COHERENCE ENHANCING SHOCK-FILTERING ON A MOBILE PHONE

Markus Mainberger

June 13, 2007

Overview

Motivation

Really? One can even phone with this thing?

- The additional functions of a mobile phone have increased a lot, e.g. taking pictures.
- Obviously also the computational power has increased.
  But: Abilities of the used processors are still limited

Question:
Is it possible to apply highly demanding image processing routines on a modern mobile phone?

The used Routine

Coherence-Enhancing Shock Filters

Figure: Example images of size 256×256 for almost *artistic results* of coherence-enhancing shock filtering. (Parameters: line width $= 3$, number of iterations $= 20$)

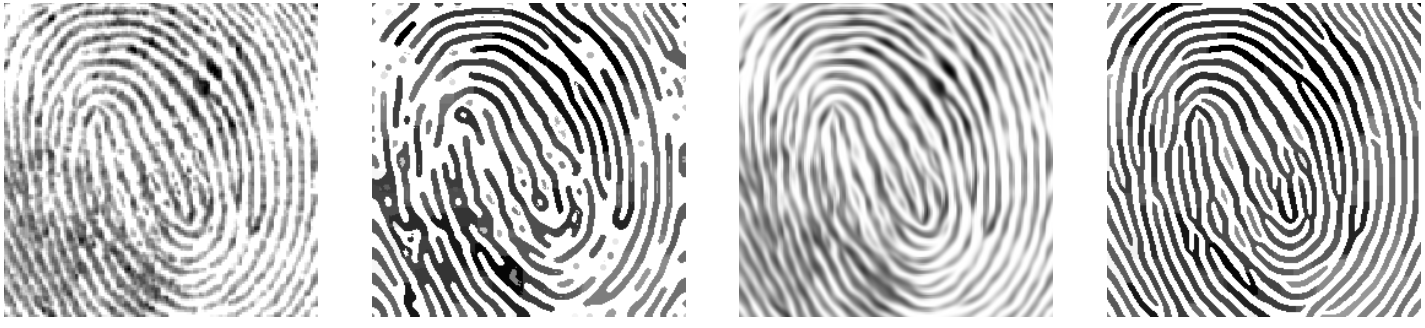

Figure: *Finger print image*, 256×256. From *left to right:* original image, conventional shock filtered image, coherence-enhancing diffusion filtered image, coherence-enhancing shock filtered image

Shock Filters

- *Shock filters* belong to the class of morphological image enhancement methods.
- PDE based formulation:

$$u_t = -\text{sign}(\Delta u)|\nabla u|$$

  with original image f as initial condition.
- Evolution under this PDE:
  Produces at time t a *dilation* ($\Delta u < 0$)/*erosion* ($\Delta u >= 0$) process (disc shaped structuring element of radius t).
- Improvements lead to

$$u_t = -\text{sign}(v_{\eta\eta})|\nabla u|$$

  where $\eta \| \nabla v$ and $v := K_\sigma * u$.

Structure Tensor

- Shock filter performance depends strongly on direction $\eta \| \nabla (K_\sigma * u)$.
- Adjacent gradients $\nabla (K_\sigma * u)$ with same direction but opposite orientation cancel.

$\Rightarrow$ Use a more reliable descriptor of local structure: the *structure tensor*

$$\text{consider} \quad J_0(\nabla u) = \nabla u \nabla u^\top \quad \text{instead of} \quad \nabla u.$$

Average orientations:

$$J_\rho(\nabla u) = K_\rho * (\nabla u \nabla u^\top)$$

Properties:

- 2×2 matrix, symmetric, positive semidefinite.
- *Eigenvectors* describe the direction where local contrast is maximal/minimal.
- Contrast measured by *eigenvalues*.

Coherence-Enhancing Shock Filters

Let ω be the normalised eigenvector, corresponding to the largest eigenvalue of the structure tensor J_ρ
Reformulate the shock filter as:

$$u_t = -\text{sign}(v_{\omega\omega})|\nabla u|$$

- Due to ω: shocks orthogonal to flow direction.
- Steady state after finite time.
- *Structure scale* σ: size of the flow like patterns (line thickness of $2\sigma - 3\sigma$).
- *Integration scale* ρ: averages orientation information (ρ should be larger than σ).

