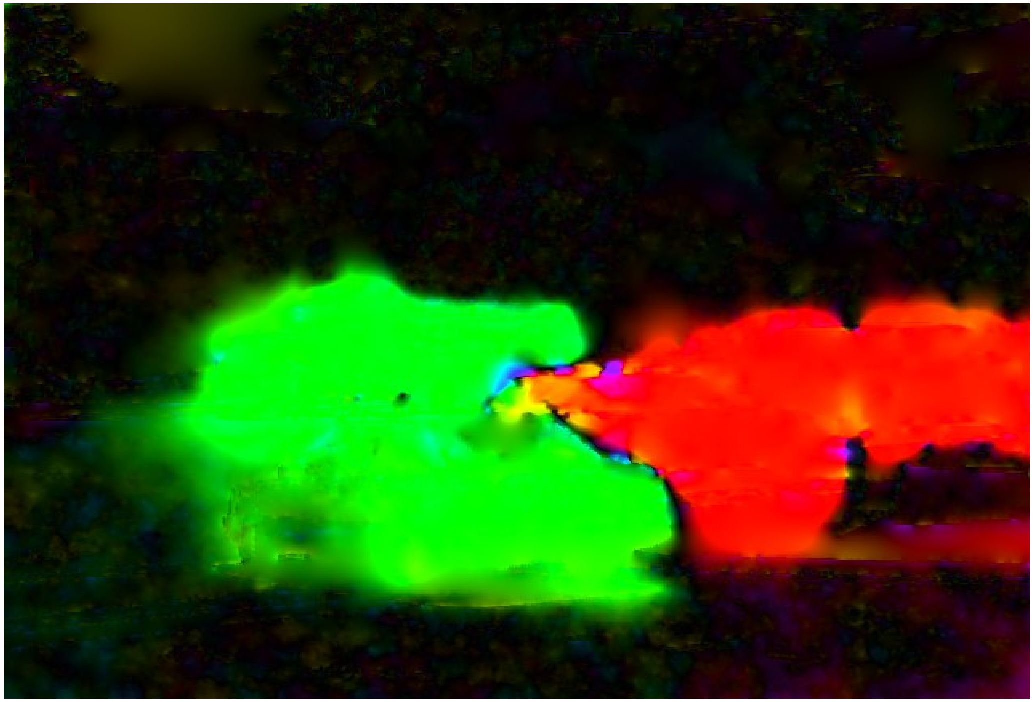
GPU Programming in Computer Vision



Philipp Hübner Stefan Bartels Shrikant Vinchurkar

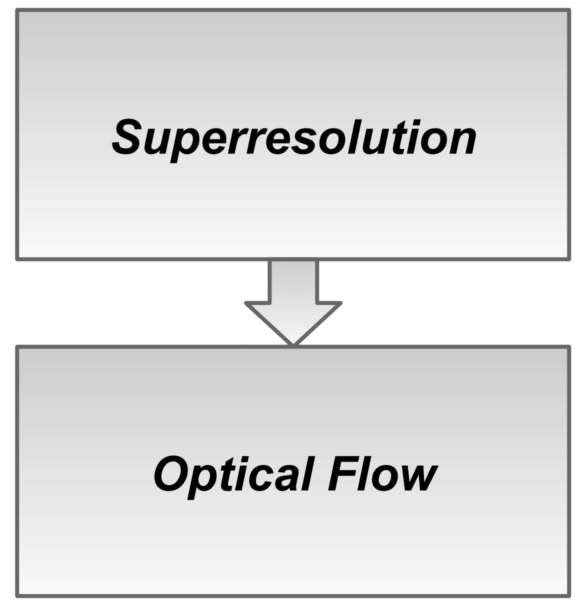
Overview

- Project Overview
- ***Optical Flow***
  - → Problem and Application
  - → Hierarchical Horn and Schunck
  - → Implementation on GPU
  - → Results and Benefits
- Superresolution
  - → Problem and Theory
  - → Implementation on GPU
  - → Results and Benefits
- Summary

Philipp Hübner Stefan Bartels Shrikant Vinchurkar

Project Overview

- ***Optical Flow***
  - → Understanding the theory
  - → Implementing a GPU Version of Hierarchical Horn and Schunck
- ***Superresolution***
  - → Understanding more theory ...
  - → Using results of optical flow
- Benchmarking different kinds of memory



