Image Smoothing via L_0 Gradient Minimization

GPU programming in Computer Vision

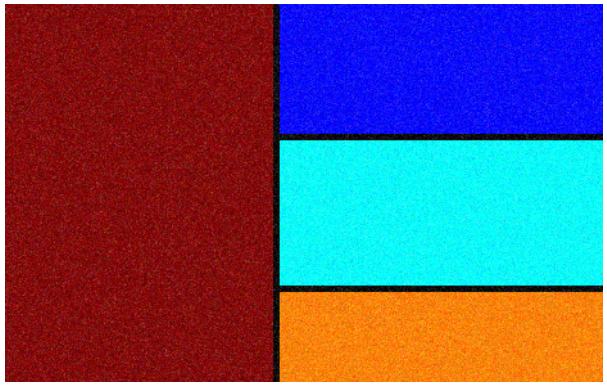
Marat Faizov,
Elena Voskoboinikova,
Afshin Loni

31 March, 2015

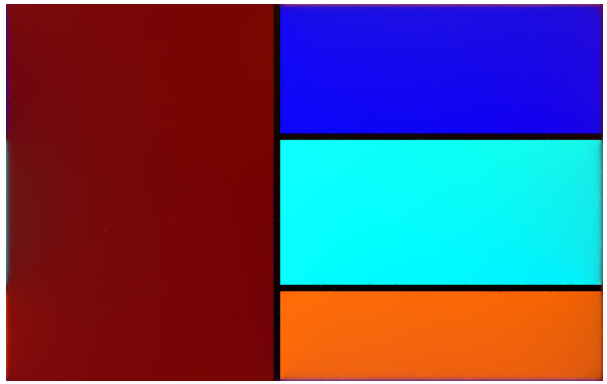
Table of Contents

1. Introduction
2. Description of the algorithm
3. cuFFT library
4. Implementation
5. Results
6. Comparison with Matlab
7. Summary

Introduction



input image



$\lambda = 0.10$

Main goals:

- to enhance, sharpen fundamental edges
- to remove noise
- to diminish insignificant details

Main idea of the method

To compute the gradient measure:

$$C(S) = \#\{p : |\partial_x S_p| + |\partial_y S_p| \neq 0\}, \tag{1}$$

where S is the smoothed result of the input signal, $\nabla S_p = (\partial_x S_p, \partial_y S_p)$ is the gradient of pixel p, $\#$ is the counting operator, that gives the L_0 norm of gradient.

The result S must be structurally similar to the input I:

$$\min_S \{\sum_p (S_p - I_p)^2 + \lambda C(S)\}, \tag{2}$$

where I is the input image, S is the computed result, λ is a weight controlling the significance of $C(S)$ (smoothing parameter).

Main idea of the method

$$\min_S \{\sum_p (S_p - I_p)^2 + \lambda C(S)\} \tag{2}$$

How to find the minumim (2)?

- to introduce **auxilary variables** h_p and v_p
- to control the similarity between variables (h, v) and $(\partial_x S, \partial_y S)$ using **parameter** β
- to rewrite the function that needs to be minimized

$$\min_{S,h,v} \{\sum_p (S_p - I_p)^2 + \lambda C(h, v) + \beta \left((\partial_x S_p - h_p)^2 + (\partial_y S_p - v_p)^2\right)\}, \tag{3}$$

where $C(h, v) = \#\{p : |h_p| + |v_p| \neq 0\}$, β is an automatically adapting parameter.

Description of the algorithm

Subproblem 1: computing (h, v)

To solve

$$\min_{h,v}\{\sum_{p}\left((\partial_x S_p - h_p)^2 + (\partial_y S_p - v_p)^2\right) + \frac{\lambda}{\beta}C(h,v)\}, \tag{4}$$

where $C(h,v)$ returns the number of nonzero elements in $|h| + |v|$.

How?