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

Implementation

The original implementation follows roughly these steps:

Input: Original image $f = u^0$, line width $\sigma >= 3$, number of iterations n.

Output: Filtered image u^n.

1. Apply a convolution using a discretised Gaussian ($v^k := K_\sigma * u^k$).
2. Compute structure tensor $K_\rho * (\nabla u^k \nabla u^{k\top})$ with $\rho = 3 \cdot \sigma$ (derivatives are approximated by Sobel masks).
3. Look for the direction with largest contrast (corresponds to the eigenvector ω).
4. Compute second directional derivative of v^k i.e $v^k_{\omega\omega}$ in this direction using standard finite difference masks.
5. Compute the dilation/erosion of u^k for each pixel using a square of size $\sigma/3$ as structuring element.
6. For all pixels: If $v^k_{\omega\omega} < 0$ return result from dilation for u^{k+1} else return result from erosion for u^{k+1}.
7. Repeat step 1. - 6. until number of iterations n is reached.

The Mobile Phone

- T. Krattinger used a Nokia N80 with 206 MHz ARM926EJ-S Processor and 19MB Ram.
- Operating system Symbian supports C++ applications.
- For an image of size 410×410 the original implemantation needs roughly 8 minutes.

$\Rightarrow$ This is not reasonable for mobile applications.

The goal of Krattinger's diploma thesis was to improve the running time by code optimisation to make the routine appliable on the mobile phone.

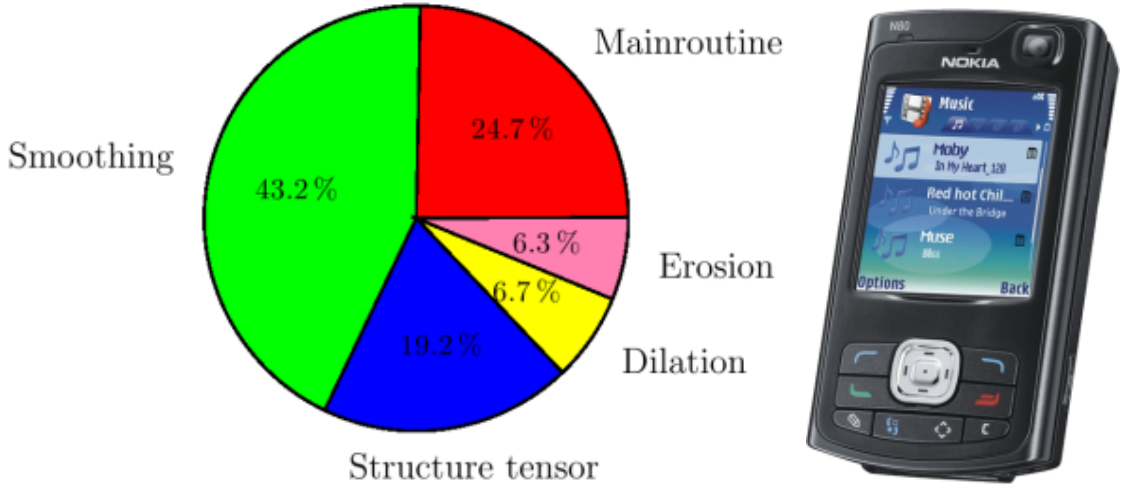

Figure: *Left:* Runtime diagram. *Right:* The Nokia N80 mobile phone

Fixed-point Arithmetic

- The AFM926EJ-S does not support floating point numbers. Instead they are simulated by a sequence of integer operations.

