial I / \partial (x, y)$
3. Calc. structure matrix $G = \sum_{x \in W(p)} \nabla I(x) \cdot \nabla I(x)^T$
4. for $k = 1$ to $maxIter$
   a) Image difference $\eta(x) = I(x) - J(x + v^k)$
   b) Calc. mismatch vector $b = \sum_{x \in W(p)} \eta(x) \cdot \nabla I(x)$
   c) Calc. updated motion $v_{k+1} = v_k + G^{-1} b$
   d) if $\| v_{k+1} - v_k \| < eps$ then stop (converged)
5. Report final motion vector v

Feature point tracking
CUDA implementation

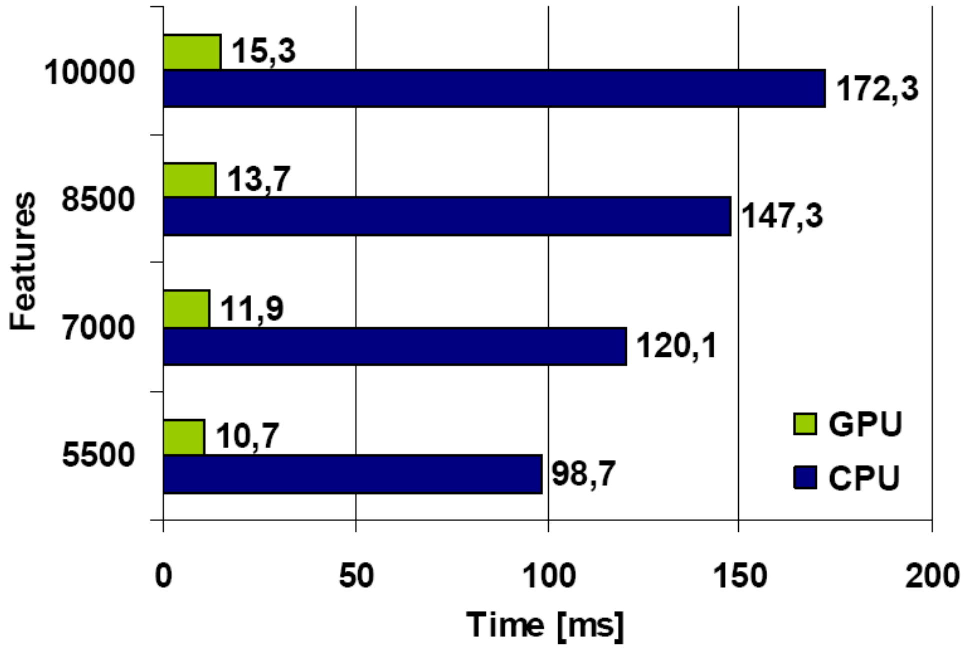
- Gaussian image pyramid
  - Convolution + subsampling
- Feature point tracking (key issues)
  - One kernel call for each pyramid level
  - One thread = one point
    - GPU under-utilization if # points is too small (e.g. some hundred points)
  - Reduce # of texture fetches, especially in inner loop
  - Each thread needs lot of shared memory
    - Especially for bigger window sizes (9x9, 11x11, ..) this leads to low multiprocessor occupancy
  - Difficult to find best compromise (thread block size, # registers per threads…)
    - Lot of experimentation necessary, need to implement different variants

Feature point tracking Results

/ATION

Feature point tracking
Runtime comparison

- GPU impl.: CUDA, GTX 280
- CPU impl.: OpenCV (using IPP), 2.4 Ghz Xeon Quad-Core
- FullHD (1920 x 1080), window size = 5 x 5, #levels = 6, maxIter = 10, eps = 0.03