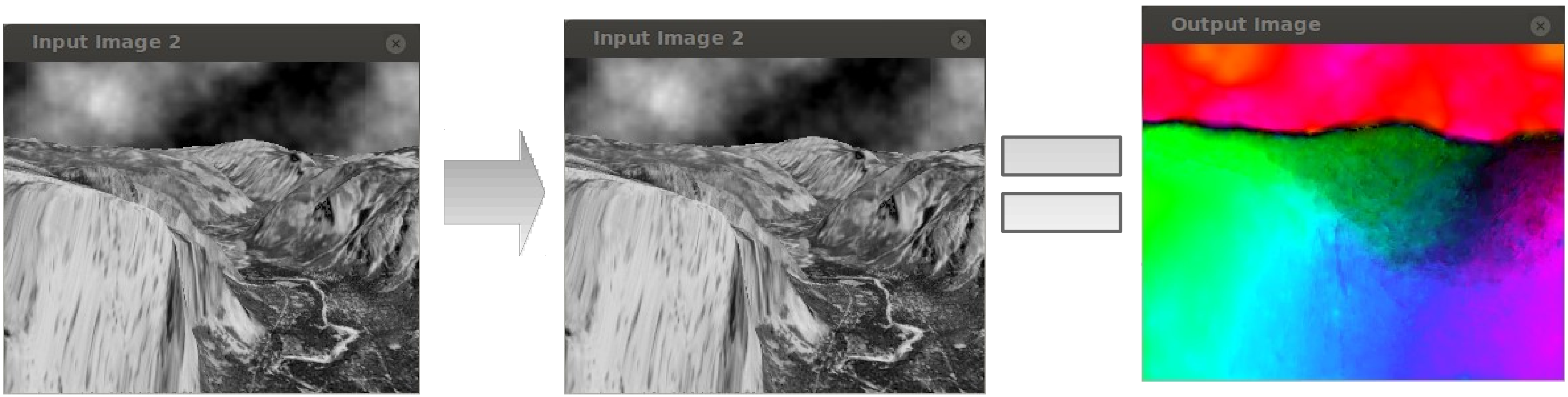
Optical Flow – the Problem

"**Optical flow** or **optic flow** is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer (an eye or a camera) and the scene" (Wikipedia)

Applications:

- Motion Estimation
- Video Compression
- Time to Collison / Collison Avoidance
- Segmentation

Optical Flow



Horn and Schrunk Method

Global approach: *Horn und Schrunk Method*

Goal: Identification of a dense field of optical flux u for each pixel of given images I1 and I2:

$$I_2(\mathbf{x} + u(\mathbf{x})) = I_1(\mathbf{x})$$

Together with the smoothness term we get the energy equation ***E(u)*** to minimize:

$$E(u) = \int_\Omega (I_2(\mathbf{x} + u(\mathbf{x})) - I_1(\mathbf{x}))^2 + \lambda |\nabla u|^2 \quad \mathrm{d}\mathbf{x}$$

with the gradient magnitude:

$$|\nabla u| = \sqrt{{u_1}_x^2 + {u_1}_y^2 + {u_2}_x^2 + {u_2}_y^2}$$

Horn and Schrunk Method

E(u) is highly non-convex: do a Taylor expansion for the I2:

$$I_2(\mathbf{x} + u(\mathbf{x})) = I(\mathbf{x} + u(\mathbf{x}), t + 1) \approx I(\mathbf{x}, t) + \nabla I^\top u + \underbrace{I_t}_{\frac{\mathrm{d}I}{\mathrm{d}t}}$$

and plug it into the energy equation:

$$\begin{aligned} E(u) &= \int_\Omega (I(\mathbf{x}, t) + \nabla I^\top u + I_t - I(\mathbf{x}, t))^2 + \lambda |\nabla u|^2 \quad \mathrm{d}\mathbf{x} \\ &= \int_\Omega (\nabla I^\top u + I_t)^2 + \lambda |\nabla u|^2 \quad \mathrm{d}\mathbf{x} \end{aligned}$$

Horn and Schrunk Method

According to Euler-Lagrange calculus we get:

$$\begin{aligned} 0 &= I_x^2 u_1 + I_x I_y u_2 + I_x I_t - \lambda \Delta u_1 \\ 0 &= I_x I_y u_1 + I_y^2 u_2 + I_y I_t - \lambda \Delta u_2 \end{aligned}$$

A robustification term is integrated into the equation:

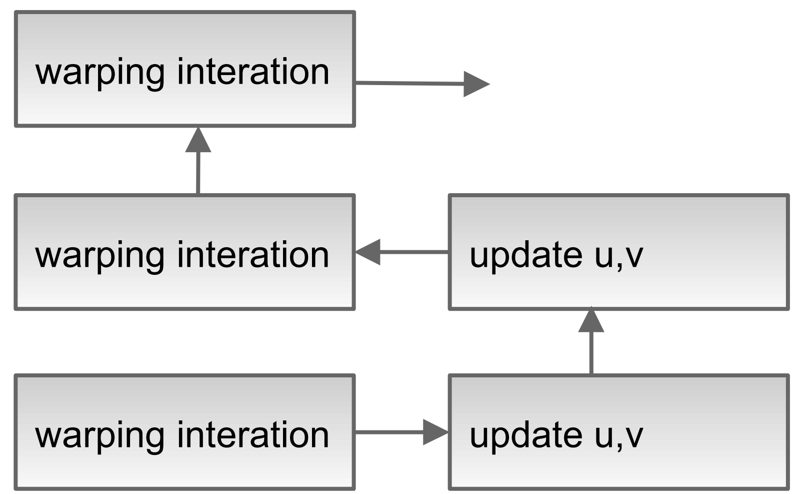
$$E(u) = \int_\Omega \Phi_D((\nabla I^\top u + I_t)^2) + \lambda \Phi_R(|\nabla u|^2) \quad \mathrm{d}\mathbf{x}$$

