Speeded-Up Speeded-Up Robust Features

Paul Furgale, Chi Hay Tong, Gaetan Kenway

University of Toronto Institute for Aerospace Studies

April 14^{th},2009

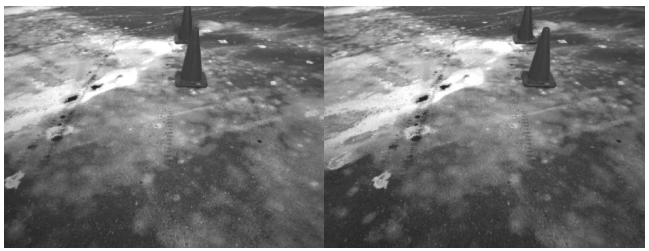
Introduction

- Motivation
- Algorithm description
  - Compute integral image
  - Compute interest point operator
  - Find min/max of the interest point operator
  - Find sub-pixel/sub-scale interest point
  - Compute interest point orientation
  - Compute interest point descriptor
- Results
- Future Work

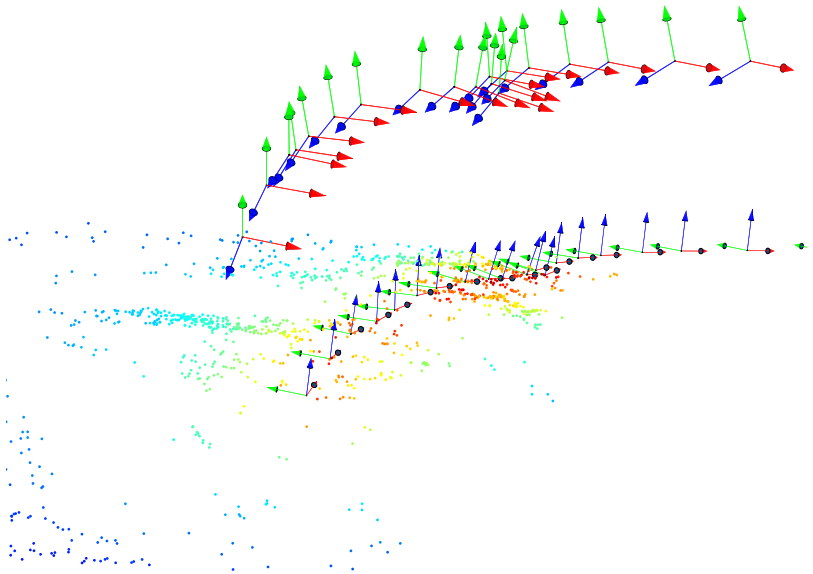
Motivation

Autonomous Robot Navigation.

Get from stereo images

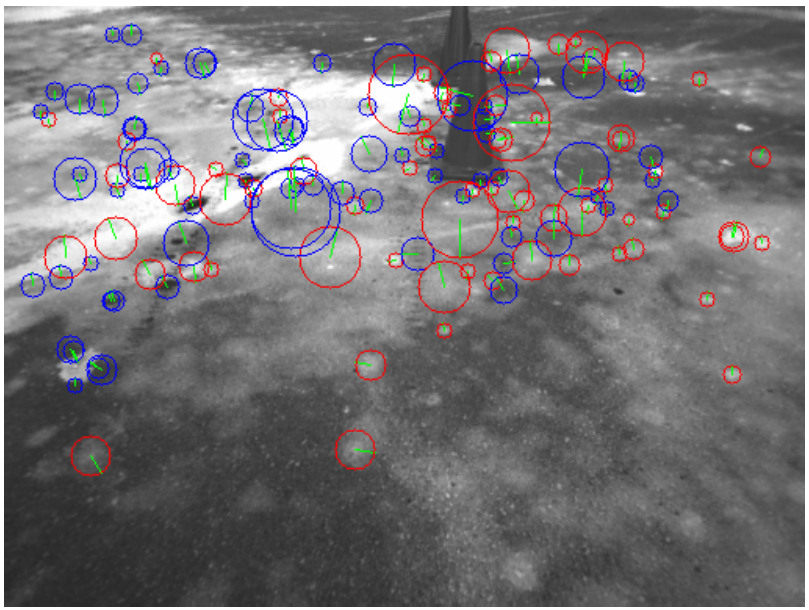


to a 3D motion estimate and terrain model



Algorithm

- Scale Invariant Feature Transform (SIFT) finds scale-invariant keypoints and creates a rotation-invariant descriptor vector (like a fingerprint) to uniquely identify the feature.
- Speeded Up Robust Features (SURF) does the same thing but is much faster as it approximates the operations.