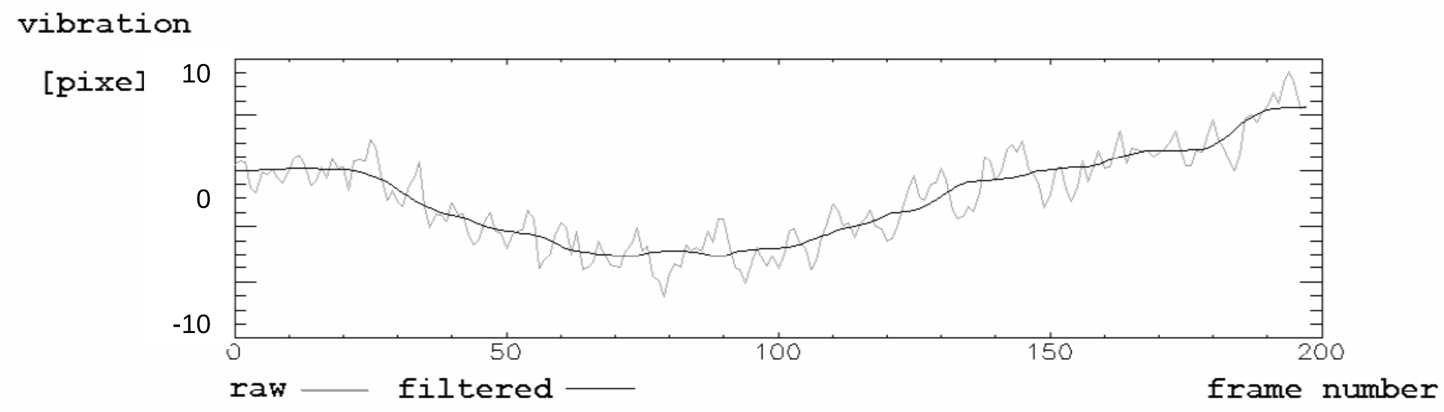


Feature point tracking
Application: Stabilization

- Problem
  - Annoying film experience due to image ‚vibration‘
  - Possible reasons
    - shaky camera
    - instability in film transport during film scanning (‚worn out perforations‘)
- What we want
  - Reduce/remove these vibrations
  - … but leave intended (typically ‚smooth‘) camera motion intact

Feature point tracking
Application: Realtime HD Stabilization

- Algorithm outline
  - Track feature points throughout sequence
  - Robustly estimate ‚global‘ motion between consecutive frames
    - 2-parameter translational model (dx, dy)
    - Higher-parameter model possible (e.g. affine model)
  - Filter signal to get amount of correction
  - Warp frames with correction



Feature point tracking Stabilization Demo

- Stabilization with 2-parameter translational model
  - Works **realtime** (> 25 fps) for Full HD resolution
    - GPU: GTX 285, CPU: QuadCore Xeon
  - [Video_Steyrer_gasse]

Image warping
Introduction

- Given an source image *I* and a nonlinear mapping *M*, calculate the mapped image *M*(*I*)
- Image warping examples
  - Rotation, Scaling, ..
  - Arbitrary mesh deformations
- Mapping function *M*
  - Typically defined pixel-wise

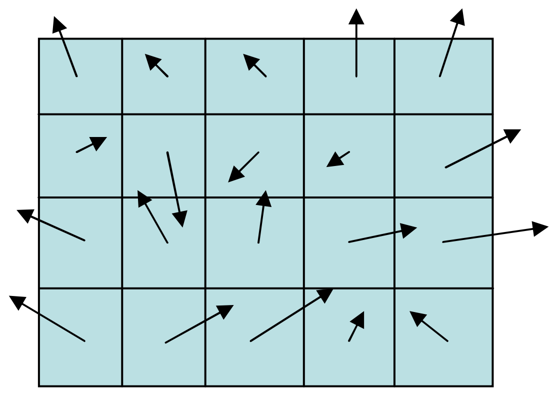


Image warping
Algorithm

- Use accumulator image A and weight image W
  - floating-point or fixed-point
- Algorithm
  - For each source pixel p
    - Determine destination location $dst = M(p)$
    - Increment the four surrounding pixels in accumulator and weight image $\rightarrow$ ‚bilinear writing‘
  - Warped image $M(I) = \frac{A}{W}$
    - Pixel-wise division
    - Pixels with weight zero are marked as ‚holes‘ (no source pixel mapped to them)