To estimate each h_p and v_p individually:

$$\sum_{p}\min_{h_p,v_p}\{(h_p - \partial_x S_p)^2 + (v_p - \partial_y S_p)^2 + \frac{\lambda}{\beta}H(|h_p| + |v_p|)\}, \tag{5}$$

where

$$H(|h_p| + |v_p|) \quad = \quad \begin{cases} 1 & \text{if} \quad |h_p| + |v_p| \neq 0, \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

using the condition for the minimum:

$$(h_p, v_p) \quad = \quad \begin{cases} (0,0) & \text{if} \quad (\partial_x S_p)^2 + (\partial_y S_p)^2 \leq \frac{\lambda}{\beta}, \\ (\partial_x S_p, \partial_y S_p) & \text{otherwise} \end{cases} \tag{7}$$

Description of the algorithm

Subproblem 2: computing S

To solve

$$\min_S \{\sum_p (S_p - I_p)^2 + \beta \left((\partial_x S_p - h_p)^2 + (\partial_y S_p - v_p)^2\right)\} \tag{8}$$

by diagonalizing derivative operators by Fast Fourier Transform (FFT):

$$S = F^{-1}\left(\frac{F(I) + \beta(F(\partial_x)^* F(h) + F(\partial_y)^* F(v))}{F(1) + \beta(F(\partial_x)^* F(\partial_x) + F(\partial_y)^* F(\partial_y))}\right), \tag{9}$$

where F is the FFT operator, $F()^*$ is the complex conjugate.

Algorithm: L_0 Gradient Minimization

Input: image I, smoothing weight λ, parameters β_0, β_{max}, rate κ
Initialization: $S = I$, $\beta = \beta_0$, $i = 0$
repeat
    With $S^{(i)}$, solve for $h_p^{(i)}$ and $v_p^{(i)}$
    With $h^{(i)}$ and $v^{(i)}$, solve for $S^{(i+1)}$
    $\beta = \kappa\beta$, $i++$
until $\beta \geq \beta_{max}$
Output: result image S

cuFFT Library

Discrete Fourier Transform (DFT)

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{kn}{N}},$$

where x_k is a complex-valued vector (time domain), X_k is the frequency domain.

cuFFT

- NVIDIA CUDA Fast Fourier Transform (FFT) product
- modeled after FFTW (efficient CPU-based FFT library)
- to optimize the transform a mechanism called *plan* is used

cuFFT Library

Fourier Transform Setup

a *plan* (of execution)

- uses internal building blocks to optimize the transform
- is built for the given configuration and particular hardware selected

to create a plan using:

- cufftPlan1D() / cufftPlan2D() / cufftPlan3D()
- cufftPlanMany() (supports batched input and strided data layout)

cuFFT Library

Computing $F(h)$ using cuFFT

```
{
    cufftHandle rcfftPlan;
    cufftComplex *d_ffth;
    ...
    cudaMalloc(&d_ffth, img_size*sizeof(cufftComplex));
    cufftPlanMany(&rcfftPlan, 2, sizes,embed, 1, h * w,
    embed, 1, h * w, CUFFT_R2C, nc);
    ...
    cufftExecR2C(rcfftPlan, d_h, d_ffth);
    cudaDeviceSynchronize();
    ...
    cufftDestroy(rcfftPlan);
    cudaFree(d_h);
    cudaFree(d_ffth);
}
```

Implementation

Arrays on the device:

h_p, v_p :	d_h, d_v
$F(1)$:	d_fft1
$F(I)$:	$d_fftimgIn$
$(F(\partial_x)^* F(\partial_x) + F(\partial_y)^* F(\partial_y))$:	d_MFT
$(F(\partial_x)^* F(h) + F(\partial_y)^* F(v))$:	d_FS

Parameters set:

$\lambda = 0.02$ (adjustable, to maintain coarsness)