Compute integral image

$$I_{\Sigma}(x,y) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i,j)$$

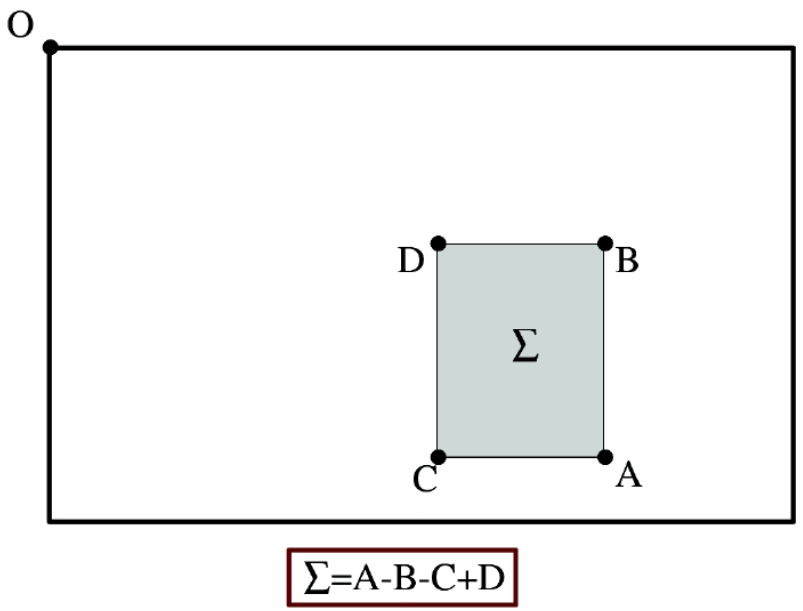


Illustration of an area lookup using an integral image.
Source: Bay et al.

Compute integral image (GPU)

- Transpose and convert the image to normalized floats
- Compute the scans for all rows (columns) of the image (CUDPP library)
- Transpose the column-scanned image back
- Scan all the rows of the column-scanned image

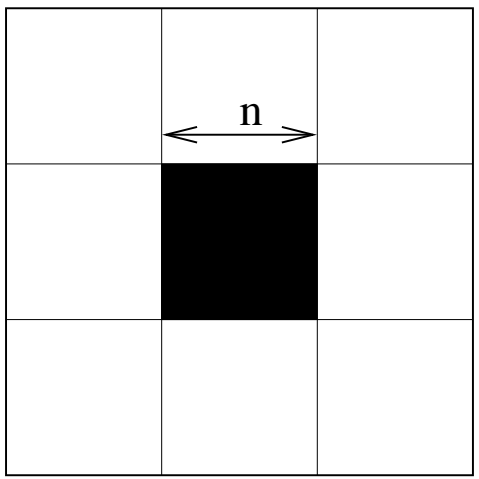
(a) Input test image.

(b) CPU RMS error: 0.0397

(c) GPU RMS error: 0.0066

Compute interest point operator

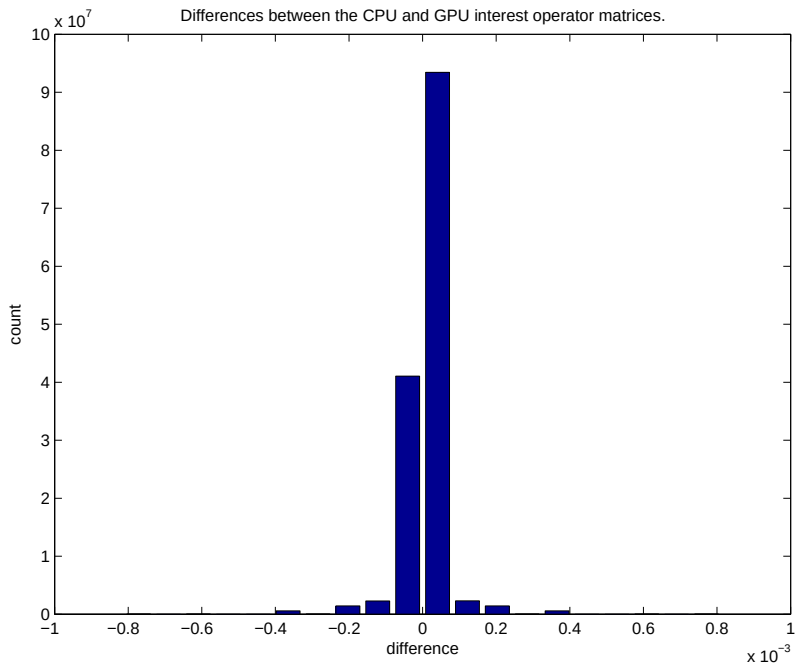
- Goal: Find interest points at different sizes in the image
- Two different kernels employed:
  - Determinant of Hessian operator
  - Center surround extrema
- Tried to reverse engineer SURF fast hessian;
  However, CenSurE kernel produces better results



