Fast High-Quality Panorama Stitching

GTC 2010, San Jose | James Fung & Timo Stich, NVIDIA
GPU Panorama Stitching

Example imaging pipeline

Panorama generated from three 3840x2880 (10MP) images completed in 0.577s on a GTX280 GPU
GPU Panorama Stitching
Image Pipeline: Panorama Stitching

Left Image

Right Image

Radial Distortion Correction

Keypoint Detection & Extraction (Shi-Tomasi/SIFT)

Keypoint Matching

Recover Homography (RANSAC)

Projective Transform

Image Stitching
Radial Distortion Removal

\[ \delta x = x (\kappa_1 r^2 + \kappa_2 r^4 + \ldots) \]
\[ \delta y = y (\kappa_1 r^2 + \kappa_2 r^4 + \ldots) \]

Images taken with a Nikon D70 18-70mm NIKKOR Lens
Radial Distortion Removal

Point Samples

Bilinearly Interpolation (hw)

Bicubic Interpolation
Radial Distortion Removal

- Linear Interpolation is “Free”!
- Apply hardware linear interpolation to approximate higher order interpolation (see Simon Green’s “Bicubic” SDK example)
- Excellent Texture Cache behaviour

<table>
<thead>
<tr>
<th>Interpolation Method</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quadro 570m</td>
</tr>
<tr>
<td></td>
<td>4 SMs</td>
</tr>
<tr>
<td>Nearest Neighbor</td>
<td>70.99 ms</td>
</tr>
<tr>
<td>Linear Interpolation</td>
<td>71.06 ms</td>
</tr>
<tr>
<td>Bicubic (4 samples)</td>
<td>107.26 ms</td>
</tr>
</tbody>
</table>

Input Image: 3008x2000 (6MP) RGB
Image Pipeline: Panorama Stitching

Left Image

Right Image

Radial Distortion Correction → Keypoint Detection & Extraction (Shi-Tomasi/SIFT) → Keypoint Matching → Recover Homography (RANSAC) → Projective Transform → Image Stitching
Corner Detection

$$A = \sum_{u} \sum_{v} w(u,v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Compute matrix $A$, in region $(u,v)$ around a point $(x,y)$, of Gaussian weighted $(w(u,v))$ image derivatives $(I_x, I_y)$

Harris: Compute $M_c$, (based on Eigenvalues of $A$, $\lambda_1$, and $\lambda_2$) by computing the determinant and trace of $A$

$$M_c = \lambda_1 \lambda_2 - \kappa (\lambda_1 + \lambda_2)^2 = \text{det}(A) - \kappa \text{trace}^2(A)$$

Shi-Tomasi: Compute Eigenvalues, $\lambda_1$, and $\lambda_2$ and threshold on $\min(\lambda_1, \lambda_2)$

$$\lambda_1, \lambda_2 = \frac{\text{tr}(A) \pm \sqrt{\text{tr}(A)^2 - 4 \text{det}(A)}}{2}$$
Feature Detection

Shi-Tomasi + Non-maximal Suppression
Corner Detection @ 1024x768
GPUs vs Intel E5440 CPU

- GPU C1060 (2.3 ms)
- GPU Quadro 570m (12.4 ms)
- CPU (47.2 ms)

Time (ms)

- Compute Derivatives & Eigenvalues
- Compact Eigenvalues
- Non-maximal Suppression
- Generate Points
Corner Detection
Corner Detection

- No single threshold can eliminate clutter and maintain weaker features

Indistinct
Corner Detection

- No single threshold can eliminate clutter and maintain weaker features

Indistinct cluttered features

Loss of salient points
Modified Shi-Tomasi

- Take ratio of \( \min(\lambda_1, \lambda_2) \) to its neighbourhood
- Reduces clutter, maintains distinctive (though weaker) features
Corner Detection: Dynamic Method

\[
A = \sum_u \sum_v w(u, v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}
\]

Compute matrix \( A \), in region \((u,v)\) around a point \((x,y)\), of Gaussian weighted \((w(u,v))\) image derivatives \((I_x, I_y)\)

Dynamic Thresholding: Compute Eigenvalues

\[
\lambda_1, \lambda_2 = \frac{\text{tr}(A) \pm \sqrt{\text{tr}(A)^2 - 4 \det(A)}}{2}
\]

