

Real-time Image Segmentation

Miklos Homolya, Ravikishore Kommajosyula, Gaurav Kukreja

Technical University of Munich

April 3, 2014

Overview

- 1 Introduction
- 2 Algorithm
 - Binary Image Segmentation
 - Grayscale Image Segmentation
 - Primal-Dual Method
- 3 CUDA Implementation
- 4 Optimizations
 - Texture Memory
 - OpenGL Interoperability

Problem Definition

Binary Image Segmentation

Energy functional

$$E_1(u) := \int_{\mathbb{R}^N} |\nabla u| + \lambda \int_{\mathbb{R}^N} |u(x) - f(x)| \, dx$$

Functional derivative

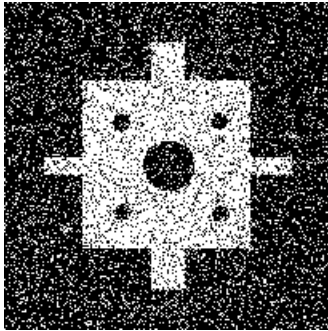
$$\frac{\delta E_1}{\delta u} = -\operatorname{div} \left(\frac{\nabla u}{|\nabla u|} \right) + \lambda \frac{u - f}{|u - f|}$$

Gradient descent solver

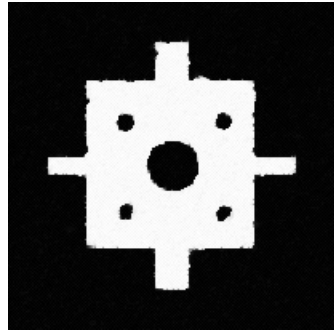


Tony F. Chan, Selim Esedoglu and Mila Nikolova (2005)
Finding the Global Minimum for Binary Image Restoration

Sample Result



Noisy binary image.



Restored binary image.

Grayscale Image Segmentation

Euler-Lagrange equation

$$\operatorname{div} \left(\frac{\nabla u}{|\nabla u|} \right) - \lambda s(x) - \alpha \nu'(u) = 0$$

where $s(x) = (c_1 - f(x))^2 - (c_2 - f(x))^2$, and $\alpha \nu'(u)$ forces u into $[0; 1]$.

Gradient descent solver



Tony F. Chan, Selim Esedoglu and Mila Nikolova (2004)
Algorithms for Finding Global Minimizers of Image
Segmentation and Denoising Models

Sample Result



Grayscale input image.



Segmentation (without thresholding).

Primal-Dual Method

Motivation: Gradient descent solver has slow convergence.

Primal variable $u \in \mathcal{C}$

$$u : \Omega \rightarrow [0; 1]$$

Dual variable $\xi \in \mathcal{K}$ ($\xi \sim \text{grad } u$)

$$\xi : \Omega \rightarrow \{(x, y) : x^2 + y^2 \leq 1\}$$

Algorithm:

$$\xi^{n+1} = \Pi_{\mathcal{K}}(\xi^n - \sigma \nabla \bar{u}^n)$$

$$u^{n+1} = \Pi_{\mathcal{C}}(u^n - \tau(\text{div} \xi^{n+1} + s))$$

$$\bar{u}^{n+1} = u^{n+1} + (u^{n+1} - u^n) = 2u^{n+1} - u^n$$

$\Pi_{\mathcal{C}}$ and $\Pi_{\mathcal{K}}$ clamp the range to fit \mathcal{C} and \mathcal{K} respectively.

Result

- A single iteration is costlier than for the gradient descent solver, but we could reduce iteration count from 2000 to 160.
- Huge impact on performance.

CUDA Implementation

- Update kernels calls from CPU to have synchronization
- Update ξ and update u implemented as two kernels
- Image arrays for u^{n-1} and u^n swapped after each iteration
- Branching to avoid invalid memory accesses

CUDA Implementation

- Swapping images after each iteration makes things difficult
- Cannot be used in gradient calculation. Can be used in divergence calculation
- Texture memory used on intermediate results ξ_x and ξ_y
- Improves the FPS by 12%

How to use OpenGL Interop?

- Set current threads OpenGL context to use for OpenGL interop with CUDA **device**.

```
cudaGLSetGLDevice(device);
```

- Create OpenGL Pixel Buffer, and register to use as CUDA buffer.

```
gl.glGenBuffers(1, &pixels);  
gl.glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pixels);  
size_t size = w * h * 4 * sizeof(unsigned char);  
gl.glBufferData(GL_PIXEL_UNPACK_BUFFER, size, 0,  
                GL_DYNAMIC_DRAW);  
cudaGraphicsGLRegisterBuffer(&pixels_CUDA, pixels,  
                             cudaGraphicsMapFlagsWriteDiscard);
```

How to use OpenGL Interop?

Inside the Display Loop,

- Before starting kernel, map pixel buffer to a CUDA pointer.

```
cudaGraphicsMapResources(1, &pixels_CUDA, 0);  
cudaGraphicsResourceGetMappedPointer(&d_pixels, &size,  
    pixels_CUDA);
```

- Pass CUDA pointer as parameter for kernel. The kernel writes to the buffer in **RGBA8** format.
- After kernel execution, unmap pixel buffer.

```
cudaGraphicsUnmapResources(1, &pixels_CUDA, 0);
```

- Draw buffer

```
glDrawPixels(w, h, GL_RGBA, GL_UNSIGNED_BYTE, 0);
```

Performance Improvement from Optimizations

After Texture Memory Optimization and OpenGL Interop, FPS increases by 33 % from 6.66 to 8.80 FPS.

Demo

Thank you for your attention.