CenSurE filters

Compute interest point operator (GPU)

- 4 kernel calls for all scales
- Each thread computes CenSurE kernel at unique position (pixel) and size.
- Integral image queried using tex2D lookup



Non-max suppression

- Search all images at all sizes to determine which points are a max/min
  - Point is a maxima/minima if it is greater than/less than all 26 of its neighbors (3x3x3 cube)
- Images are divided into blocks 16x8 pixels, each pixel is assigned to a thread
- Interest point operator values are loaded into shared memory
- Atomic increments are utilized to ensure consistent indexing of maxima/minima
- CPU and GPU implementation produce exactly the same results

Sub-pixel interpolation

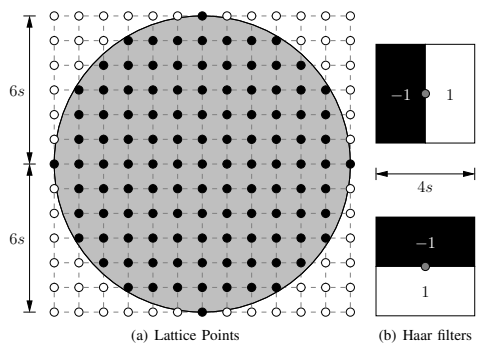
- Once max/min points are found, sub-pixel interpolation is performed using a quadratic approximation
- One block for each interest point is launched
  - 27 threads load nearest neighbors into shared memory
- 1 thread performs actual interpolation

$$L(\mathbf{x}+\Delta\mathbf{x}) = L(\mathbf{x}) + \left(\frac{\partial L}{\partial \mathbf{x}}\right)^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \left(\frac{\partial^2 L}{\partial \mathbf{x}^2}\right) \Delta\mathbf{x}$$

- Solution of 3x3 system of equation is required - computed explicitly
- Possibly more efficient implementation;
  However, not bottleneck of computation

Compute interest point orientation

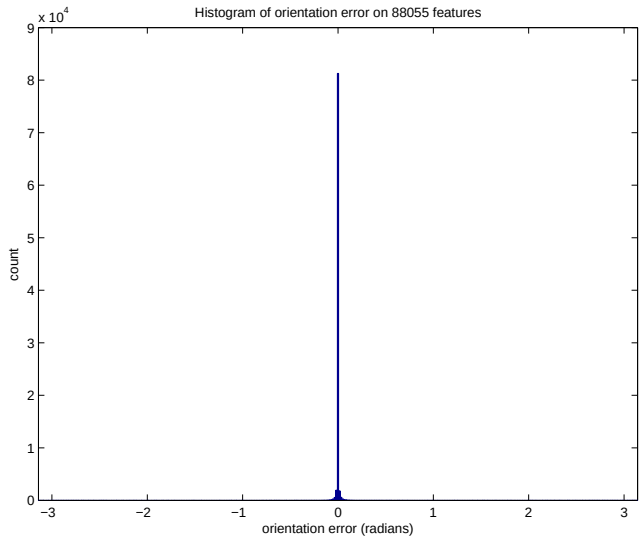
- Goal: Produce a repeatable orientation for the interest point.
- Evaluate d_x, d_y, and $\theta = \text{atan2}(d_y, d_x)$ at each sample point.
- Weight the wavelet values by a Gaussian with $\sigma = 2s$
- Find the largest vector $[\sum d_x, \sum d_y]$ in a sliding orientation window of size $\frac{\pi}{3}$
- That vector defines the orientation $\phi = \text{atan2}(\sum d_y, \sum d_x)$



Sample points for the Haar wavelets.