This brought into LES and solved via SOR in a coarse to fine approach

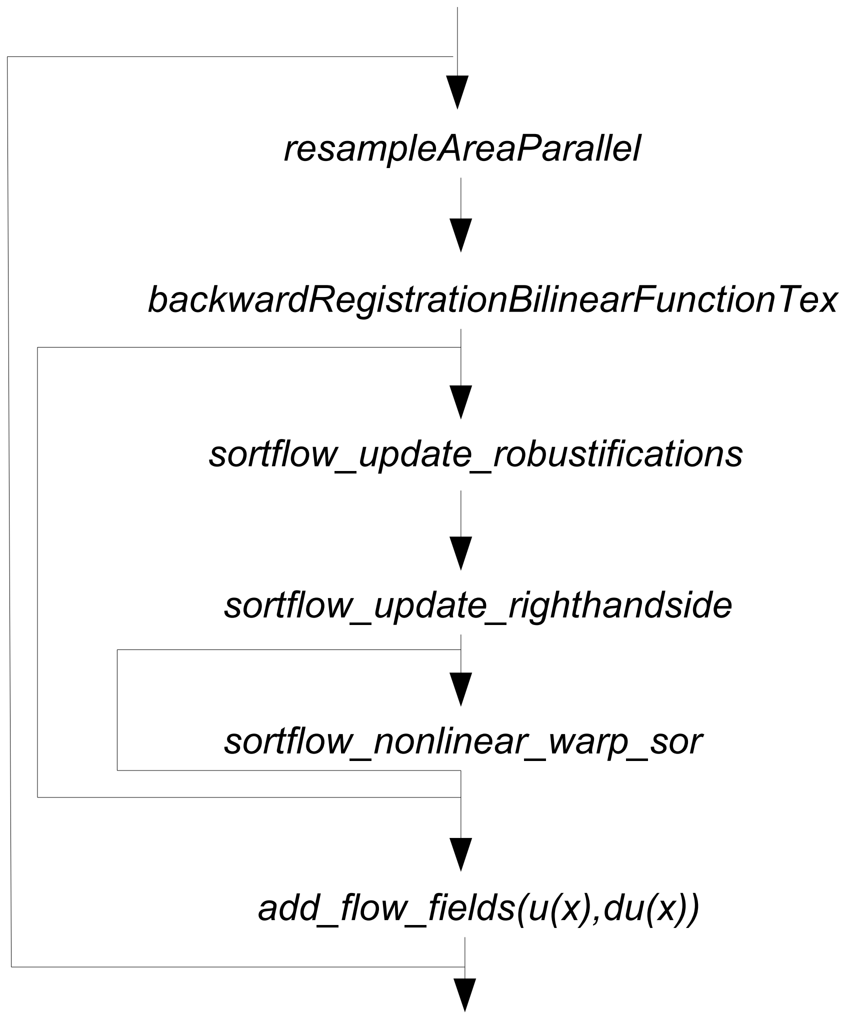
GPU Implementation

The (raw) algorithm:

1. Create the image pyramid (before)
2. Set *u,v* to zero
3. Start with the lowest resolution
4. Resample *u,v* to actual resolution
5. Warp Image with *u,v*
6. Solve Euler-Lagrange scheme for *du, dv*
7. Update *u,v* with *du, dv*
8. Go to higher resolution



