Fast PDE-Based Image Analysis in Your Pocket

Andreas Luxenburger, Henning Zimmer, Pascal Gwosdek, and Joachim Weickert

Mathematical Image Analysis Group, Faculty of Mathematics and Computer Science,
Building E1 1, Saarland University, 66041 Saarbrücken, Germany
{luxenburger,zimmer,gwosdek,weickert}@mia.uni-saarland.de

Abstract. The increasing computing power of modern smartphones opens the door for interesting mobile image analysis applications. In this paper, we explore the arising possibilities but also discuss remaining challenges by implementing linear and nonlinear diffusion filters as well as basic variational optic flow approaches on a modern Android smartphone. To achieve low runtimes, we present a fast method for acquiring images from the built-in camera and focus on efficient solution strategies for the arising partial differential equations (PDEs): Linear diffusion is realised by approximating a Gaussian convolution by means of an iterated box filter. For nonlinear diffusion and optic flow estimation we use the recent *fast explicit diffusion (FED)* solver. Our experiments on a recent smartphone show that linear/nonlinear diffusion filters can be applied in realtime/near-realtime to images of size 176×144. Computing optic flow fields of a similar resolution requires some seconds, while achieving a reasonable quality.

1 Introduction

The prevailing problems in image analysis – such as solving partial differential equations (PDEs)– have widely been considered to be a challenging and computationally intensive task. If favourable results had to be computed in a reasonable time, researchers were forced to port their algorithms from desktop architectures to super-computers, which are difficult to work on, not to mention their immense costs.

In the near future, however, this trend could go in the completely opposite direction: On the algorithmic side, researchers spend more and more efforts on simple, yet efficient solution strategies. On the hardware side, the computing power of modern embedded systems such as smartphones is steadily increasing. Furthermore, powerful application development frameworks for standard programming languages ease the implementation on such platforms. Joining the ongoing work from the two mentioned research directions could thus allow to perform challenging image analysis tasks on small handheld devices that are already today in almost everybody's trouser pocket.

In order to prove the basic feasibility of this ambitious goal, the present paper shows some prototypical examples by implementing linear and nonlinear diffusion filters [1], as well as two variational optic flow approaches [2,3] on a recent smartphone (HTC Desire, 1 GHz) running the Android operating system. As users expect mobile image analysis applications to achieve interactive runtimes, we put an emphasis on efficient, but still simple to implement solvers for the occurring partial differential equations (PDEs). In the linear diffusion case, we analytically solve the PDE. This comes down to

A.M. Bruckstein et al. (Eds.): SSVM 2011, LNCS 6667, pp. 544–555, 2012.

a Gaussian convolution of the image, which is approximated by an iterated box filter [4] that achieves a realtime performance for camera images of size 240×160 pixels. In the nonlinear case where an analytical solution is not possible, we opt for an explicit solver that is speeded up by the recently proposed *fast explicit diffusion scheme* [5], resulting in a near-realtime performance. A similar explicit solver, however operating in a coarse-to-fine manner is used for optic flow estimation. Here, our implementation allows to compute flow fields on standard test sequences in the order of some seconds, while achieving a reasonable quality in terms of error measures.

Related Work. Several earlier works applied image analysis on smartphones for tasks like image enhancement and image-based applications. To our surprise there is no diffusion framework with interactive runtimes on smartphones, yet.

A general image processing framework for the Android platform including basic operations such as a box filter has been proposed by Wells [6]. A closed-source nonlinear diffusion filter is available for the iPhone [7], but it does not allow to tune any parameters and is rather slow (20 seconds for an image with 320×320 pixels). Another algorithm that shares principle properties with diffusion schemes is the coherence enhancing shock filter for the iPhone [8] that only needs about 3 seconds on a similar resolution. Recently, the OpenCV framework has been ported to Android devices [9]. It includes many filters and also computer vision methods, e.g. linear diffusion as well as a pyramidal Lucas and Kanade [10] and a Horn and Schunck optic flow algorithm [3]. Based on this framework, there is also an implementation of the combined local-global (CLG) optic flow method [2,11]. However, these approaches cannot provide the optimal runtime possible on mobile devices since OpenCV introduces an additional abstraction layer which was not optimised for particular platforms like Android. Furthermore, standard numerical solvers are used that additionally decrease the performance. Also sparse feature matching approaches based on SIFT or SURF features [12] have been considered on mobile platforms. While SIFT is too slow for interactive applications, SURF only takes about 3 ms per match [13], but does not give dense matches.