Dynamic Threshold: Compare \(\min(\lambda_1, \lambda_2)\) to regional \((u,v)\) minimum

\[
M_r = \frac{\min(\lambda_1, \lambda_2)}{\sum_u \sum_v \min(\lambda_1, \lambda_2)}
\]

Additional Computational Cost: One 2D convolution and a division/comparison
Dynamic Feature Detection

Dynamic Thresholding Corner Detection @ 1024x768

GPUs vs Intel E5440 CPU

- CPU (56.2 ms)
- GPU Quadro 570m (15.4 ms)
- GPU C1060 (2.7 ms)

Time (ms)

- Compute Derivatives & Eigenvalues
- Compact Eigenvalues
- Non-maximal Suppression
- Generate Points
- Dynamic Threshold
Pixels to Points: *HistoPyramids*

- How to go from pixels to (x,y) point coordinates, on the GPU?
GPU HistoPyramids

- Based on papers by Ziegler et al. (NVIDIA)
- Able to generate a list of feature coordinates completely on the GPU
- Determines:
  - What is the location of each point?
  - How many points are found?
- Applicable for quadtree data structures

See [http://www.mpii.de/~gziegler](http://www.mpii.de/~gziegler) for full information
HistoPyramids

• How do we generate a list of points on the GPU from an image buffer containing 1’s (points) and 0’s (non-points)

\[
\begin{array}{ccccccc}
1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

• Do a reduction: each level is the sum of 2x2 region “below” it
• The top level is the number of points total
HistoPyramids

- How do we generate a list of points on the GPU from an image buffer containing 1’s (points) and 0’s (non-points)

- Do a reduction: each level is the sum of 2x2 region “below” it
- The top level is the number of points total
HistoPyramids

• How do we generate a list of points on the GPU from an image buffer containing 1’s (points) and 0’s (non-points)

• Do a reduction: each level is the sum of 2x2 region “below” it
• The top level is the number of points total
**HistoPyramids**

- How do we generate a list of points on the GPU from an image buffer containing 1’s (points) and 0’s (non-points)

```
1 0 1 0 1 0 1 0  
0 1 0 0 0 1 0 0  
0 0 0 0 0 0 0 0  
0 0 1 0 0 0 1 1  
0 1 0 0 0 0 1 1  
0 0 0 1 0 0 1 1  
0 0 0 0 0 0 0 1  
0 0 0 0 0 0 0 1  
```

```
2 1 2 1 4 5 17  
0 1 0 2 2 6  
1 1 0 4  
0 0 0 2  
17
```
**HistoPyramid**

- The pyramid is now a *map* to where the point locations are
- Traverse down the pyramid, counting past points, and populate the list
- Example: at what coordinates is the 5\textsuperscript{th} point in the list

```
  1 0 1 0 1 0 1 0 2 1 2 1 4 5 17
  0 1 0 0 0 1 0 0 0 1 0 2 2 6
  0 0 0 0 0 0 0 0 1 1 0 4
  0 0 1 0 0 0 1 1 0 0 0 2
  0 1 0 0 0 0 1 1
  0 0 0 1 0 0 1 1
  0 0 0 0 0 0 0 1
  0 0 0 0 0 0 0 1
```

- **Holds points 1..4**
- **Holds points 5..9:** Point 5 must be inside (below) this quadrant
- **Holds Points 5,6:** Point 5 must be inside this quadrant
- **Time:** (C1060 GPU)
  - 1024x1024: 0.27 ms
  - 4096x4096: 2.1 ms
  - 8192x8192: 7.7 ms

This must be point 5
Image Pipeline: Panorama Stitching

- Radial Distortion Correction
- Keypoint Detection & Extraction (Shi-Tomasi/SIFT)
- Keypoint Matching
- Recover Homography (RANSAC)
- Projective Transform
- Image Stitching
Generating Descriptors

• What’s a Feature Descriptor?
  – Distinct numerical representation of an image point for matching

• Our example: SIFT
  – “Scale Invariant Feature Transform”
  – 128 element floating point vector ("key")
  – Nearest Euclidean distance between keys is the best match
Generating Descriptors

• Sparse point processing
• HistoPyramid tree organization gives good spatial locality!
• Parallel processing of single descriptor with thread cooperation
  – Shared Memory
  – Thread Synchronization
1. Calculate feature orientation (shared memory reduction)
2. Lookup rotated samples (texture cache)

- Made possible by Thread Cooperation and data dependent array indexing in compute shaders
- Good texture cache usage, constant cache (Gaussian weights)
Feature Descriptor Computation