$\Rightarrow$ Use so called *fixed-point numbers*

- less precise, but much faster
- integer computations with some additional shifts for multiplication/division
- Largest/smallest number in our routine is $\pm 1,000,000$. Thus we can use 21 bit places before the decimal point and 11 bit decimal places.

gain in time: 50%

Avoiding Divisions

- Additions/subtractions need 1 cycle
  Multiplications need 2 cycles
  Divisions need *21 cycles*

$\Rightarrow$ avoid divisions by

  - merging fractions: $\frac{1}{a} + \frac{1}{b} = \frac{b+a}{a \cdot b}$
  - resolve double fractions: $\frac{a/b}{c/d} = \frac{a \cdot d}{b \cdot c}$
  - precomputing fractions: $c := \frac{1}{a}$
    (dividing by a means multiplying by c)
  - Compute the divisor and use branches if divisor equals to a certain constant
    $\Rightarrow$ compiler can optimise computations

gain in time: 42%

Using Vector Matrices

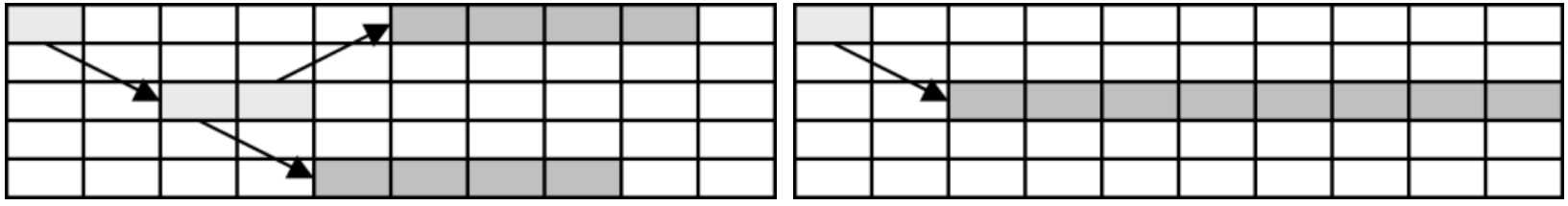

Figure: *Left* 2×4 Matrix in memory as 2D array. *Right* 2×4 Matrix in memory as 1D array.

- The rows of 2D arrays are scattered allover the memory. Prevents efficient cashing.

$\Rightarrow$ Use *1D arrays* and manage the indices in an proper way.

```
for (i=0; i<nx; i++)
  for (j=0; j<ny; j++)
    reg = mtx[i][j];
```

$\longrightarrow$

```
max = nx*ny;
for (i=0; i<max; i+=ny)
  for (j=i; j < i+ny; j++)
    reg = v[j];
```

gain in time: 22%

Other Optimisations (1)

- Loops are time consuming due to branches

⇒ *Resolving loops* to a certain extend:

```
for (i=0; i<n; i++)
  loopbody
```

⟶

```
for (i=0; i<n; i+=2){
  loopbody
  loopbody
}
```

gain in time 12%

- *Erosion and dilation* are applied using seperate procedures

⇒ Apply erosion and dilation parallel

1. Store maxima and minima from dilation/erosion process w.r.t. the columns next to each other → optimal caching
2. Use maxima/minima for determing dilation/erosion w.r.t. rows and save only the required value of these two.

gain in time 7%

Other Optimisations (2)

- Function calls are associated with branch, load and store instructions and therefore time consuming.

$\Rightarrow$ Use *inlining*

  - pays off for the routine, which determines the edge direction (see slide 8 step 3)
  - not always reasonable: long code files mean more memory load

gain in time 3.5%

- Many memory movements due to required image copies

$\Rightarrow$ Use special instructions that *copy complete memory blocks* (caution: unsafe)

gain in time 2%

- Routine uses reflected boundaries, which has to be considered in each loop.

$\Rightarrow$ *Resetting zero point*,
i.e. move array pointer after memory allocation.