Image analysis techniques are also used in more complex applications. Before mobile phones were equipped with gyroscopic sensors, simple optic flow approaches allowed to detect the ego-motion of the phone [14], turning them into wireless pointing devices. Today, similar techniques are still of interest when a highly accurate estimation of the velocity or viewing direction is needed: Recently, a sparse feature matching algorithm was used to create freehand cylindrical panoramas on an Android phone [15]. Related techniques are also used in augmented reality applications [16] where a predefined pattern is recognised and tracked in a live stream from the camera. However, all these techniques are strongly restricted by the computational power of the device. Advanced algorithms are thus usually computed in the cloud, i.e. on remote servers [13].

Paper Organisation. In Sec. 2 we present the models and the solvers for diffusion filtering and optic flow estimation. Apart from basic Android development concepts, Sec. 3 discusses the image acquisition and further optimisations. Screenshots and a performance analysis of our application are presented in Sec. 4. We conclude in Sec. 5 by a summary and an outlook on future work.

2 Models and Solvers

2.1 Diffusion Filtering

We assume to be given a greyscale image $f(x, y) : \Omega \to \mathbb{R}$, where $(x, y)^\top \in \Omega$ denotes the location within the rectangular image domain $\Omega \subset \mathbb{R}^2$. Our goal is then to compute a gradually smoothed result $u(x, y, t) : \Omega \times [0, T] \to \mathbb{R}$, where $t \in [0, T]$ represents the evolution time of the filter, i.e. a larger evolution time leads to a stronger smoothing, and $u(x, y, 0) = f(x, y)$.

Homogeneous Diffusion. The most basic diffusion filter is a linear, homogeneous diffusion process [17] that computes the unknown u as the solution of the parabolic partial differential equation (PDE)

$$u_t \;=\; \mathrm{div}\,(\nabla u) \;=\; \triangle u \;:=\; u_{xx} + u_{yy} \;\;, \tag{1}$$

with reflecting boundary conditions. Colour images are treated channel-wise.

It is well-known that an analytical solution to (1) can be computed as $u(x, y, t) = (K_{\sqrt{2t}} * f)(x, y)$, where K_σ denotes a Gaussian of standard deviation σ, and $*$ is the convolution operator. Homogeneous diffusion filtering thus comes down to a Gaussian convolution of the given image. A straightforward way to implement this for discrete, digital images is to perform a discrete convolution with a sampled and truncated Gaussian. However, there are more efficient implementations, e.g. by a d-fold iterated box filter (IBF). This filter approximates the 2-D Gaussian kernel K_σ by a convolution

$$K_\sigma = \underbrace{B_L * B_L * \ldots * B_L}_{d\text{-times}} \quad \text{with} \quad B_L(x, y) := \begin{cases} \frac{1}{L^2}, & x, y \in [-\frac{L}{2}, \frac{L}{2}] \\ 0, & \text{else} \end{cases} \;\;, \tag{2}$$

with $L = 2\,l+1$, and $l \in \mathbb{N}$ [4]. Each B_L can be applied consecutively and is separable in space. Moreover, its implementation requires only two additions per pixel and direction using a sliding-window algorithm: The solution $\tilde{v}_k$ at position k of a 1-D signal v is given by $\tilde{v}_k = \tilde{v}_{k-1} - v_{k-l-1} + v_{k+l}$. Thus, box filters are independent from the standard deviation of the kernel. However, their runtime depends on the number of iterations d, resulting in a trade-off between the approximation error and the runtime.