$\beta_0 = 2\lambda$

$\beta_{max} = 10^5$

$\kappa = 2$

Results

Details flattening + sharp edge preservation

input image

$\lambda = 0.02$

Results

Details flattening + sharp edge preservation

input image

$\lambda = 0.01$

Results

Smoothing for edge detection

input image

$\lambda = 0.01$

$\lambda = 0.02$

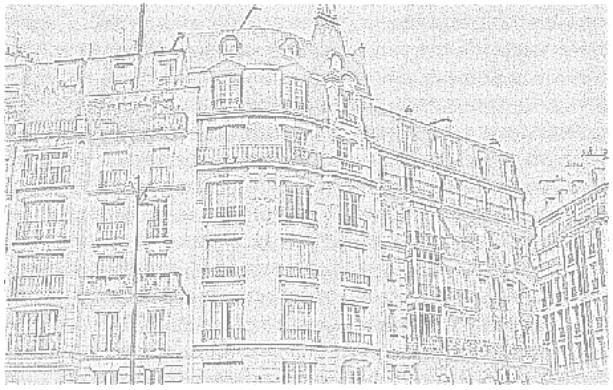
$\lambda = 0.10$

Results

Smoothing for edge detection

(a) Input image

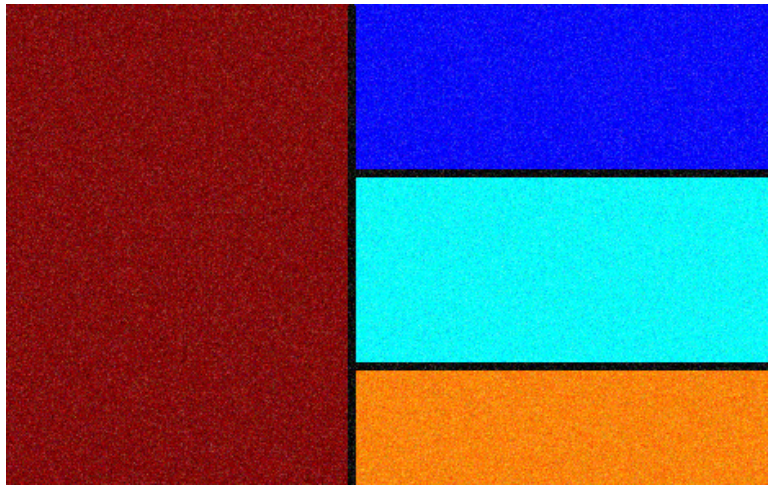
(b) The image after smoothing $\lambda = 0.02$



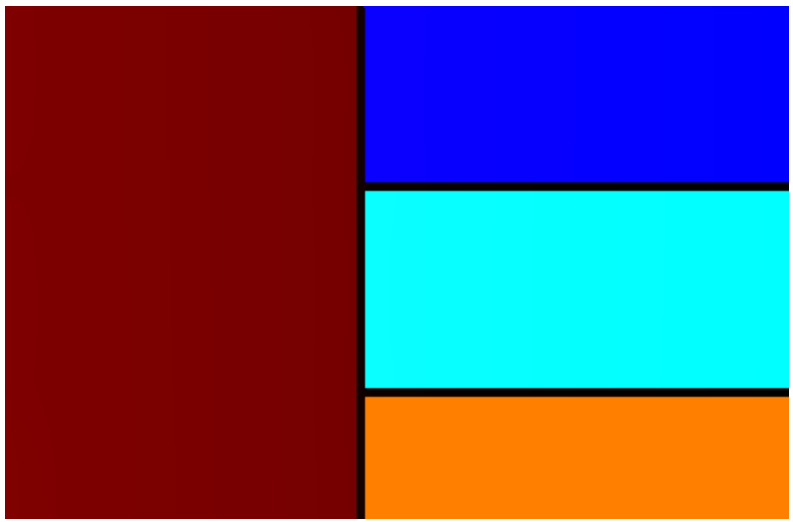
(c) Sketching filtering of (a)

(d) Sketching filtering of (b)