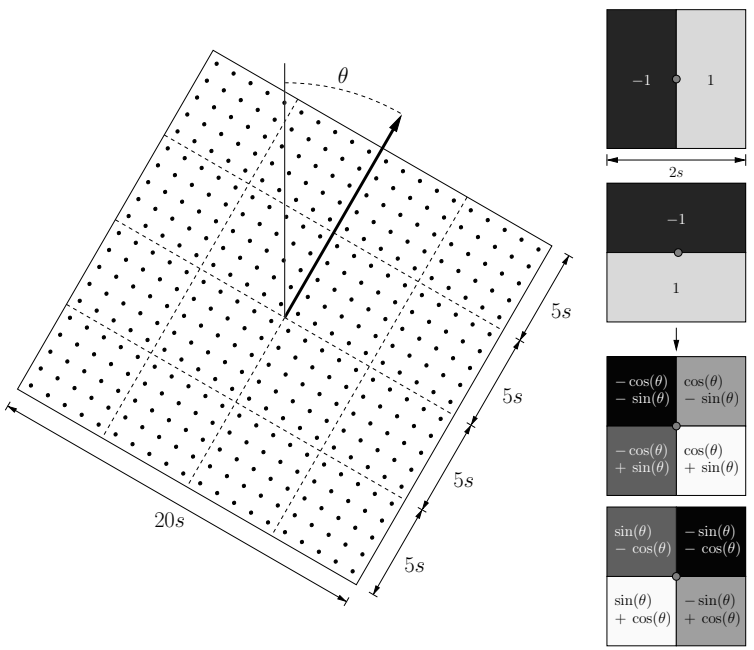
Compute interest point orientation (GPU)

- Minimize memory bandwidth by loading sample points into shared memory
- Parallel, bitonic sort by angle adapted from the sample code.
- Each thread calculates one $\frac{\pi}{3}$ window.
- Reduction to find the maximum vector.



Difference between the GPU and CPU orientation

Compute interest point descriptor



(a) Lattice Points (b) Haar filters

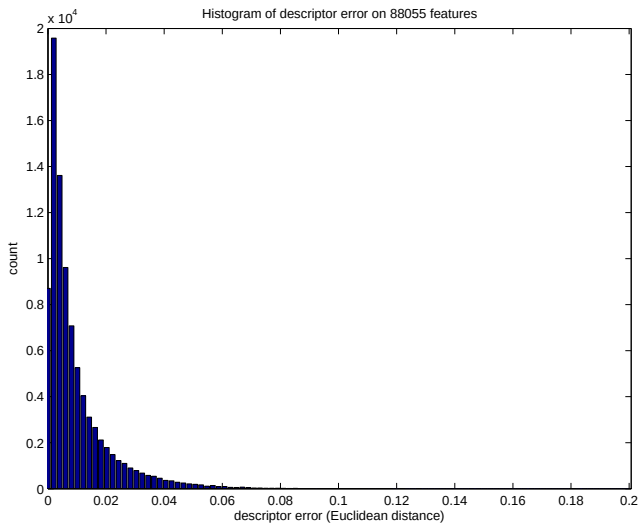
- Overlay a $20s \times 20s$ square region
- Subdivide equally region into 4×4 square subregions
- Compute orientation-aligned Haar wavelet responses (filter size $2s$) at 5×5 regularly spaced sample points
- Weight by Gaussian ($\sigma = 3.3s$), sum to form descriptor vector for each subregion:
  $\mathbf{v} = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$
- Concatenate descriptor vectors to form a 64-dimensional vector
- Normalize to form a unit vector

Compute interest point descriptor (GPU)