Nonlinear Isotropic Diffusion. The major problem of homogeneous diffusion is the blurring of semantically important image edges as the filter performs the same smoothing at each location. To overcome this problem, Perona and Malik [1] introduced a nonlinear diffusion process

$$u_t \;=\; \mathrm{div}\,\left(g\left(|\nabla u|^2\right)\,\nabla u\right) \;=\; \partial_x\left(g\left(|\nabla u|^2\right)\,u_x\right) + \partial_y\left(g\left(|\nabla u|^2\right)\,u_y\right) \;\;, \tag{3}$$

where the decreasing diffusivity function $g(|\nabla u|^2) := (1 + |\nabla u|^2/\lambda^2)^{-1}$ reduces the smoothing at evolving edges that are indicated by large values of $|\nabla u|^2$. The parameter λ serves as a contrast parameter. Catté *et al.* [18] later proposed a regularised version where the result u (that occurs in the argument of g) is presmoothed by a Gaussian

convolution with standard deviation σ_p. This has several advantages, like reducing the staircasing artefacts and the sensitivity to noise. To process colour images, we apply the filter to each channel, but use a joint diffusivity function where we sum up the gradient magnitudes of each channel in the argument of g.

In the nonlinear case, no analytical solution exists which leaves us with computing an approximate solution by discretising the PDE. The simplest solution scheme is given by an explicit finite difference discretisation of (3) that reads as

$$\begin{aligned}\frac{u_{i,j}^{k+1}-u_{i,j}^{k}}{\tau} = & \frac{1}{h_x}\left(\frac{g_{i+1,j}+g_{i,j}}{2}\frac{u_{i+1,j}^{k}-u_{i,j}^{k}}{h_x} - \frac{g_{i,j}+g_{i-1,j}}{2}\frac{u_{i,j}^{k}-u_{i-1,j}^{k}}{h_x}\right) \\ & + \frac{1}{h_y}\left(\frac{g_{i,j+1}+g_{i,j}}{2}\frac{u_{i,j+1}^{k}-u_{i,j}^{k}}{h_y} - \frac{g_{i,j}+g_{i,j-1}}{2}\frac{u_{i,j}^{k}-u_{i,j-1}^{k}}{h_y}\right), \quad (4)\end{aligned}$$

where $u_{i,j}^{k} \approx u(i\,h_x, j\,h_y, k\,\tau)$ with h_x and h_y denoting the grid size in x- and y-direction, τ is the time step size and $g_{i,j}$ approximates the value of the diffusivity at grid point (i,j). Occurring derivatives have been discretised using standard finite difference approximations. Solving (4) for the update $u_{i,j}^{k+1}$ then gives the actual iterative scheme.

Explicit schemes are simple to implement, but stability can only be guaranteed for small time step sizes ($(\tau/h_x^2 + \tau/h_y^2) \leq 0.5$). Thus, a lot of iterations are needed to reach a reasonably large evolution time. This restriction has recently been eased in the fast explicit diffusion (FED) scheme [5]. Here, some extremely large (unstable) time steps are used in combination with some small (stable) steps. As could be shown, the combination of variable step sizes within one cycle guarantees unconditional stability of the whole scheme. This allows FED to advance faster than any other explicit scheme: While classical explicit schemes with n fixed time steps achieve a stopping time in $\mathcal{O}(n)$, FED lifts this to $\mathcal{O}(n^2)$. However, as the result is only guaranteed to be stable after a whole cycle, it is important to update the diffusivities g after the completion of a cycle, and not in between. For computing the varying FED step sizes τ_k, we use the available open source library[1].

Note that we do not discuss anisotropic diffusion filters [19] in this paper. However, they can be implemented in a similar way as the presented nonlinear isotropic filters.

2.2 Variational Optic Flow

For optic flow estimation we are given an image sequence $f(x,y,z)$, where z denotes the temporal dimension of the sequence. We further assume that the sequence has been presmoothed by a Gaussian convolution of standard deviation σ_f. We then aim at computing the flow field $\boldsymbol{w} := (u,v,1)^\top$ that describes the displacements from time z to z+1. Using a variational approach, the flow field is found by minimising an energy functional consisting of a data term that models constancy assumptions on image features, and a smoothness term that enforces the flow field to be smoothly varying in space.