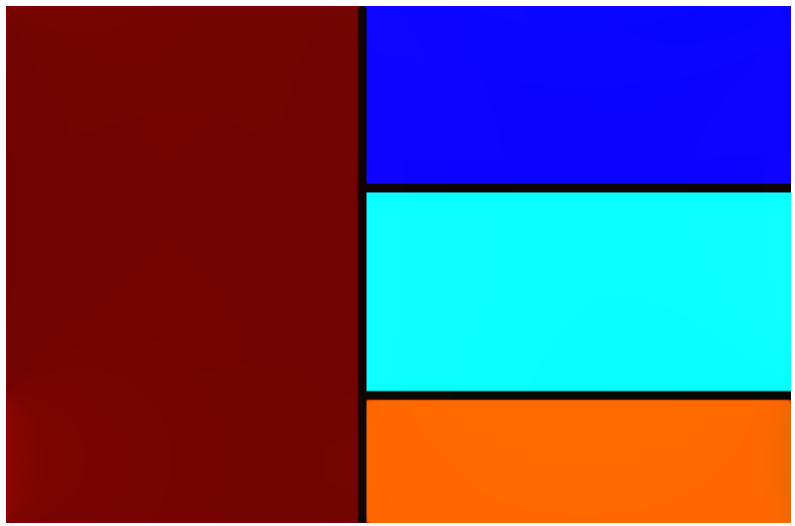
Comparison with Matlab



(a) input

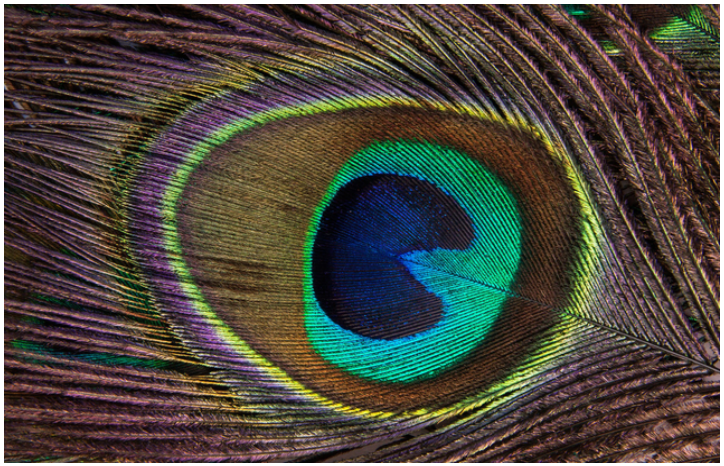


(b) [Xu, Lu] Matlab $\lambda = 0.1$, $\kappa = 2$

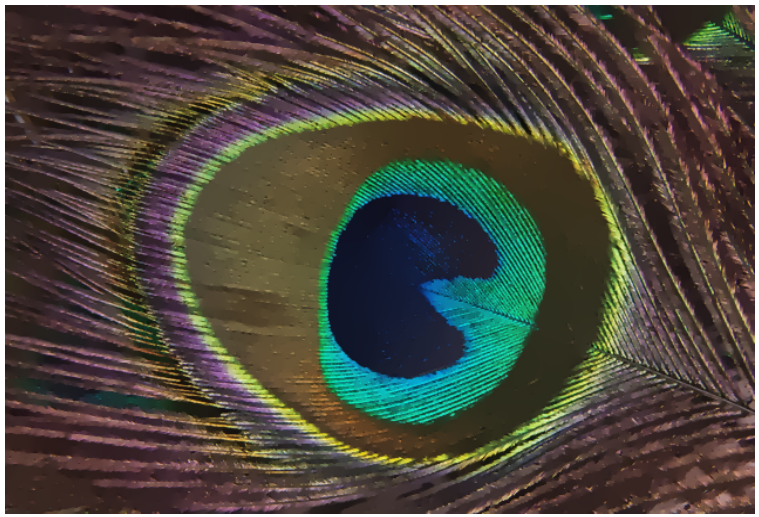


(c) [Xu, Lu] ours $\lambda = 0.1$, $\kappa = 2$

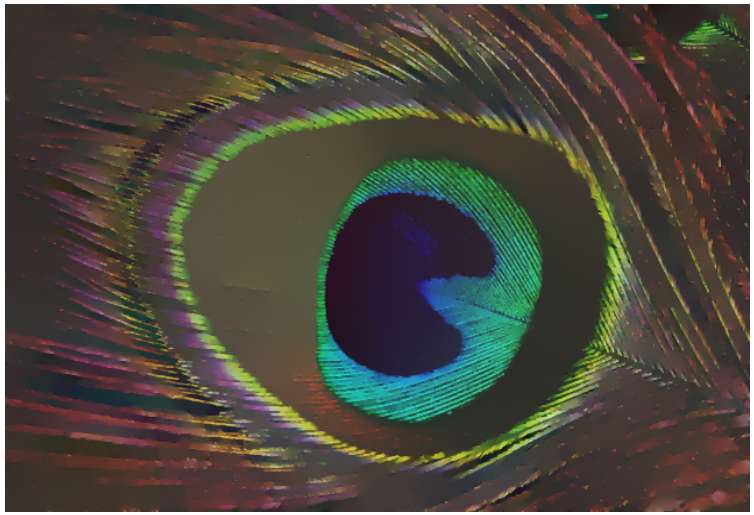
Comparison with Matlab



(a) input