gain in time 1%

Result

Overall gain in time: about 83%

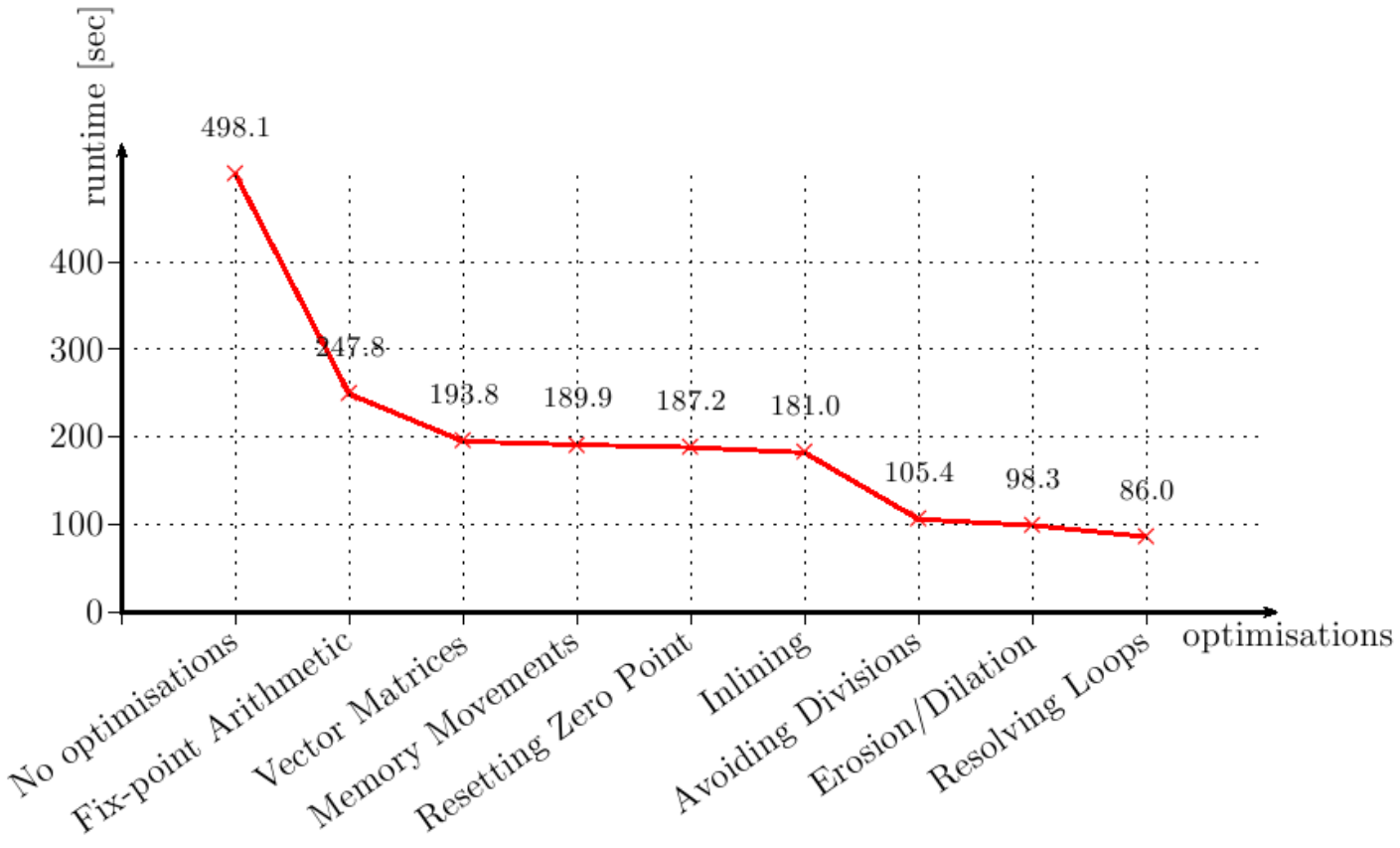

Figure: Visualisation of the *gain in time* through the applied optimisations (underlying imgage size: 410×410 pixels)

Summary

- It is possible to apply highly demanding image processing routines on a modern mobile phone!
- Original implementation needed roughly 8 minutes for an image of size 410×410 pixels
  Now: An image of this size is processed in 85 seconds
- Given highly demanding image processing routines, one can increase the running time and make them useful for many of today's devices by simple and basic code optimisations.

Yes, it is even possible to phone with this thing! But why phone?
I've got a Coherence-Enhancing Shock Filter!

Thank you!

Feel free to ask questions!

References:

Tobias Krattinger.

Hochleistungsarithmetik auf einem Mobiltelephon.

Diploma thesis, NTB Interstaatliche Hochschule für Technik Buchs, November 2006.

Joachim Weickert.

Coherence-Enhancing Shock Filters.

In G. Krell B. Michaelis, editor, *Pattern Recognition. Lecture Notes in Computer Science*, volume 2781, pages 1–8. Springer, Berlin, 2003.