[1] available at `http://www.mia.uni-saarland.de/Research/SC_FED.shtml`

The energy proposed in the seminal variational optic flow approach of Horn and Schunck [3] can be written as

$$E(u,v) = \int_\Omega \left(\boldsymbol{w}^\top J\, \boldsymbol{w} + \alpha \left(|\nabla u|^2 + |\nabla v|^2 \right) \right) \mathrm{d}x\, \mathrm{d}y \ , \tag{5}$$

using the motion tensor notation $J := (f_x, f_y, f_z)^\top (f_x, f_y, f_z)$ and where the parameter α steers the influence of the smoothness term. To minimise the above energy, we solve the corresponding elliptic Euler-Lagrange equations, which give a necessary condition for a minimiser. For the energy (5), the Euler-Lagrange equations are given by

$$0 = J_{11} u + J_{12} v + J_{13} - \alpha \triangle u \ , \tag{6}$$
$$0 = J_{12} u + J_{22} v + J_{23} - \alpha \triangle v \ , \tag{7}$$

where J_{mn} denotes the entry in row m and column n of the matrix J.

Similar to the solution of the diffusion PDEs, we solve the Euler-Lagrange equations by a stabilised, explicit gradient descent scheme which reads as

$$\frac{u_{i,j}^{k+1} - u_{i,j}^k}{\tau} = \triangle u_{i,j}^k - \frac{1}{\alpha} \left([J_{11}]_{i,j}^k \; u_{i,j}^{k+1} + [J_{12}]_{i,j}^k \; v_{i,j}^k + [J_{13}]_{i,j}^k \right) \ , \tag{8}$$
$$\frac{v_{i,j}^{k+1} - v_{i,j}^k}{\tau} = \triangle v_{i,j}^k - \frac{1}{\alpha} \left([J_{12}]_{i,j}^k \; u_{i,j}^k + [J_{22}]_{i,j}^k \; v_{i,j}^{k+1} + [J_{23}]_{i,j}^k \right) \ , \tag{9}$$

where the stabilisation is achieved by using the new value at time level $k+1$ for u and v in the first and second equation, respectively. The derivatives occurring in the expressions $[J_{mn}]_{i,j}^k$ are discretised by central finite differences. Solving the equations (8) and (9) for the unknown increments $u_{i,j}^{k+1}$ and $v_{i,j}^{k+1}$ then gives the iterative solution scheme, which we again speed up by using variable FED time step sizes. Additionally, we embed the solution in a multiscale coarse-to-fine strategy (CFED) that computes solutions on small image resolutions and uses them (after upsampling) as initialisation on the next finer scale. This results in a further speed-up as the number of required solver iterations at each level is reduced and because iterations on coarse scales are fast to compute.

If we are given colour image sequences, we sum up the data terms for all channels. This comes down to summing up the motion tensors of each channel in a joint tensor. Furthermore, to increase the robustness of our approach, we follow Bruhn *et al.* [2] and convolve the entries of the motion tensor J with a Gaussian of standard deviation ρ. This results in a combined local-global (CLG) optic flow approach.

3 Implementation on an Android Phone

3.1 Android Basics

Android is an open software platform, i.e. an operating system and software stack for different sorts of devices which has been presented by Google in 2007. It is based on a Linux Kernel and allows to build own applications through the *Android Application Framework* by using large parts of Java SE and Android-specific classes.

The building blocks of applications are `Activity` objects. They are attached to a certain `View` representing the graphical user interface (GUI). With the help of an `Intent` one can specify a certain task and trigger an activity that implements the task. In this way, one can for example capture an image from the camera within a few lines of code. The visual content of an application is organized in a hierarchy of views. Examples for views are different layouts, but also widgets such as scroll bars or check boxes. A view hierarchy is specified in an external XML layout file that assigns an ID to each view. By referring to the ID, a view can be loaded within the application as a programmable object that can be configured and attached to event listeners for user input, e.g. gestures on the touch screen.