GPU Implementation



4. Resample *u,v* to actual resolution

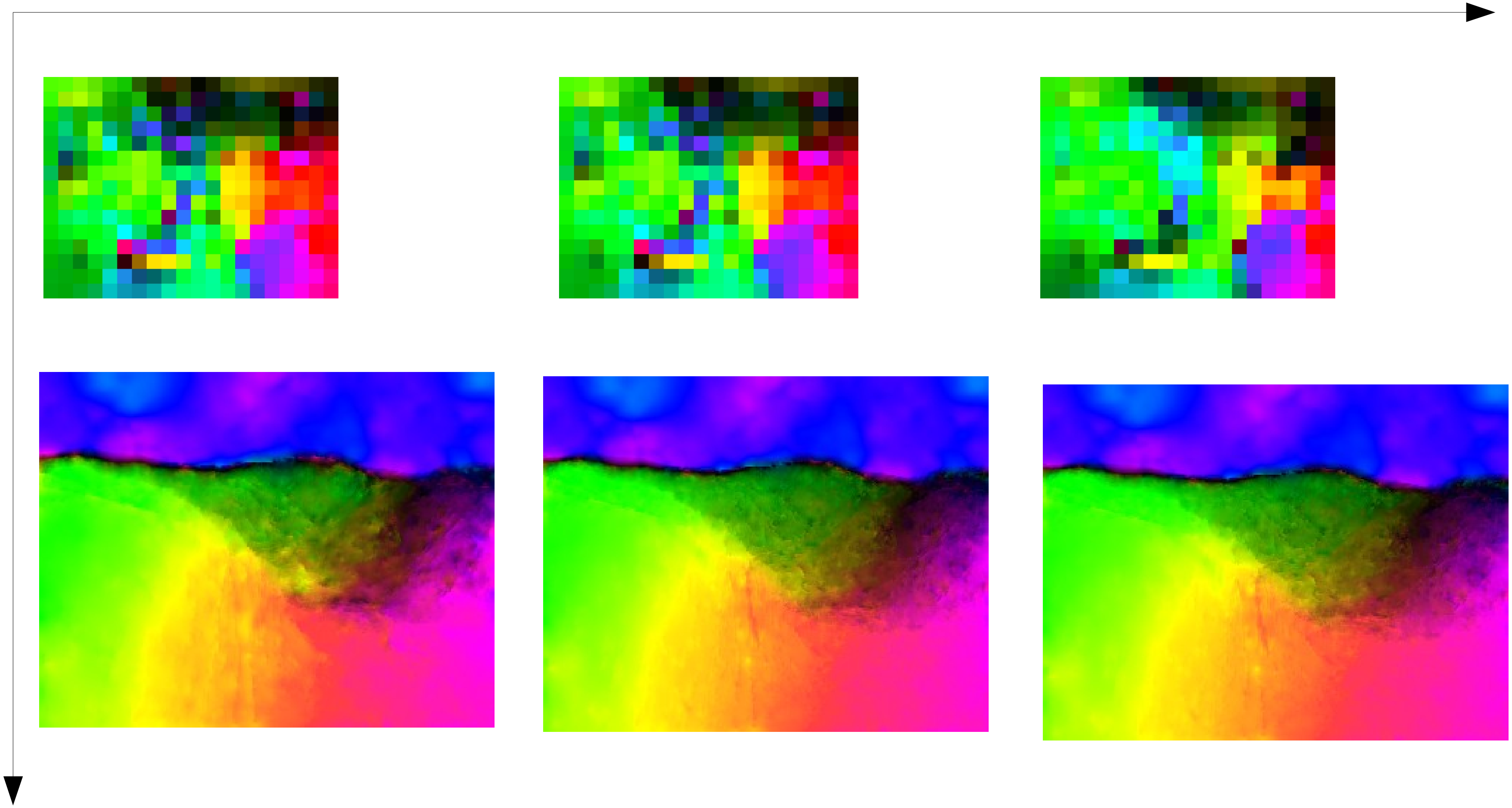
5. Warping image

6. Solving Euler-Lagrange scheme for one level and get *du,dv*

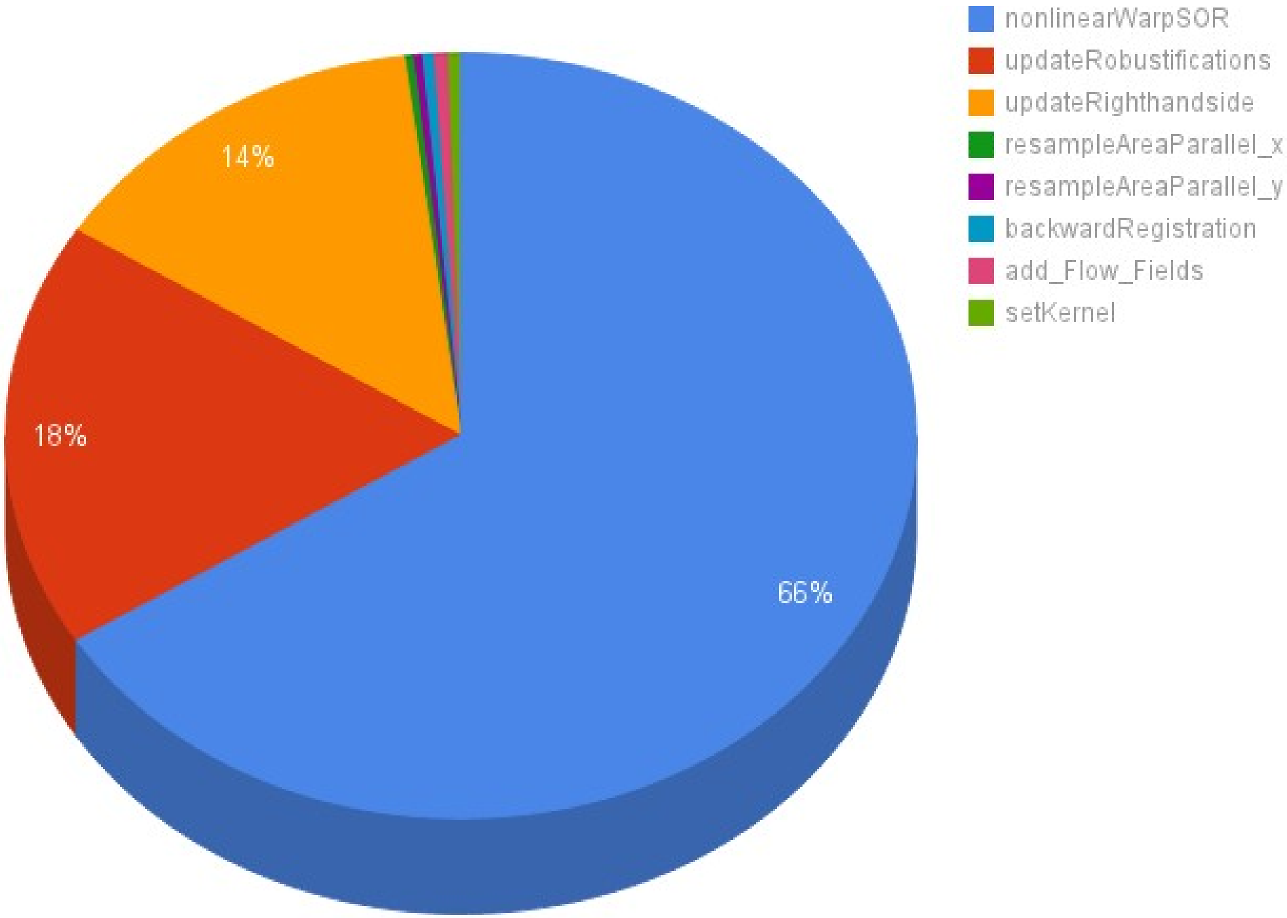
5. Update *u,v*

Flow Results

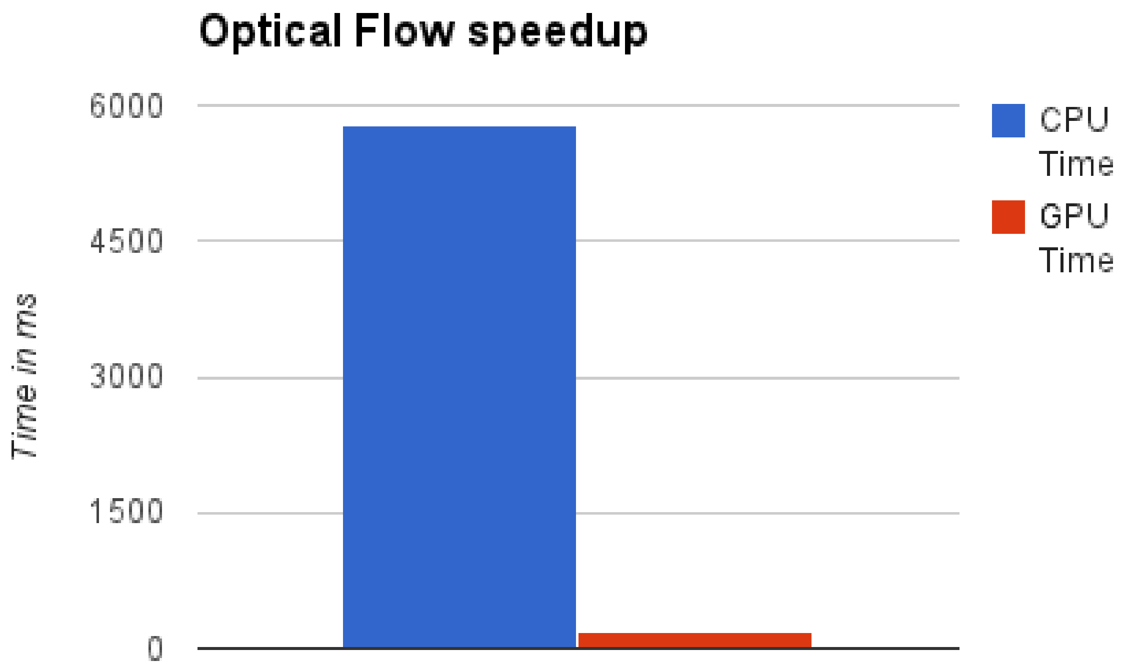
outer iterations



Where does the time go...



Results



average Speedup: 91,45

on a Tesla 1060 with 240 CUDA cores