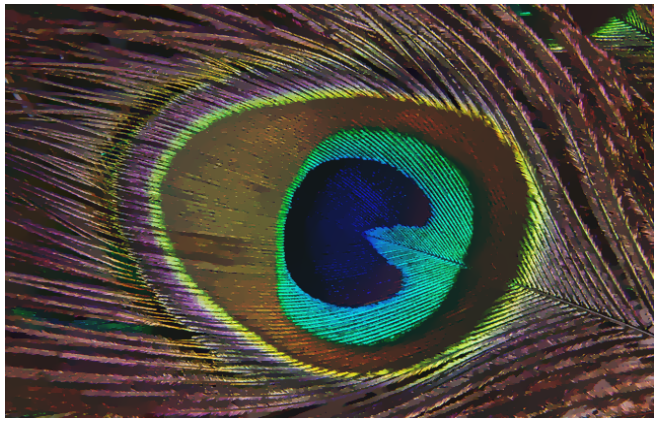


(b) [Xu, Lu] Matlab $\lambda = 0.1$, $\kappa = 8$

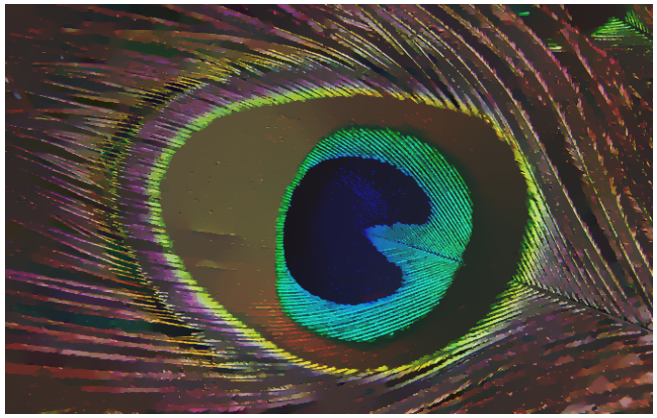


(c) [Xu, Lu] ours $\lambda = 0.1$, $\kappa = 8$

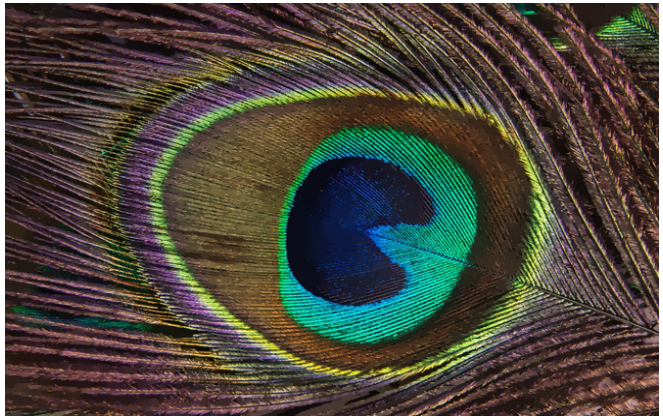
Comparison with Matlab



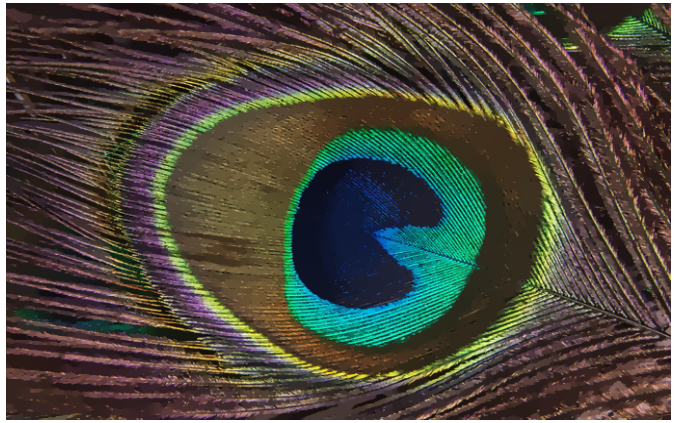
(a) [Xu, Lu] ours $\lambda = 0.01$, $\kappa = 2$