For implementing performance-critical application parts like our diffusion filters, we use the *Native Development Kit (NDK)*. It allows to implement routines in native languages such as C/C++, resulting in a significant speedup compared to Java implementations; see our experiments in Sec. 4. The NDK also supports a set of commonly used system headers for native APIs like the `math` library. The incorporation of native code in the application uses a provided build system that lists the native source files and integrates the shared libraries into the application project. These can then be easily accessed in terms of the *Java Native Interface (JNI)* [20]. Finally, the NDK allows for optimisations to the underlying hardware by targeting specific instruction sets for the ARM platform, such as the instruction-level parallelism (NEON).

For more information on Android application development, we refer the interested reader to the excellent textbook of Meier [21].

3.2 Image Processing with Android

Instead of capturing images from the camera via a predefined intent, we use a realtime camera stream by directly accessing the camera hardware. This is achieved by using the `Camera` class and applying three steps: *(i) Image Acquisition:* Obtain the camera frames as raw byte data in YUV format and convert them into the more convenient RGB format for further processing. *(ii) Image Processing:* Apply algorithms to the RGB image (or to a greyscale version). *(iii) Visualisation:* Build a bitmap structure from the processed data and set it as content of an image overlay.

Retrieval of Camera Data. Fortunately, accessing the camera hardware in Android is rather simple and can be achieved within a few lines of code; see [21], pp. 377–381. One starts by adding a `CAMERA` permission to the application manifest, which enables to retrieve a `Camera` object by calling `Camera.open()`. A `SurfaceView` providing a dedicated drawing `Surface` can then be attached to the camera. Internal camera parameters such as the preview size and the frame rate can be retrieved and modified by a `Camera.Parameters` object. On creation and destruction of the underlying surface, the camera starts and stops its preview, respectively. Implementing the `PreviewCallback` interface, raw frame data can be obtained by calling `PreviewCallback.onPreviewFrame()`.

YUV Conversion. Each camera frame is represented in a raw byte array which stores the image row-wise and encodes color information using the YCbCr colour space. Specifically, a planar YUV format (YUV 4:2:0 (NV21)) is used where a plane of 8 bit luma (Y) samples is followed by an interleaved V/U plane containing 8 bit of 2×2 subsampled chroma samples.

For image processing applications, it is most convenient to represent the images in a planar RGB format where all colour channels are orthogonal and represented in the same resolution. Thus, the obtained raw YUV data first has to be converted before applying the respective filters. Unfortunately, the Android framework still lacks such a conversion functionality which leaves us with writing our own conversion routine. Using ARM's built-in *Vector Floating Point Architecture (VFP)* that provides hardware support for half-, single- and double-precision floating point arithmetics, we extract in our conversion routine separated, two dimensional float channels, which are aggregated into a packed 32-bit integer format (ARGB) after processing. Whereas the luminance channel alone provides data for a greyscale format, the conversion into RGB components requires computationally more expensive steps, as can be seen in the the following YUV to RGB conversion formula:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{bmatrix} 1.164 & 0.000 & 1.596 \\ 1.164 & -0.391 & -0.813 \\ 1.164 & 2.018 & 0.000 \end{bmatrix} \left(\begin{pmatrix} Y \\ U \\ V \end{pmatrix} - \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} \right) . \tag{10}$$

For an efficient implementation of the above conversion, we follow [22] and use fixpoint arithmetics and process four pixels simultaneously by moving pointers on two scan lines. However, instead of using *SIMD (Single Instruction, Multiple Data)* instructions for an efficient access to precomputed lookup tables, we directly optimise the RGB calculations. Details on these optimisations will be presented in Sec. 3.4.

3.3 Software Design

To save as much system resources as possible, we only created one Activity that incorporates our two main classes: A `FilterCamera` and a `Filter` base class. A `FilterCamera` object specifies a `FrameLayout` that is set as the main content view of the application. It subsumes all components necessary for a camera preview that are discussed in Sec. 3.2. The preview can then be overlaid by at most one `Filter` object.

