Speeded-Up Robust Features

Paul Furgale, Chi Hay Tong, Gaetan Kenway

University of Toronto Institute for Aerospace Studies

April 14th, 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.
Algorithm

Compute integral image

\[ l_\Sigma(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{y} l(i,j) \]

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

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
Algorithm

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

\[
\sigma \left( x, y \right) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-\frac{1}{2} \left( \frac{x^2 + y^2}{\sigma^2} \right)} \, dx \, dy
\]

CenSurE filters
Algorithm

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

![Differences between the CPU and GPU interest operator matrices.](chart.png)
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
Algorithm

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.

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

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

**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

Histogram of orientation error on 88055 features
Algorithm

**Compute interest point descriptor**

- Overlay a 20s × 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 (σ = 3.3s), sum to form descriptor vector for each subregion:
  \[
  \mathbf{v} = \left( \sum d_x, \sum d_y, \sum |d_x|, \sum |d_y| \right)
  \]
- Concatenate descriptor vectors to form a 64-dimensional vector
- Normalize to form a unit vector
Algorithm

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 \((x, y, s, \phi)\)
  - 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 \(d_y, |d_y|\)
Algorithm

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
Results

Timing Results

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

### Timing Results

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

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

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

<table>
<thead>
<tr>
<th>Image Size</th>
<th>512x384</th>
<th>640x480</th>
<th>1024x768</th>
<th>1280x1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Count</td>
<td>957</td>
<td>1509</td>
<td>3032</td>
<td>3218</td>
</tr>
<tr>
<td>CPU-SURF</td>
<td>961.86ms</td>
<td>1461.65ms</td>
<td>3409.69ms</td>
<td>4153.78ms</td>
</tr>
<tr>
<td>OpenSURF</td>
<td>251.27ms</td>
<td>384.95ms</td>
<td>998.29ms</td>
<td>1276.63ms</td>
</tr>
<tr>
<td>GPU-SURF</td>
<td>6.75ms</td>
<td>10.83ms</td>
<td>21.10ms</td>
<td>28.00ms</td>
</tr>
</tbody>
</table>

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