1. Calculate feature orientation (shared memory reduction)
2. Lookup rotated samples (texture cache)
3. Generate local orientation histograms (one thread per histogram, shared memory, pointer indexing)

• Made possible by Thread Cooperation and data dependent array indexing in compute shaders
• Good texture cache usage, constant cache (Gaussian weights)
Feature Descriptor Computation

1. Calculate feature orientation *(shared memory reduction)*
2. Lookup rotated samples (texture cache)
3. Generate local orientation histograms (one thread per histogram, shared memory, pointer indexing)
4. Normalize Histogram *(shared memory reduction)*
5. Threshold
6. Re-normalize Histogram *(shared memory reduction)*

- Made possible by Thread Cooperation and data dependent array indexing in compute shaders
- Good texture cache usage, constant cache (Gaussian weights)
Matching Results
Image Pipeline: Panorama Stitching

1. **Radial Distortion Correction**
2. **Keypoint Detection & Extraction (Shi-Tomasi/SIFT)**
3. **Keypoint Matching**
4. **Recover Homography (RANSAC)**
5. **Projective Transform**
6. **Image Stitching**
Blending can cause artifacts
Better: Image Stitching
Binary Labeling

Pixels from Image A

Seam

Pixels from Image B
What defines a good seam?

- **Intuition**: Must not be noticeable
  - Avoid introducing new gradients

- **Breaking it down on the pixel level**:
  - Color differences between pixels of images should be minimal at seam pixels

![Image A](image_a.png) ![Image B](image_b.png)
Computing good seams

\[ C = |I_A(p) - I_B(p)|^2 + |I_A(q) - I_B(q)|^2 \]
Computing good seams

- Graph Cut to compute the minimal cost seam
  - Very fast, global optimal solver for binary label problems
  - NPP primitive (upcoming 3.2 release)

Best seam is the Minimum Cut
NPP Graphcut

- Graph is stored in Arrays
  - Weights to terminals in one array
  - Weights to neighbors in four arrays, horizontal edges are transposed

- `nppiGraphcut_32s8u(d_terminals, d_left_transposed, d_right_transposed, d_top, d_bottom, step, transposed_step, size, d_labels, label_step, pBuffer);`
Binary Labeling

Pixels from Image A

Pixels from Image B

Seam
From two images to many

- One label for each image in the set: N labels
  - Assign each pixel in the panorama one label
  - Unfortunately this is NP hard problem 😞

- Alpha-Expansion algorithm to the rescue
  - Intuition: “Keep current label or change to alpha”
  - Again binary problem solvable with Graph Cut
  - Repeat for all labels until the optimal solution is found
Alpha-Expansion

- Iteration for each label alpha:
  - Compute data term for alpha
  - Compute neighborhood terms for alpha
  - Solve binary Graph Cut
  - Use binary solution to get expanded multi-label solution
  - Compute total energy of expanded solution
  - If energy has decreased make this the current solution, otherwise discard
Alpha-Expansion

Initial Solution
Alpha-Expansion

After 6 Expansion Steps
Alpha-Expansion

After 12 Expansion Steps
Alpha-Expansion

After 18 Expansion Steps

Final Result
Image Stitching Notes

- In this example only 6 out of the 7 input images contribute to the final result
  - Image Stitching reduces the image set

- Quality improves with each iteration
  - The current result is a preview that converges to the final result
Performance

Test data set

Image 0

Image 1
Performance

- CUDA 3.0, Driver 260.16 wall clock times

### Alpha Expansion Performance

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GTX460</td>
</tr>
<tr>
<td>1</td>
<td>101.74</td>
</tr>
<tr>
<td>2</td>
<td>102.85</td>
</tr>
<tr>
<td>3</td>
<td>31.95</td>
</tr>
<tr>
<td>4</td>
<td>28.59</td>
</tr>
<tr>
<td>5</td>
<td>28.18</td>
</tr>
<tr>
<td>6</td>
<td>28.92</td>
</tr>
</tbody>
</table>
Image Pipeline: Panorama Stitching

1. **Radial Distortion Correction**
2. **Keypoint Detection & Extraction (Shi-Tomasi/SIFT)**
3. **Keypoint Matching**
4. **Recover Homography (RANSAC)**
5. **Projective Transform**
6. **Image Stitching**

**Left Image**

**Right Image**
Example Application
Questions?