We designed the diffusion filters as "cartridges" that can be plugged into the camera object and unplugged again. The base class represents a general filter object that provides a filtered image overlay, a user interface that may also be hidden, as well as a resizing option. The filter functionality itself comes with the implementation of the `PreviewCallback` interface and is specified by subclasses which represent objects for the different filters. The actual filtering routines are externalised and encapsulated within a global code library that uses native C libraries via the JNI.

3.4 Optimisations

Since the Android platform is based on a Linux kernel, it constitutes a good basis for image processing applications. Many concepts known from traditional desktop architectures carry over immediately. Moreover, the JNI allows to execute time-critical

algorithms as low-level operations. To this end, algorithms can be developed device-independently in C or C++, where the compilers provide strategies for automatic code optimisation. However, interactions with special devices such as the camera, display, or user interface still require special care and must be optimised manually.

An example for such a critical point is the visualisation of images on the display. Android supports many pixel formats including 32-bit RGBA, but it is very time consuming to transfer buffers encoded in this format to the GPU. Furthermore, our algorithms must rely on floating point accuracy to ensure the best possible approximation of the continuous model. On the other hand, the display of our smartphone can only visualise a rather small range of colour values. We thus encode and compress our results in the RGB565 format. Because this format uses only 16 bits per pixel it can be uploaded to the GPU much faster, without sacrificing visual quality. Additionally, we tried to optimise our diffusion and optic flow algorithms that take the major part of the total runtime. Common strategies like loop unrolling or reduction of reads and writes to RAM accelerated the process significantly. However, experiments indicated that no improvement can be achieved by exploiting hardware-specific extensions such as the NEON SIMD unit, probably because many operations are based on stencil operations which cause offset memory fetches. We were surprised that even purely data-parallel operations such as the YUV to RGB conversion could not be accelerated. This might be caused by the high costs for memory reads, or because the compiler by default auto-vectorises such operations already in the scalar case.

4 Experiments

4.1 Our Interactive Camera Applications

We first show screenshots of our interactive diffusion filtering application in Fig. 1. As test device we used an HTC Desire smartphone, with a Snapdragon ARMv7 CPU (1 GHz), 576 MB RAM running Android 2.2 (Froyo). The top left picture shows the application in its idle state. Besides a live preview, the GUI shows two buttons for toggling the selected filter as well as a *filter user interface (FUI)*. The latter shows several statistics like the frame rate and is displayed in the upper left corner of the GUI; see the upcoming screenshots. Furthermore, the user can save an image to the phones image gallery and can toggle the autofocus of the camera. The two following pictures show the context menus for filter and resolution selection. The second row of Fig. 1 shows filter results with linear diffusion on RGB colour images. As one can see, the filter interface offers a slider for tuning the standard deviation of the Gaussian kernel. In the settings dialog, the user can change the type of approximation by means of an iterated box filter or a discrete Gaussian convolution. The *accuracy factor* determines the number of iterations or length of the sampling interval as a multiple of the standard deviation, respectively. Moreover, a split mode can be activated, which leaves the left half of the preview unfiltered. Analogously, the third row shows results for nonlinear isotropic diffusion filtering. Here, the user can tune the contrast parameter λ and noise scale σ_p and the stopping time T. The settings dialog allows to choose among various diffusivity functions and the number of outer FED cycles M.