Overview

- Project Overview
- Optical Flow
  - → Problem and Application
  - → Hierarchical Horn and Schunck
  - → Implementation on GPU
  - → Results and Benefits
- ***Superresolution***
  - → Problem and Theory
  - → Implementation on GPU
  - → Results and Benefits
- Summary

Superresolution

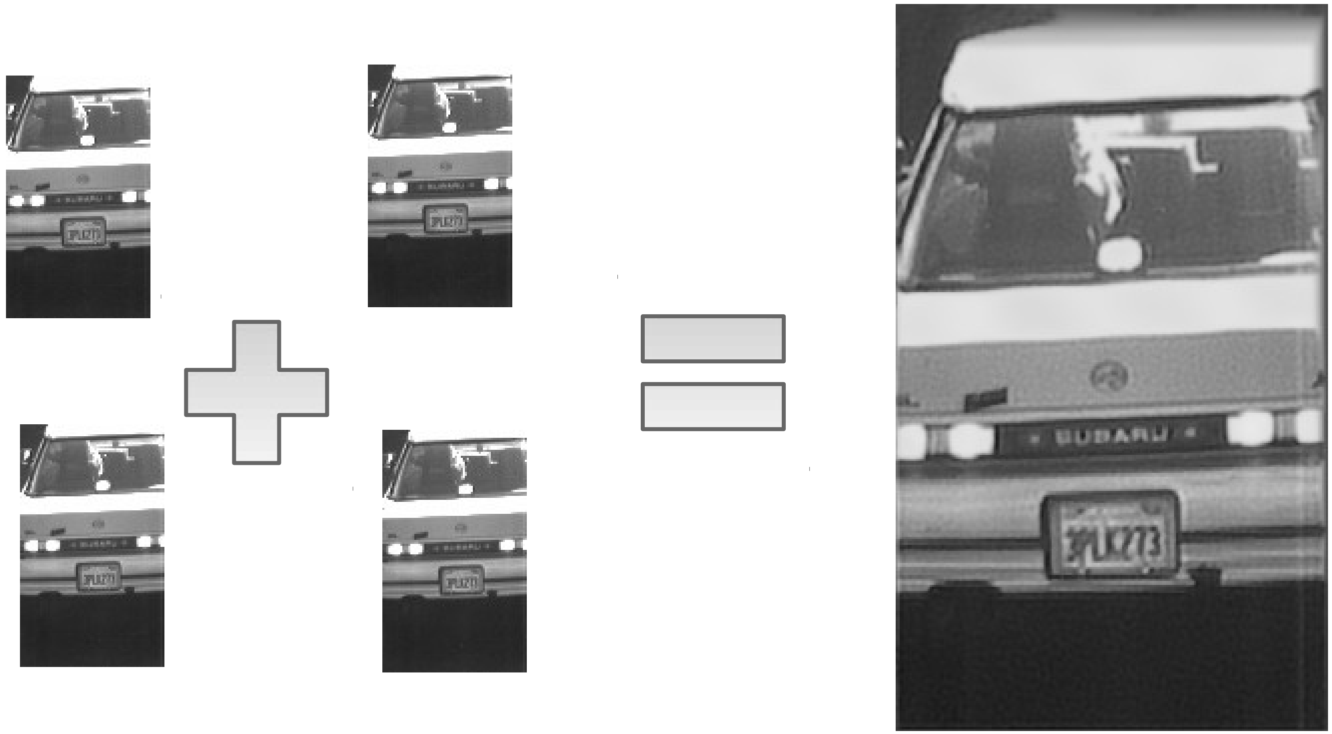
Superresolution is a term for a set of methods of upscaling video or images.