Image warping
CUDA implementation

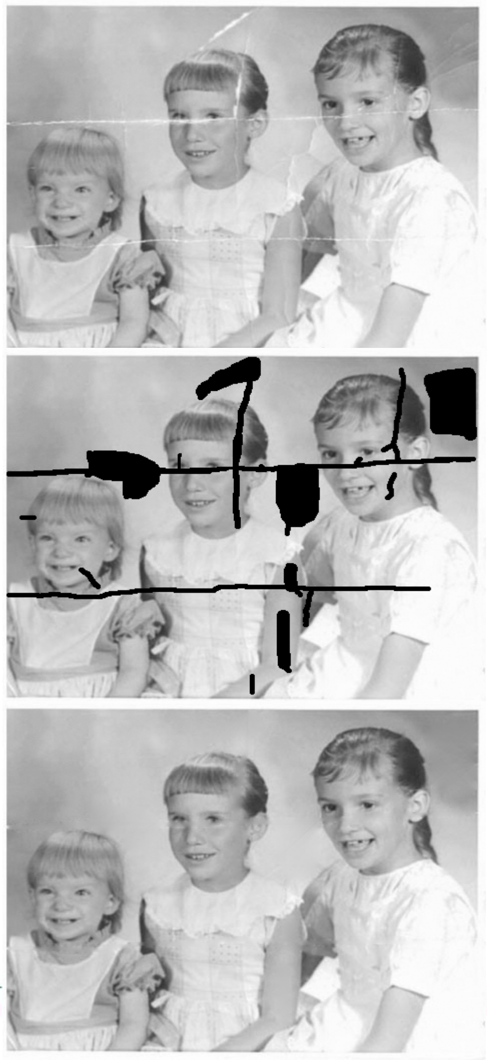
- Issues
  - Atomic operations necessary for resolving read-write hazards
    $\rightarrow$ significant performance penalty for pre-Fermi hardware
  - Reduce performance penalty
    - Determine a target region where most of threads of the current thread block will likely map to
      - Assume some sort of smoothness in mapping function *M*
    - Target region is cached in shared memory
    - Thread maps into target region $\rightarrow$ do *shared* memory atomic operation
    - Thread doesn‘t map into target region $\rightarrow$ do *global* memory atomic operation (slower)

Image warping
Source image

Image warping
Warped image

Image inpainting
Introduction

- Image inpainting
  - Fill up undefined regions in an image in the best way
  - Lot of literature about inpainting algorithms
    - Propagate structure & texture clever into hole
    - Still a hard task
- Goal
  - Develop simple and fast inpainting algorithm
  - Good parallizable
  - Suitable for holes occuring in warped images
    - Thin, crack-like appearance



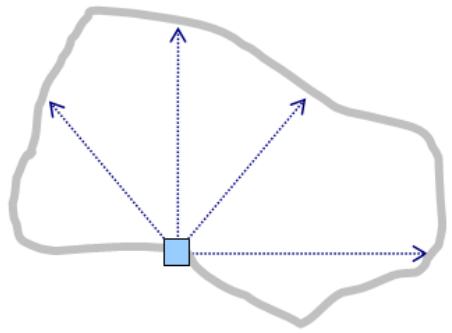
a TRADIT

Image inpainting Algorithm

- Approach
  - Uses accumulator image *A* and weight image *W*
  - Determine set of hole border pixels
  - For each border pixel
    - Propagate its intensity into the hole along a fixed set of directions (e.g. 16)
- Border pixel intensity propagation
  - Trace the line from border pixel into hole interior (Bresenham)
  - For each visited pixel *p* its value in *A* and *W* is updated

$$W(p) = W(p) + \frac{1}{d_{curr}} \qquad A(p) = A(p) + \frac{1}{d_{curr}} g_b$$

  - Inpainted image $I_{holefilled} = \frac{A}{W}$



Border pixel propagation along several directions

www.joanneum.at

a TRADITION of INNOVATION

Image inpainting
CUDA implementation

- Key issues
  - One thread = one border pixel
  - One kernel call per direction
    - All threads trace into same directions
  - Atomic operations **not** used
    - Speed reasons
    - Float atomic operations not supported for pre-Fermi GPUs
    - R/W Hazards can not occur for 8 main directions
    - R/W Hazards can occur (very seldomly) for 8 secondary directions → Induces negligible differences between CPU & GPU inpainting result
  - Warp divergence
    - Due to different paths the threads of a warp are tracing
    - Possible improvement: Group thread id's by some spatial relationship