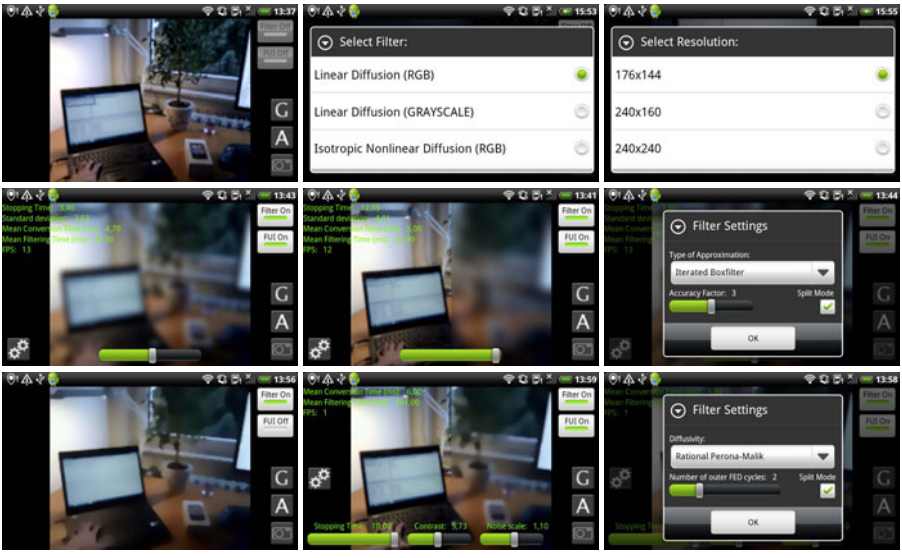

Fig. 1. Our interactive camera application. *First row:* **(a)** GUI with camera preview and controls (image gallery (G), autofocus (A) and image capture). **(b)** Filter selection. **(c)** Resolution selection. *Second row:* **(d)** Linear diffusion for RGB colour images. **(e)** Same in split-screen mode (left half remains unfiltered). **(f)** Settings dialog. *Third row:* **(g)–(i)** Same as above, but for nonlinear isotropic diffusion with the Perona Malik diffusivity [1].

Despite the efficient algorithms and the powerful computing platform, we could not achieve an optic flow estimation of reasonable quality in realtime. We thus implemented another simple application where the optic flow computation is performed in an offline process; see Fig. 2. Here, the user can pick two subsequent images of the same size from the gallery, specify the parametric settings and invoke the flow calculation. Besides the seminal approach of Horn and Schunck [3], our application also encompasses the CLG method of Bruhn *et al.* [2] where it achieves runtimes of about 10 seconds for images of size 316×252 pixels.

4.2 Performance Analysis

We now turn to a detailed performance analysis of the linear and nonlinear diffusion routines. To this end, we used simple time stamps that are placed before and after the invocation of a procedure. Resulting runtimes have been averaged over 100 frames to reduce the influence of distorting factors like background processes.

Linear Diffusion. For linear diffusion, we compare two solution strategies: *(i)* a discrete convolution with a sampled Gaussian and *(ii)* an approximation via an iterated box filter (IBF). For both strategies, we exploit separability and symmetry of the two dimensional filter masks. Additionally, we compare implementations of the two strategies in Java and native C (using the JNI). Considering the achieved runtimes shown in Table 1,

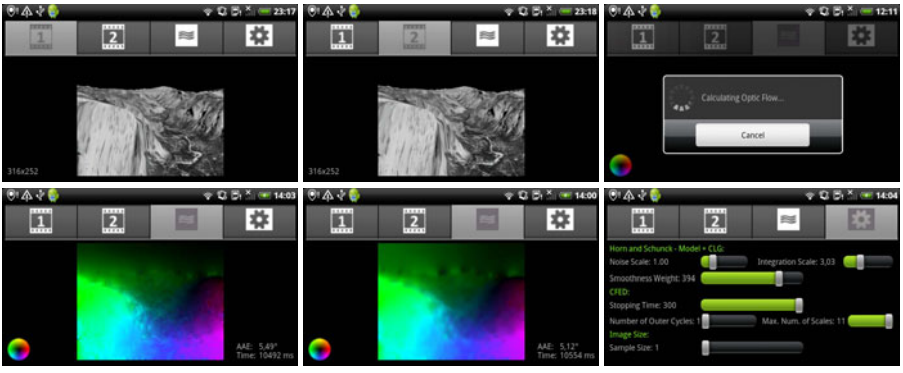

Fig. 2. Offline optic flow application. *First row:* **(a)** First image of the *Yosemite without clouds* sequence. **(b)** Second image. **(c)** Progress dialog. *Second row:* **(d)** Result with the Horn and Schunck model (CFED solver) ($\alpha = 394, \sigma_f = 1.0, T = 300, M = 1 \Rightarrow \text{AAE} = 5.49^\circ$). Flow field visualised by colour code shown in bottom left corner. **(e)** Result with the CLG model ($\rho = 3.03$, other parameters as before $\Rightarrow \text{AAE} = 5.12^\circ$). **(f)** Parameter adjustment dialog.