<u>Idea:</u> Generally, information is extracted from several low-res images and combined in a upscaled image.

<u>Drawbacks:</u>

- *Stills - if the object doesn't move, nothing can be extracted*
- *Very fast movements - blur problems*

Superresolution



Superresolution

Image I is degraded by a linear operator F.

Several degraded images, transformed by another linear operator T:

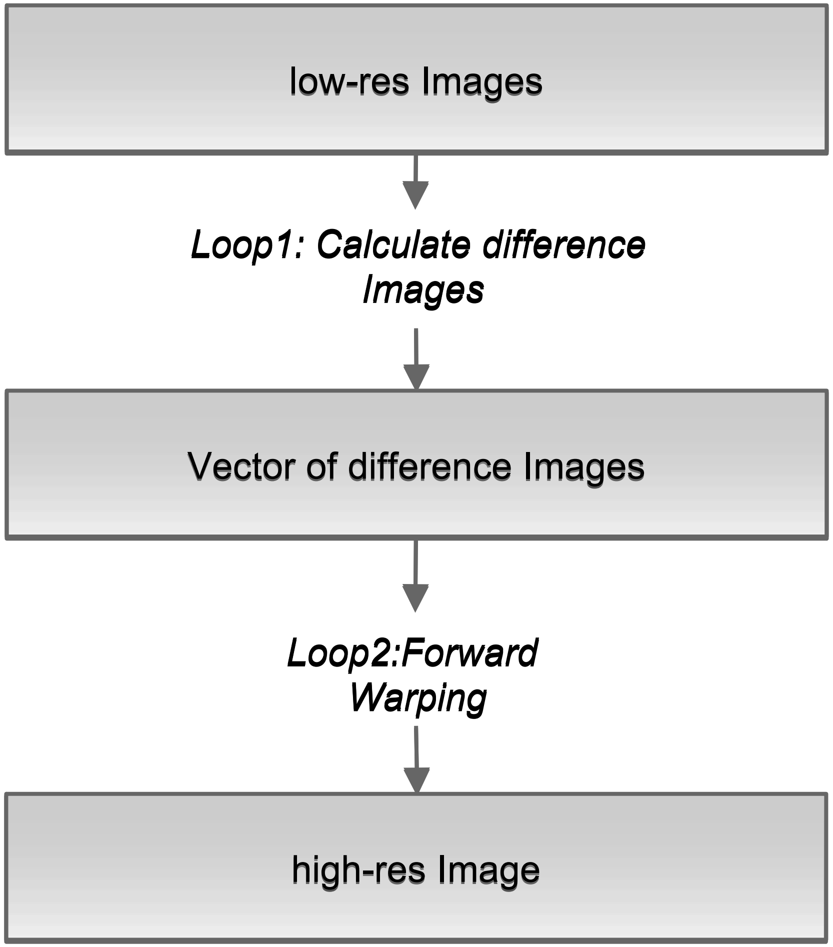
$$\begin{aligned} I_F^1 &= FT^1 I \\ I_F^2 &= FT^2 I \\ &\vdots \\ I_F^n &= FT^n I \end{aligned}$$

This is a large system of linear equations...

Finally, a energy function is fomulated:

$$E(I) = \sum_{i=1}^{n} \mu \left\| FT^i I - I_F^i \right\|_1 + \lambda \left\| \nabla I \right\|_1$$

Superresolution



Superresolution: Loop 1

upscaled input images

u_org (10. iteration)

backwardRegistrationBilinearValueTex

warped u_org

gaussBlurSeperateMirrorGPU (constant)

blurred warped u_org

resampleAreaParallelSeperate

downsampled blurred warped u_org

dualL1Difference

difference image

Superresolution: Loop 2

resampleAreaParallelAjoint

resampled difference images

gaussBlurSeperateMirrorGPU

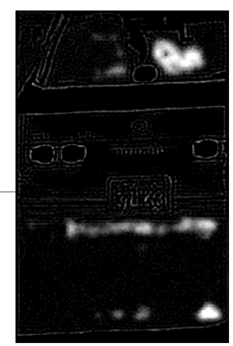
resampled blurred difference images

ForewardRegistrationBilinearAtomic (raceCondition!)