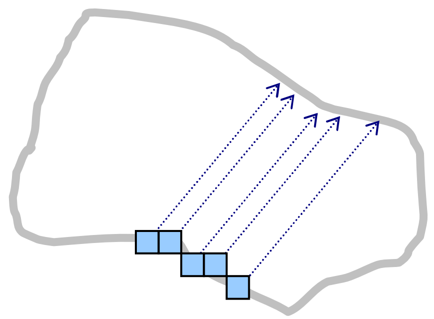
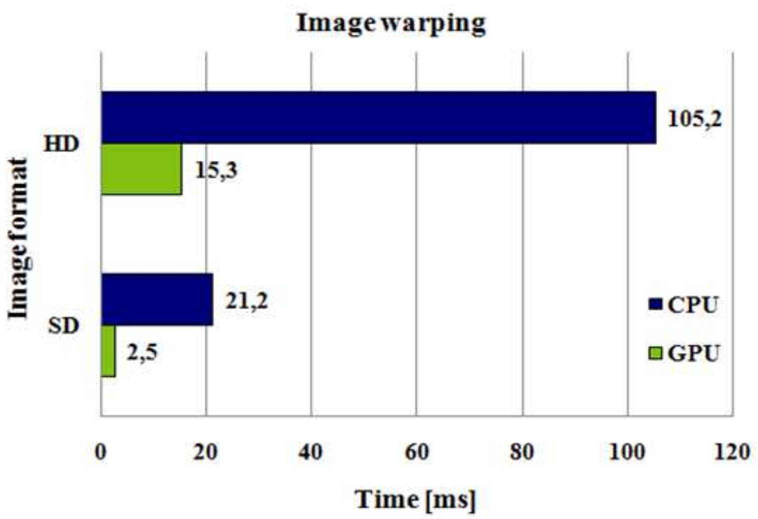


Image warping & inpainting
Runtime comparison

- GPU impl.: CUDA, GTX 285
- CPU impl.: Own optimized impl., **one** CPU-thread (but uses multi-threaded IPP-functions), Intel Xeon Quad-Core 3.0 Ghz
- Average runtime over sequence, warping functions = motion fields, ~ 1.3 % of warped images to be inpainted



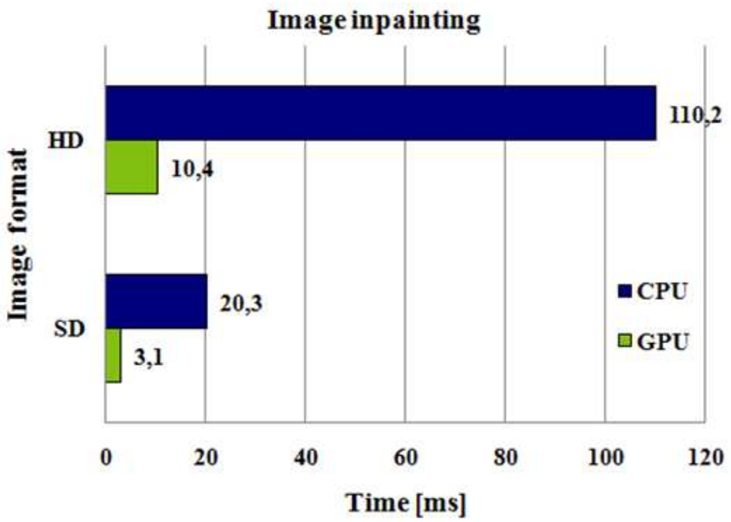


Image inpainting Results

Image inpainting & warping Application: Time-Stretching

- Time-Stretching effect
  - Insert synthetically generated frames in video sequence to achieve slow-motion effect
- Generate synthetic frame between image I_1 and I_2
  - Calculate pixel-wise motion (optical flow) between I_1 and I_2
    - Fast GPU methods available
  - Scale motion according to desired timepoint
  - Warp I_1 with scaled motion
  - Fill holes in warped image