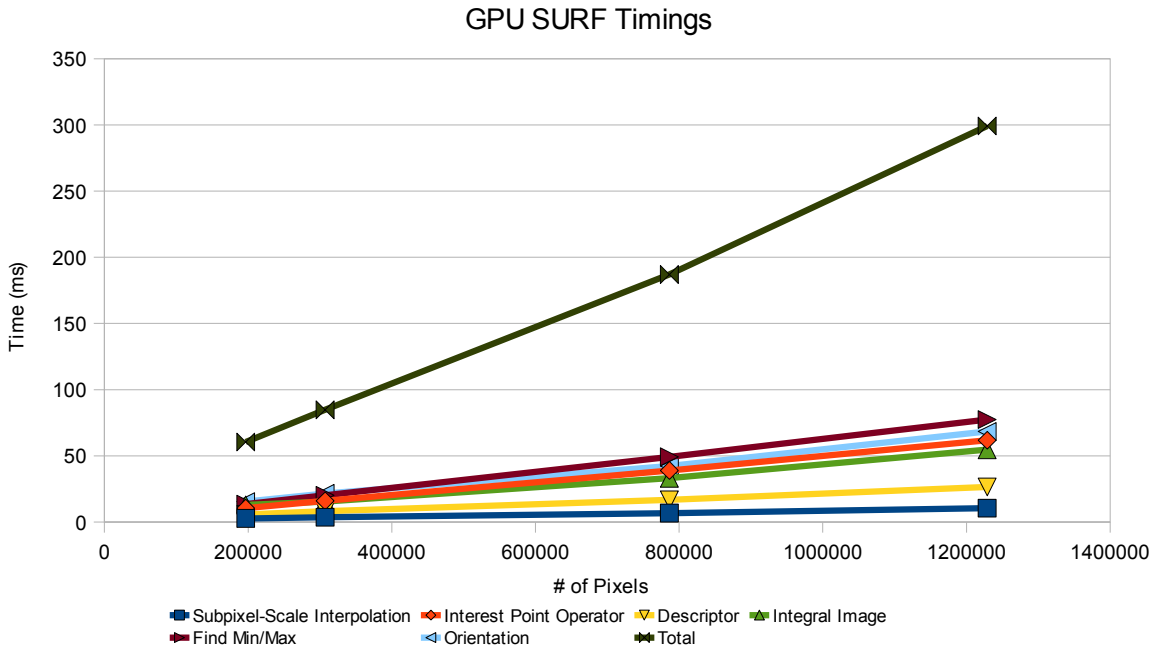
- Two kernel calls (less shared memory requirements)
- Compute unnormalized descriptors
  - 16 blocks per interest point, 25 threads per block
  - Load interest point parameters (x, y, s, ϕ)
  - Compute trigonometric rotations $(\sin(\phi), \cos(\phi))$
  - Compute sample point locations
  - Load integral image lookups (9)
  - Compute axis-aligned Haar responses (d_x, d_y)
  - Rotate and store the Haar responses
  - Load absolute values $|d_x|$ into another memory block
  - Sum the $d_x, |d_x|$ responses (reduction)
  - Write back unnormalized responses to global memory
  - Repeat for $d_y, |d_y|$

Compute interest point descriptor (GPU)

- Normalize descriptor to form a unit vector
  - 1 block per interest point, 64 threads per block
  - Load and square values of the descriptor
  - Sum (reduce) the squared values
  - Compute the square root of the sum (length)
  - Divide each component by the length and write back to global memory



Timing Results



- Time contribution of each functional block is nearly independent of image size
- Very near linear scaling with image size

Timing Results

Image Size	512x384	640x480	1024x768	1280x1024
Integral Image	35.1%	29.9%	27.3%	28.2%
Interest Point Operator	28.4%	31.0%	32.1%	31.9%
Non-max Supression	36.5%	39.1%	40.6%	39.9%
Interp Extremum	11.4%	10.9%	10.2%	9.9%
Orientation	64.9%	64.7%	64.4%	65.0%
Descriptor	23.7%	24.4%	25.5%	25.1%