(b) [Xu, Lu] ours $\lambda = 0.02$, $\kappa = 2$



(c) [Xu, Lu] Matlab $\lambda = 0.01$, $\kappa = 2$



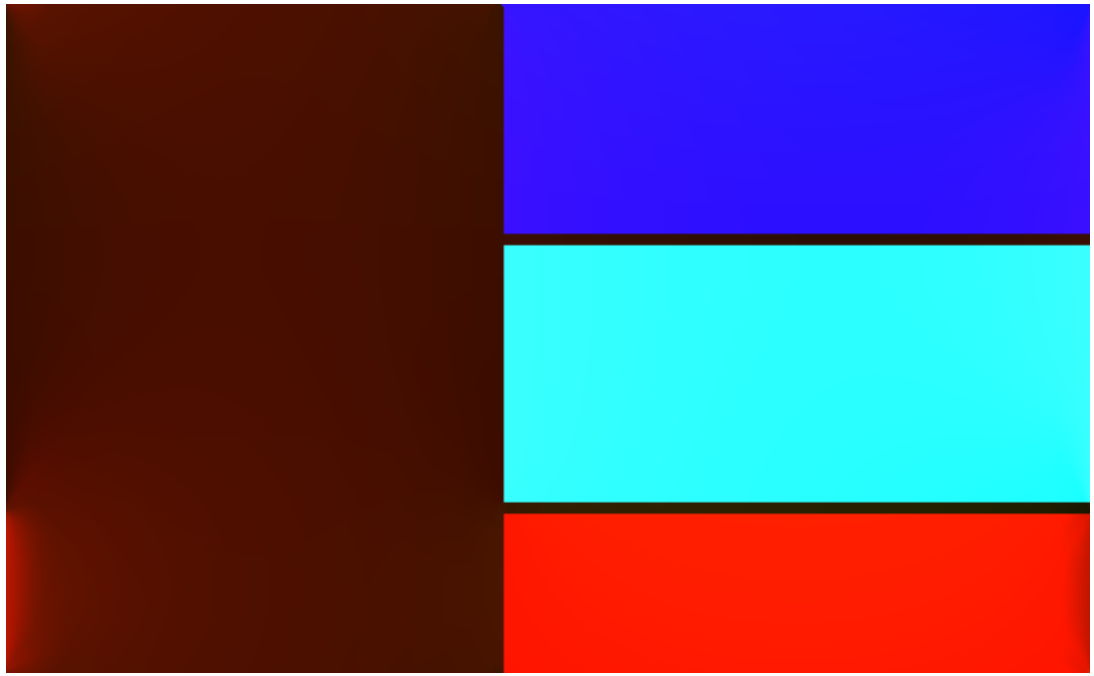
(d) [Xu, Lu] Matlab $\lambda = 0.02$, $\kappa = 2$

Comparison with Matlab

Input image:	feather.png
Size of the input image:	640×427
Number of channels:	3
GPU:	GeForce GTX 750 Ti
Number of iterations:	23

	$\lambda = 0.02$, $\kappa = 2.0$	$\lambda = 0.1$, $\kappa = 2.0$	$\lambda = 0.1$, $\kappa = 8.0$
Our time	82ms	105ms	27ms
Matlab time	7873ms	4697ms	2501ms

Summary



[Xu, Lu] ours $\lambda = 0.5$, $\kappa = 2$

Summary

- the runtime is obviously higher in the CPU version comparing to GPU
- more speedup is possible using shared memory for our kernels
- most of the runtime is dedicated to cuFFT library command
  - for each iteration:
    $\approx$ 1.5 ms for kernels
    $\approx$ 6 ms for cuFFT commands
- more possible developments of the code are:
  - doing the bilateral filtering
  - creating the sketch output using OpenCL

Resources

Xu, L., Lu, C., Xu, Y., Jia, J. 2011. Image Smoothing via L_0 Gradient Minimization. *ACM Trans. Graph. 30*, 6, Article 174 (December 2011), 11 pages. DOI = 10.1145/2024156.2024208
http://doi.acm.org/10.1145/2024156.2024208