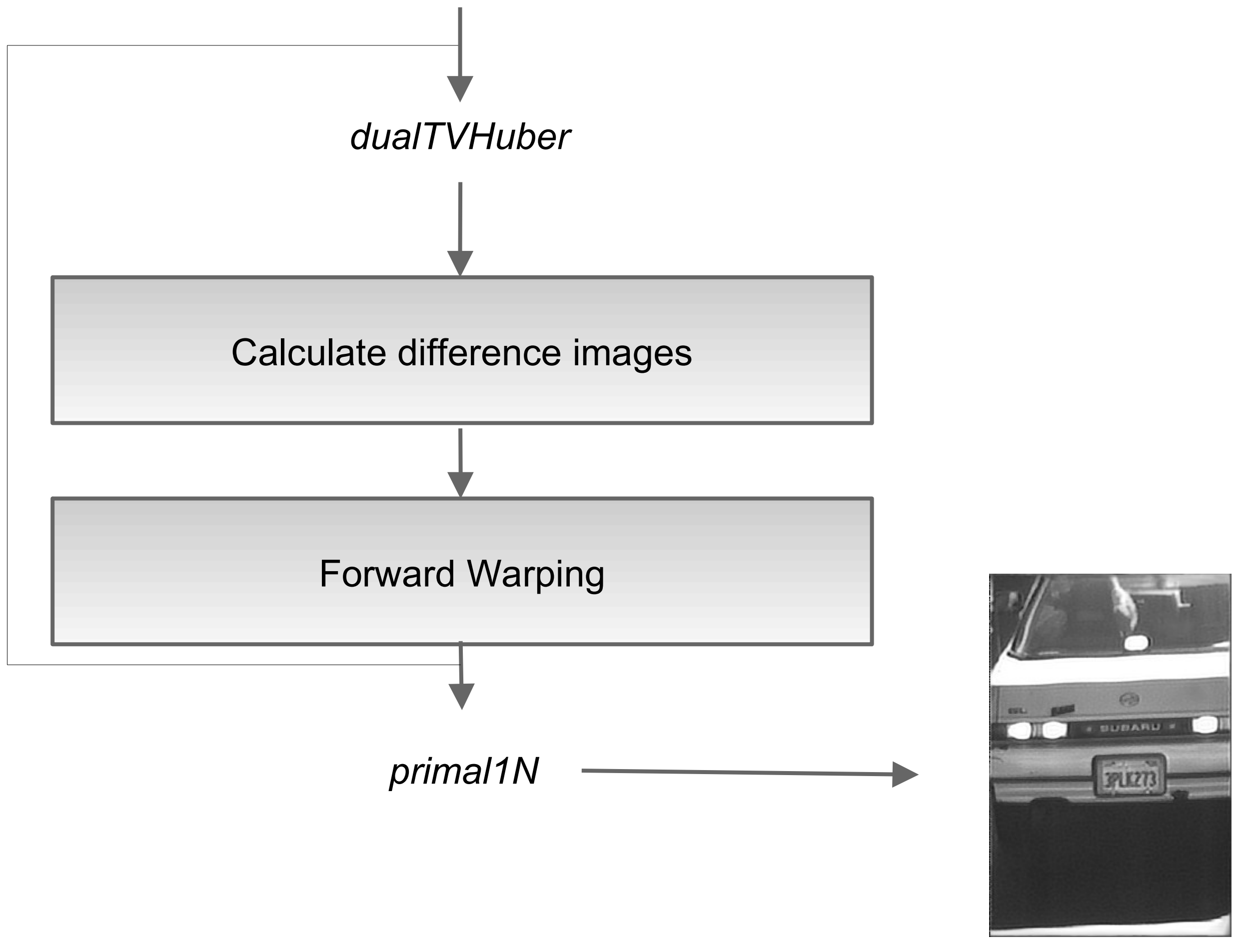
forward warped

addKernel

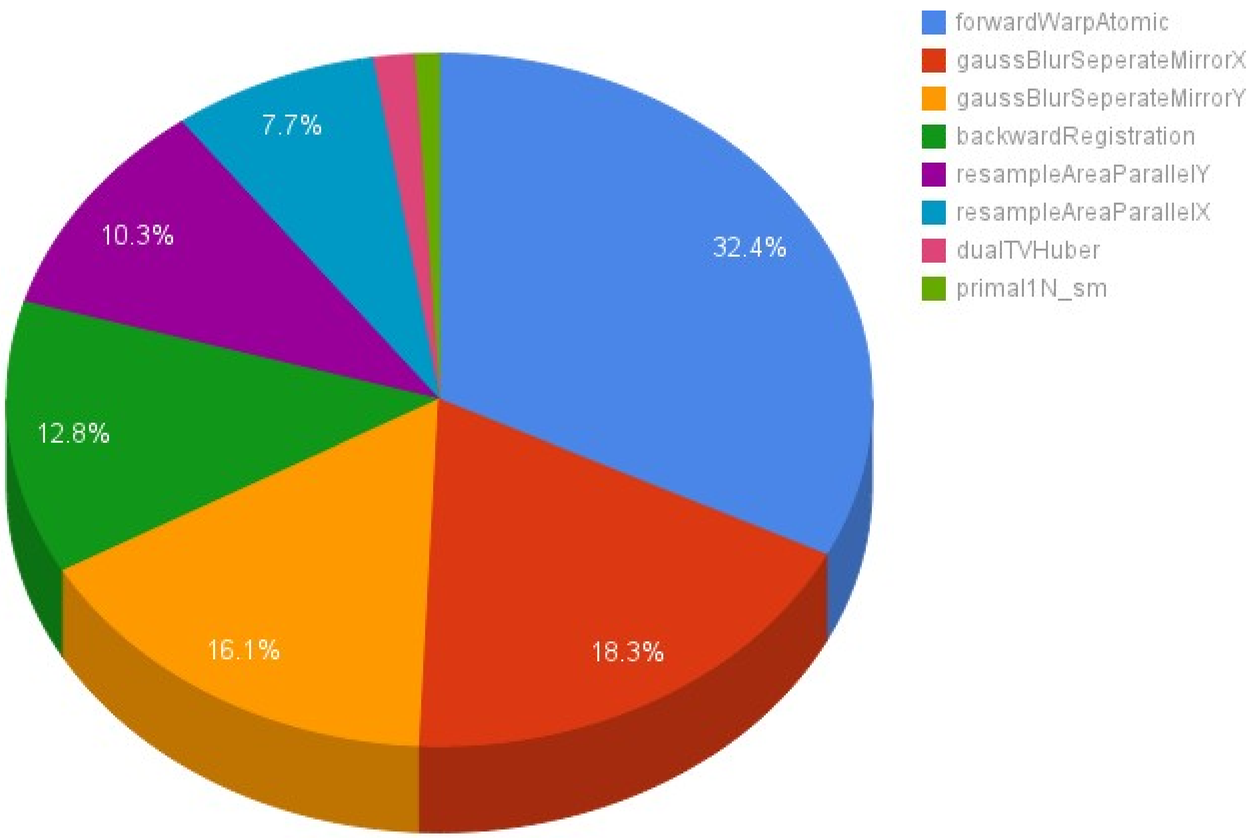
sum of differences



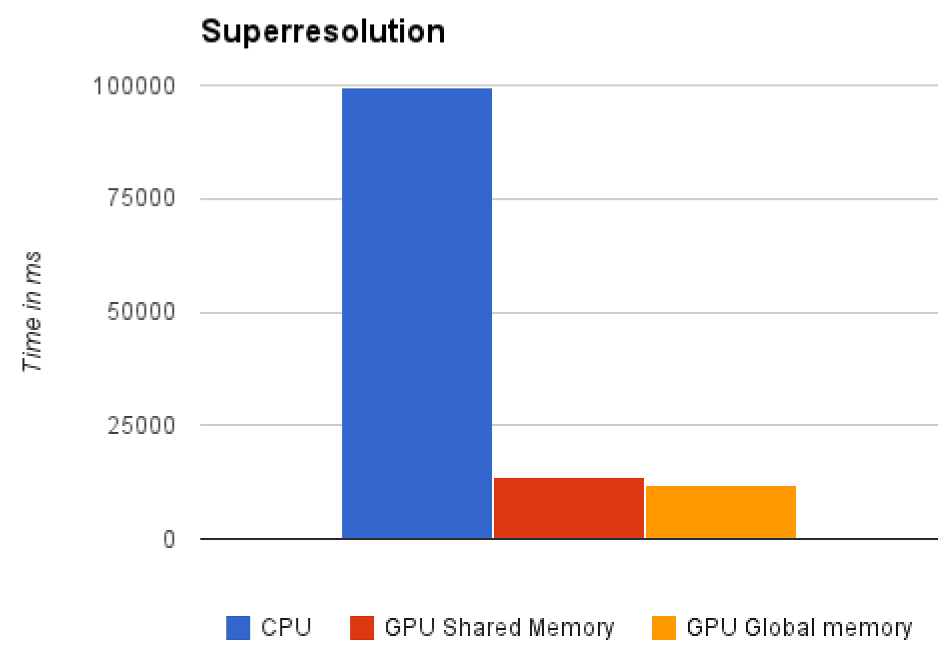
Superresolution



Where does the time go...



Results



average Speedup: 7,45

on a Tesla 1060 with 240 CUDA cores

Memory Experiments

- ***GaussBlur***
  - → constant memory (kernel)
    - → shared
    - → global → **best**
    - → texture → same as global
- ***BackwardWarping***
  - → texture → **best**
  - → global
- ***ForwardWarping***
  - → atomics (race conditons)

Difficulties

- Understanding the algorithms
- Getting local environment run
- Understanding the framework
- Getting debugging posibilties
- How to get intermediate output
- Shared memory failure
- Incorrect working of CUDA printf()

Difficulties

```
float hx = (float)nx_in / (float)nx_out;
float factor = (float)(nx_out)/(float)(nx_in);

resampleAreaParallelSeparate_x<<< dimGrid, dimBlock >>>( in_g, help_g, nx_out, ny_in,
                                  hx, pitchf1_in, pitchf1_out, factor);

// this cost us a lot of time -> resize grid to y_out
gridsize_y = (ny_out % LO_BH) ? ((ny_out / LO_BH)+1) : (ny_out / LO_BH);
dimGrid = dim3( gridsize_x, gridsize_y );

float hy = (float)ny_in / (float)ny_out;
factor = scalefactor*(float)ny_out / (float)ny_in;

resampleAreaParallelSeparate_y<<< dimGrid, dimBlock >>>( help_g, out_g, nx_out, ny_out,
                                  hy, pitchf1_out, factor );
```

Lessons Learned

- Shared memory doesn't always work faster. It depends strictly upon amount of computation you do
- Use of different kind of memories (constant, texture) helps !
- As known, the speedup in the application comes with the cost of extra development efforts

Literature

Markus Unger , Thomas Pock , Manuel Werlberger , Horst Bischof, A convex approach for variational super-resolution, Proceedings of the 32nd DAGM conference on Pattern recognition, September 22-24, 2010, Darmstadt, Germany

Nils Papenberg , Andrés Bruhn , Thomas Brox , Stephan Didas , Joachim Weickert, Highly Accurate Optic Flow Computation with Theoretically Justified Warping, International Journal of Computer Vision, v.67 n.2, p.141-158, April 2006