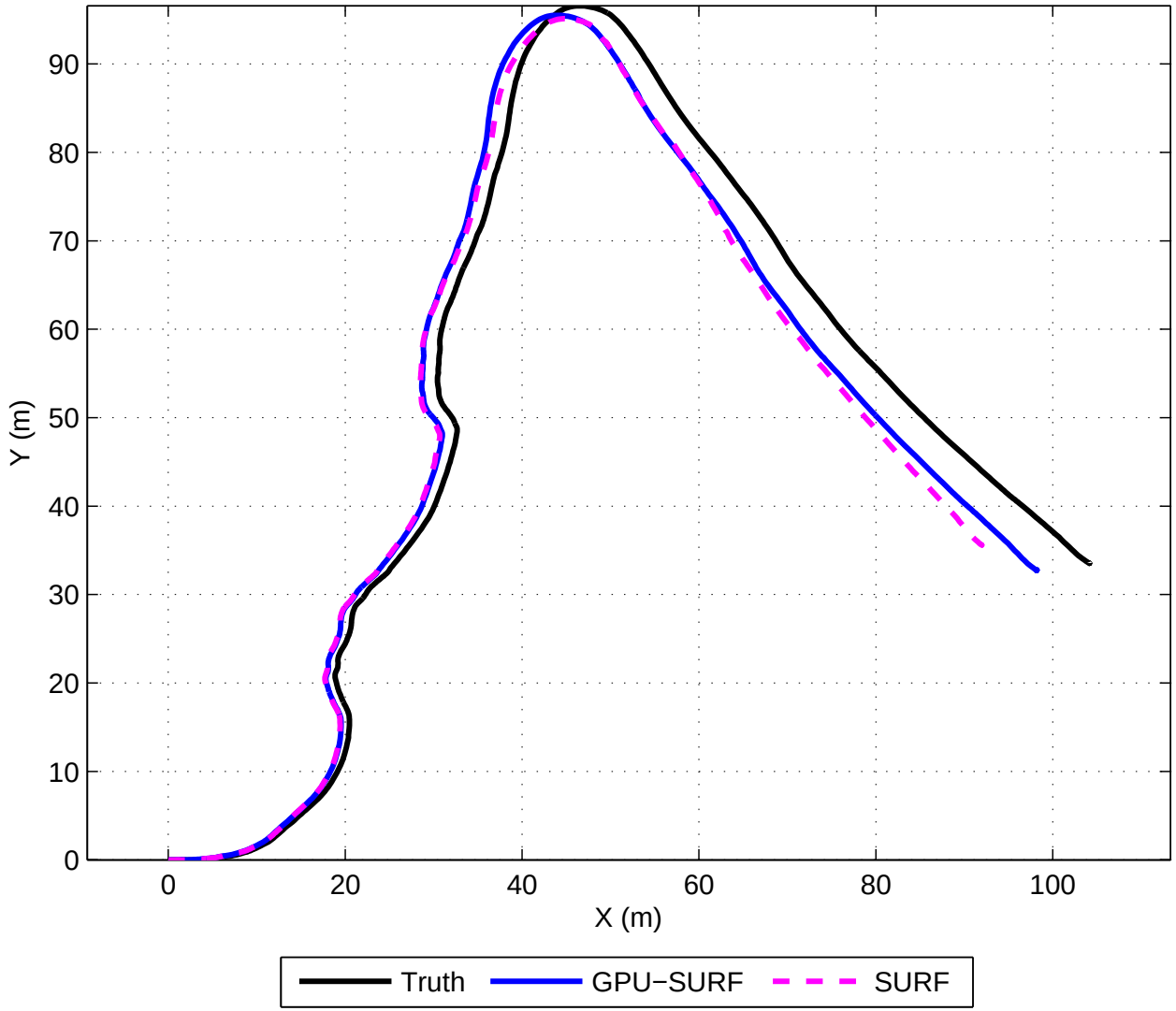
Functional block timings split by 1) independant of feature count (top) 2) dependent on feature count (bottom)

Speedup Profile

- Comparing GPU implementation vs. OpenSURF and modified OpenSURF
  - Average over 10 trials for CPU, 1000 trials for GPU
  - Does not include memory initializations
  - Currently using synchronous memory transfers

Image Size	512x384	640x480	1024x768	1280x1024
Feature Count	957	1509	3032	3218
CPU-SURF	961.86ms	1461.65ms	3409.69ms	4153.78ms
OpenSURF	251.27ms	384.95ms	998.29ms	1276.63ms
GPU-SURF	6.75ms	10.83ms	21.10ms	28.00ms
Modified OpenSURF Percent Speedup	14250%	13496%	16160%	14835%
OpenSURF Percent Speedup	3722%	3554%	4731%	4559%

Results



SURF: 3.73698 Hz GPU-SURF: 9.17844 Hz

Future Work

- Compute features for both images of a stereo pair and match the features on the GPU.
- Copy the features back to the CPU and track them (in time) while the next pair is being matched.
- Run on our robot.

Acknowledgements

- Dr. Moshovos
- CUDA Programming Guide
- Emacs

Questions