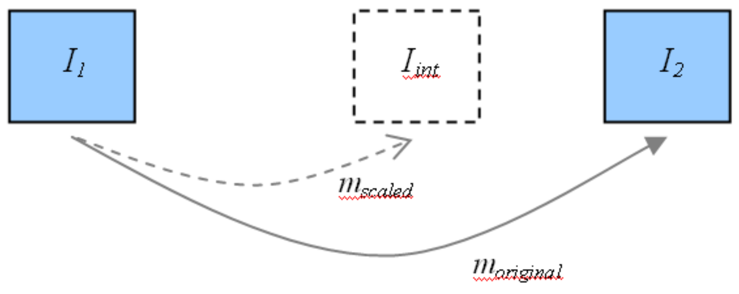
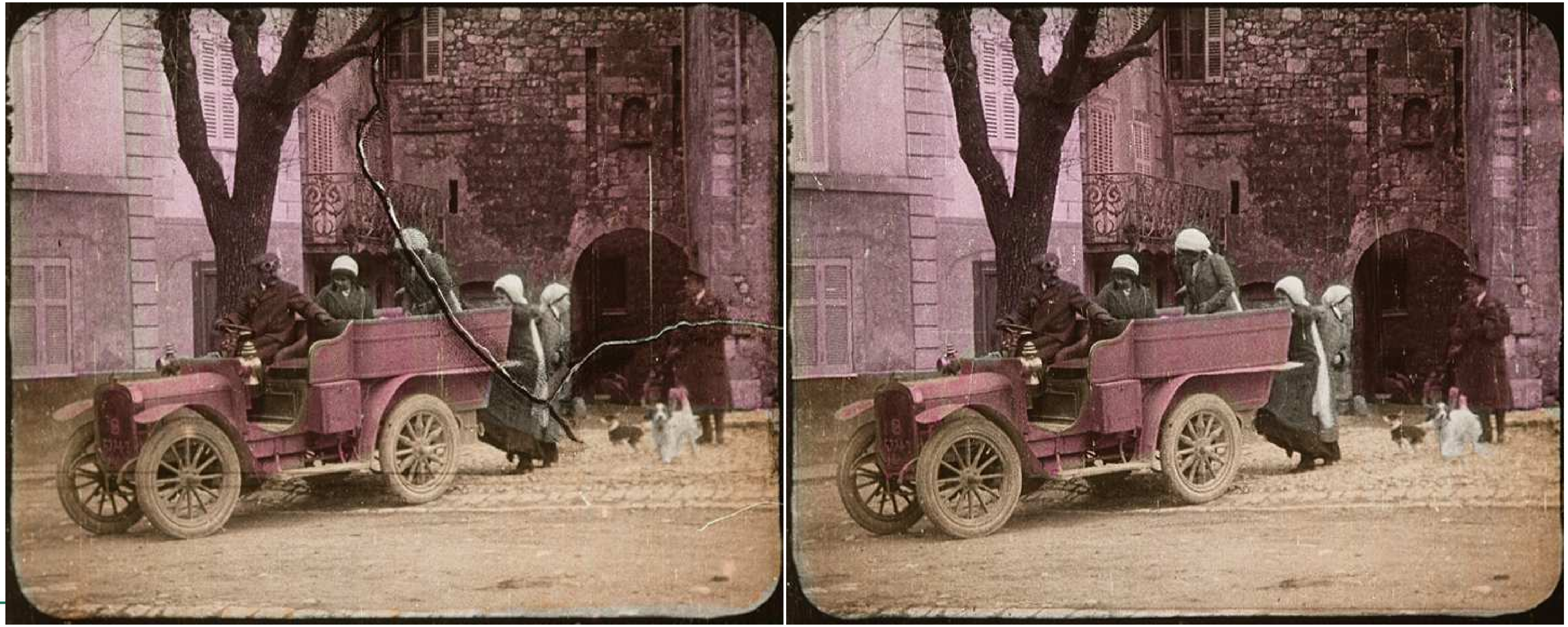


Image inpainting & warping
Demo: Time-Stretching

- Stretching Factor 2.0
- [DemoVideo ,TU Munich pedestrian area]

Image inpainting & warping
Application: Restore damaged frames

- Use neighbor frames to generate ‚replacement‘ for damaged frame



Acknowledgments

- Silesian University, Poland
  - Jakub Rosner
- JOANNEUM RESEARCH, Austria
  - Florian Putz, Hermann Fuerntratt, Werner Bailer, Peter Schallauer, Georg Thallinger
- Work was supported by European Union projects
  - 2020 3D Media (http://www.20203dmedia.eu)
  - FascinatE (http://www.fascinate-project.com)
  - PrestoPRIME (http://www.prestoprime.eu)

SEVENTH FRAMEWORK PROGRAMME

Contact

Hannes Fassold

DIGITAL - Institute of Information and Communication Technologies

JOANNEUM RESEARCH Forschungsgesellschaft mbH
Steyrergasse 17, A-8010 Graz, AUSTRIA

E-mail: hannes.fassold@joanneum.at
Web: http://www.joanneum.at/digital