GPUGeoS: Geodesic Distance Transforms for Image and Video Editing

Timo Stich - tstich@nvidia.com
Overview

- Motivation
  - Interactive Image Segmentation
  - Graph Cut vs Geodesic Distance Transform

- GPUGeoS Implementation
  - Block Strategies
  - Grid Strategies

- Benchmarks
Motivation

Take a Picture

Image Segmentation
(This Talk)

Play Golf in Ireland!

Photo Credits: Golf Girl, Bob Cotter; Cliffs of Moher, Peter Gorman, CC 2.0
Interactive Segmentation

- Strokes define sets of pixels that are FG and BG
- Anchor the solution (spatial constraint)
- Sample of the Color Distributions (color constraint)
Edge-Aware Segmentation

Naïve Segmentation + Edge-Aware Segmentation = Edge-Aware Segmentation
Edge-Aware Segmentation with GDT

- Proposed by Criminisi et al. 2010

- For this talk focus on main part: Computing Geodesic Distance Transforms

- Geodesic means that distances vary per pixel
  - Euclidean Distance + Color Distance -> Edge Awareness

- For more details please see the original paper!
# Graph Cut vs Geodesic Distance Transform

<table>
<thead>
<tr>
<th></th>
<th>Geodesic Distance Transform</th>
<th>Graph Cut*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge-Aware</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Boundary</td>
<td>Soft</td>
<td>Hard</td>
</tr>
<tr>
<td>Fidelity</td>
<td>Local Optimum</td>
<td>Global Optimum</td>
</tr>
<tr>
<td>Performance Scaling</td>
<td>Scales Linearly</td>
<td>Size &amp; Problem Dependent</td>
</tr>
<tr>
<td>Memory</td>
<td>~2 x Image Size</td>
<td>~10 x Image Size</td>
</tr>
</tbody>
</table>

*For more information on Graph Cut please replay my GTC 2012 talk*
Serial Geodesic Distance Transform

- Two-Pass Updates in Scan Order
  - Top-Left to Bottom-Right
  - Bottom-Right to Top-Left

- Tight data dependencies kill parallelization
  - Parallelism per update but 4 threads don’t fill a GPU!

- Need an alternative approach!
Parallel Geodesic Distance Transforms

- Based on work by Weber et al. 2008

- Four instead of two sweeps
  - Removes the most critical dependency!

- Exploit Parallelism on all hierarchy Levels
  - Per Update
  - Per Block
  - Per Grid
4-Sweep Update per Pixel (Down Sweep)

\[
X = \min \left[ L, U, R \right] + D(L) + D(U) + D(R)
\]
Data Dependencies per Update

\[
X = \min \left( \begin{array}{c}
X \\
L \\
U \\
R \\
X \\
\end{array} \right) + \left( \begin{array}{c}
D(L) \\
D(U) \\
D(R) \\
& \\
\end{array} \right)
\]

- \(D(*)\) : Independent
- : Dependent on previous Row
- : Input Independent

Compute \(D(L)\) | Fetch \(X\) | Fetch \(L\) | \(\min(X, D(L) + L)\)
Compute \(D(U)\) | Fetch \(X\) | Fetch \(U\) | \(\min(X, D(U) + U)\)
Compute \(D(R)\) | Fetch \(X\) | Fetch \(R\) | \(\min(X, D(R) + R)\)

\(\min = X\)
Summary: Parallelism Per Update

- Full parallelism in Prefetch phase
  - Compute of D(*)
  - Fetch of X
- 3-way parallelism for computing intermediate mins
  - Each dependent on one element of previous row
- 1-way for final min and write out
Naïve Approach

- How to make sure that previous row is available?

- Naïve Solution: One Kernel per Row
  - Global Sync between kernels ensures data is valid for each kernel

- Unfortunately, not a good idea...
  - Kernel launch overhead gets significant
  - Each kernel has very limited total parallelism
Multiple Rows per Kernel

- Weber et al: One block computes multiple rows
  - Means more work per thread
  - Threads sync after each row and communicate results
  - Significantly reduces # of total kernel launches

Problem:
- How to resolve intra-row dependencies between blocks?
Data Dependencies per Block

Data Available

Block A

Not Available Yet

Not Available Yet

L  U  R

X
Data Dependencies per Block

Block A

Block A+1
Data Dependencies per Slice of Rows

Solution: Overlap the blocks to compensate, write out only valid updates
(Does throw away some of the computation)
Summary: Parallelism Per Slice Update

- **CUDA Block Parallelism**
  - 2-Warp wide Blocks (64 Elements)
  - 3 Warps per Direction (2x3x32 = 192 Threads per Block)
  - 8 Rows per Block (Output Width: 50 Elements)

- **BUT**: For a 2Kx2K image that’s still only 41 Blocks per Kernel
  - Tesla K20 has 15 SMX, that’s ~ 2.7 (small) Blocks per SMX
  - If we could increase blocks per kernel, perf would go up...
Summary: Parallelism Per Update (Recap)

- Full parallelism in Prefetch phase
  - Compute of D(*)
  - Fetch of X

- 3-way parallelism for computing intermediate mins
  - Each dependent on one element of previous row

- 1-way for final min and write out

X 8 Rows per Block!
Scoreboard to Resolve Row Dependencies

- Same Pattern as Element Update, this time on Blocks:
  - L
  - U
  - R
  - X

- Scoreboard Implementation:
  - Memory in GMEM
  - Wait: busy fetch to volatile*
  - Update: __threadfence() & __syncthreads, then write

- Scoreboard Implementation:
  - Wait until Dependencies are available
  - Signal this blocks data is available
Summary: Parallelism with Scoreboard

- CUDA Block Parallelism
  - 2-Warp wide Blocks (64 Elements)
  - 3 Warps per Direction (2x3x32 = 192 Threads per Block)
  - 8 Rows per Block (Output Width: 50 Elements)

- SINGLE Kernel launch of all blocks
  - Prefetch phase runs fully in parallel
  - Compute phase dependencies resolved via Scoreboard
## Benchmarks

<table>
<thead>
<tr>
<th>Image Size</th>
<th>GPU GeoS w/o Scoreboard</th>
<th>GPU GeoS w/ Scoreboard</th>
<th>Speedup w/ Scoreboard</th>
<th>GPU Graph Cut (from NPP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 MegaPixel</td>
<td>50.9 ms</td>
<td>23.1 ms</td>
<td>2.2 x</td>
<td>32.1 ms</td>
</tr>
<tr>
<td>2.9 MegaPixel</td>
<td>81.6 ms</td>
<td>37.6 ms</td>
<td>2.2 x</td>
<td>72.6 ms</td>
</tr>
<tr>
<td>6.0 MegaPixel</td>
<td>103.7 ms</td>
<td>50.6 ms</td>
<td>2.0 x</td>
<td>176.9 ms</td>
</tr>
<tr>
<td>10.0 MegaPixel</td>
<td>178.8 ms</td>
<td>85.3 ms</td>
<td>2.1 x</td>
<td>1001.7 ms</td>
</tr>
</tbody>
</table>
Questions?

▪ Contact
  – tstich@nvidia.com

▪ References
  – Antonio Criminisi, Toby Sharp, Carsten Rother, Patrick P'erez, Geodesic image and video editing, ACM Transactions on Graphics (TOG), v.29 n.5, p.1-15, October 2010