two observations can be made: The IBF solver is significantly faster than the Gaussian solver, and the native C implementation outperforms its Java counterpart. Furthermore, using greyscale instead of RGB images will give a speedup of a factor 3.

Table 1. Benchmark of linear diffusion on RGB images data with varying resolution and fixed stopping time ($T = 10$). We compare implementations based on a discrete Gaussian convolution with $\sigma = \sqrt{2T} \approx 4.47$ (Gauss) and an approximation via an iterated box filter (IBF). Additionally, we compare a Java to a native C implementation (using the JNI). We wish to note that using the full camera resolution of 5 MP seems infeasible for interactive/realtime applications.

	Java		C	
Resolution [px]	Gauss	IBF	Gauss	IBF
176×144	370 ms	91 ms	361 ms	49 ms
240×160	571 ms	168 ms	564 ms	88 ms
320×240	1174 ms	444 ms	1141 ms	241 ms

Nonlinear Isotropic Diffusion. In the nonlinear diffusion case, we spent some efforts to analyse the runtime fractions of the different processing steps; see Fig. 3. As one can see, the major part of the computation time (98%) is spent for the FED scheme. This is good news as it shows that the runtime of the pre- and post-processing steps (image acquisition, YUV conversion, visualisation, etc.) can be neglected and no further optimisations are required in this respect. It is thus more interesting to further analyse the runtime fractions in the FED scheme itself. Interestingly, the update of the nonlinear operator which comes down to computing the diffusivities after each FED cycle consumes 18% of the time. The rest is spent for the actual iteration steps. As the nonlinear update consumes a considerable amount of time, we also analysed its building blocks,

which are a presmoothing that takes 68% of the time and the computation of the diffusivity. Note that the presmoothing is already efficiently realised by two IBF iterations. With a naive implementation, the fraction of the presmoothing would be even higher.

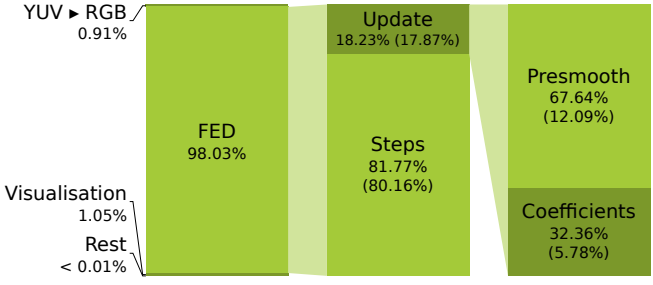

Fig. 3. Runtime analysis for nonlinear Perona-Malik diffusion on RGB images (176×144 pixels, $M = 2$, $\lambda = 4$, $\sigma_p = 1.0$, $T = 10$, resulting in 2 inner FED steps). Brackets give fraction w.r.t. overall running time of 588.1 ms.

5 Conclusions and Outlook

Our paper showed that recent smartphones offer a promising platform for the development of challenging mobile image analysis applications. As a *proof-of-concept*, we presented efficient implementations of linear and nonlinear diffusion as well as basic variational optic flow methods on an Android smartphone (HTC Desire). A main observation is that a careful choice of the solver for the arising PDEs is a key to good performance. For linear diffusion, we used a classical iterated box filter, whereas we opted for the recent FED solver and its coarse-to-fine variant in the context of nonlinear diffusion and optic flow, respectively. These solvers allow for small runtimes and are simple enough to be easily implemented as well as optimised on a mobile platform.

We hope that our work sparks the development of further interesting image analysis applications on mobile devices like smartphones. