GPU TECHNOLOGY CONFERENCE

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

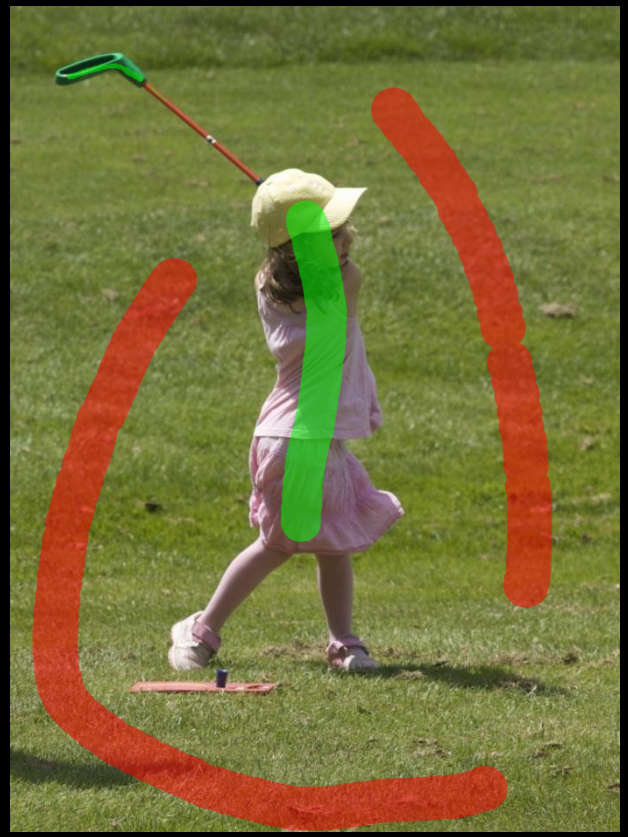
Image Segmentation
(This Talk)

Play Golf in Ireland!

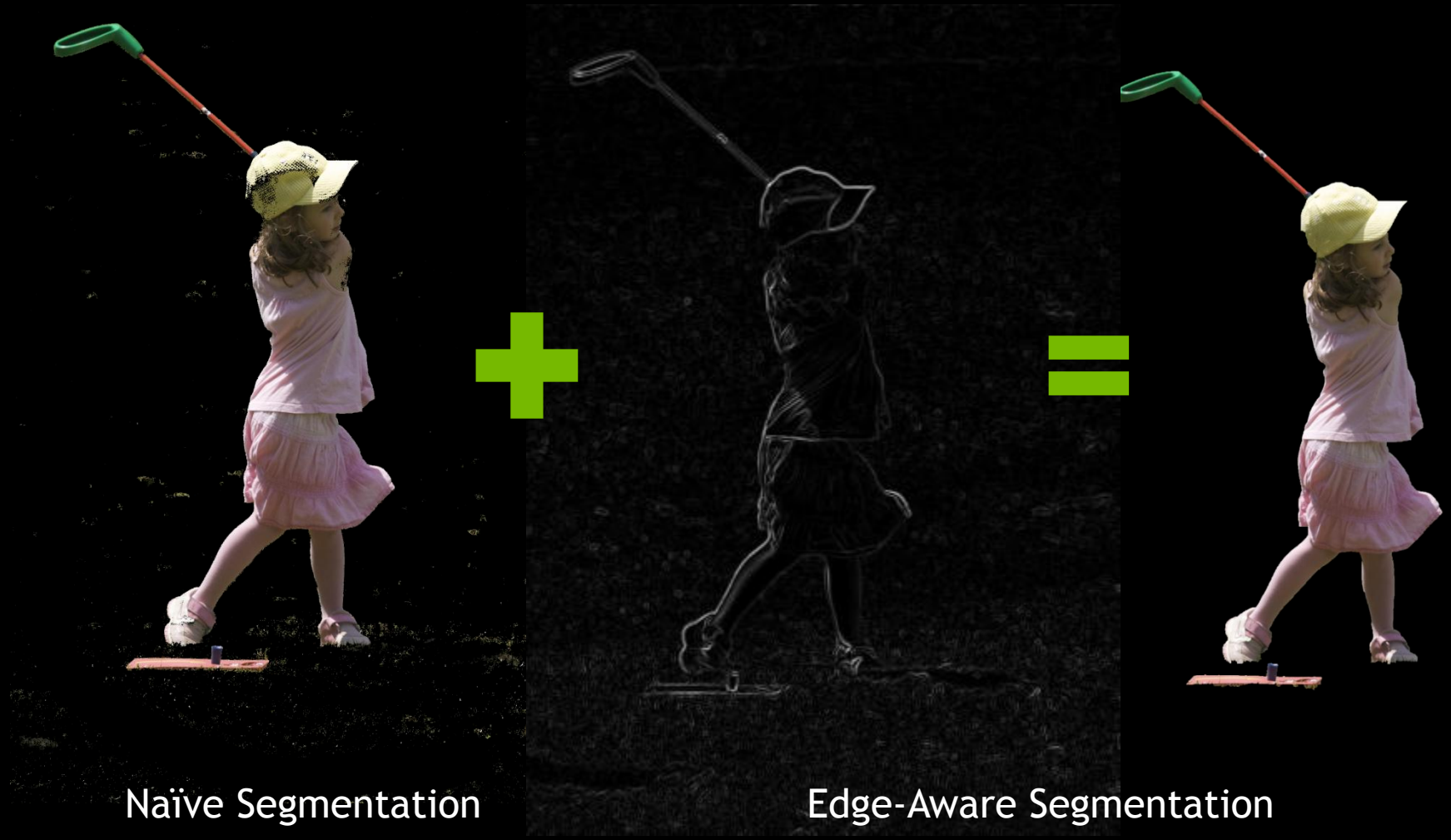
Photo Credits: Golf Girl, Bob Cotter; Cliffs of Moher, Peter Gorman, CC 2.0

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



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!

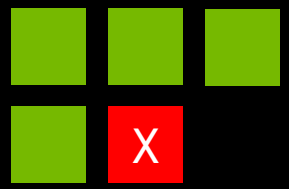
Graph Cut vs Geodesic Distance Transform

	Geodesic Distance Transform	Graph Cut*
Edge-Aware	Yes	Yes
Selection Boundary	Soft	Hard
Fidelity	Local Optimum	Global Optimum
Performance Scaling	Scales Linearly	Size & Problem Dependent
Memory	~2 x Image Size	~10 x Image Size

*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

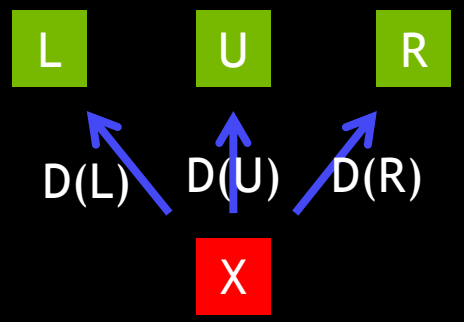


- Tight data dependencies kill parallelization
  - Parallelism per update but 4 threads don't fill a GPU!
- Need an alternative approach!

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 hierachy Levels
  - Per Update
  - Per Block
  - Per Grid

4-Sweep Update per Pixel (Down Sweep)



$$X = \mathrm{MIN}\begin{bmatrix} L + D(L) \\ U + D(U) \\ R + D(R) \\ X \end{bmatrix}$$

Data Dependencies per Update

$$X = \mathrm{MIN}\left[\begin{array}{lcl} L & + & D(L) \\ U & + & D(U) \\ R & + & D(R) \\ X & & \end{array}\right]$$

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 parallelsim for computing intermediate mins
  - Each dependent on <u>one</u> element of previous row
- 1-way for final min and write out

Naïve Approach

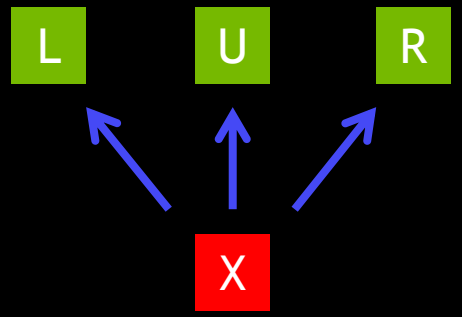
- How to make sure that previous row is available?
- Naïve Solution: One Kernel per Row
  - Global Sync between kernels ensures data is valid for each kernel
- Unfortunately, not a good idea…
  - Kernel launch overhead gets significant
  - Each kernel has very limited total parallelism

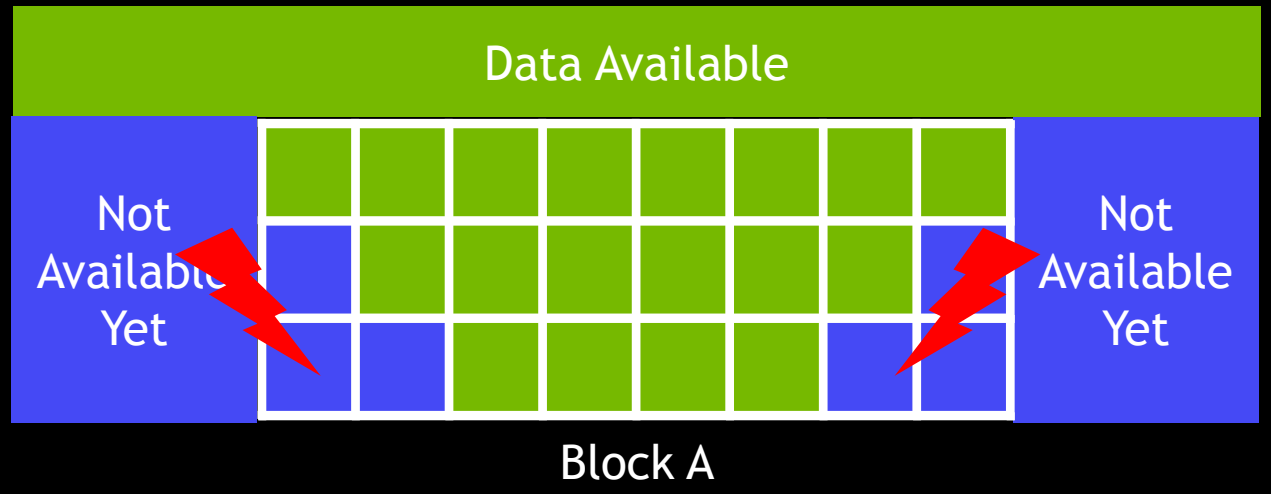
Multiple Rows per Kernel

- 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

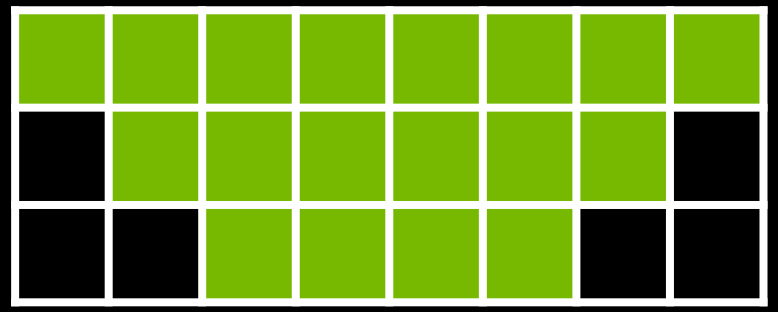
- Problem:
  - How to resolve intra-row dependencies between blocks?

Data Dependencies per Block

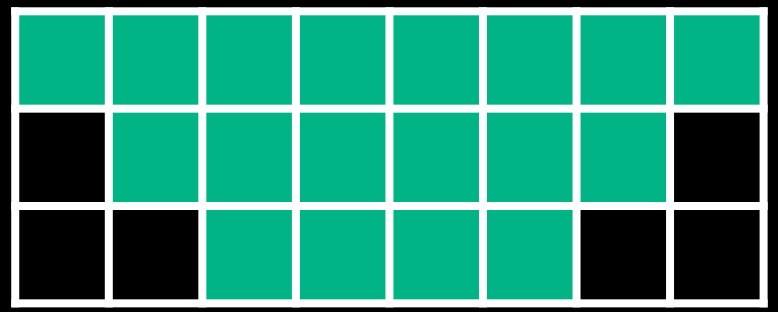




Data Dependencies per Block

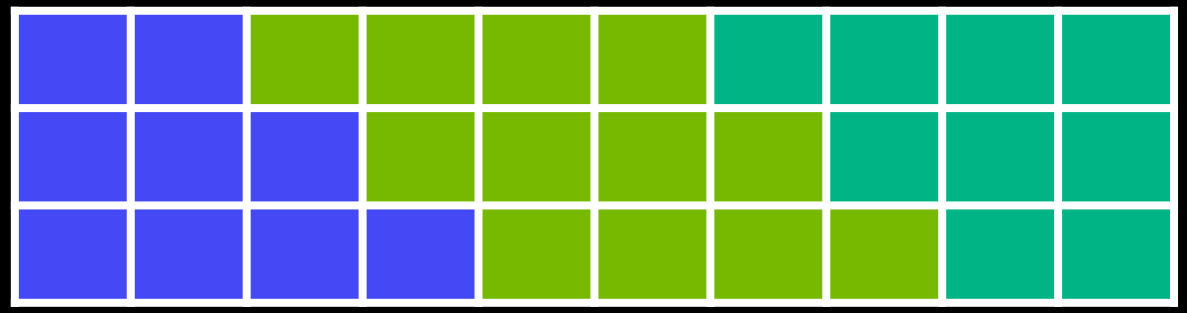


Block A



Block A+1

Data Dependencies per Slice of Rows



Block A-1 Block A Block A+2

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…

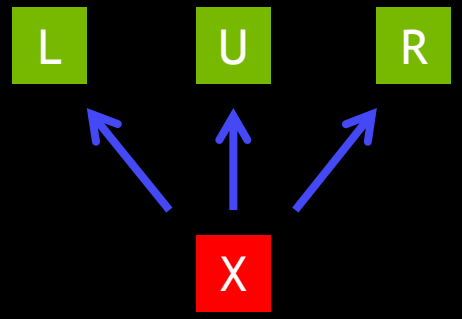
Summary: Parallelism Per Update (Recap)

- Full parallelism in Prefetch phase
  - Compute of D(*)
  - Fetch of X
- 3-way parallelsim for computing intermediate mins
  - Each dependent on <u>one</u> 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:



Prefetch | Block Update

Scoreboard: Wait until Dependencies are available

Scoreboard: Signal this blocks data is available

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

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 depencies resolved via Scoreboard

Benchmarks

GK110/ Image Size	GPU GeoS w/o Scoreboard	GPU GeoS w/ Scoreboard	Speedup w/ Scoreboard	GPU Graph Cut (from NPP)
1.2 MegaPixel	50.9 ms	23.1 ms	2.2 x	32.1 ms
2.9 MegaPixel	81.6 ms	37.6 ms	2.2 x	72.6 ms
6.0 MegaPixel	103.7 ms	50.6 ms	2.0 x	176.9 ms
10.0 MegaPixel	178.8 ms	85.3 ms	2.1 x	1001.7 ms

GPU TECHNOLOGY CONFERENCE

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
  - Weber, O., Devir, Y. S., Bronstein, A. M., Bronstein, M. M., and Kimmel, R. 2008. Parallel algorithms for approximation of distance maps on parametric surfaces. In Proceedings of ACM SIGGRAPH.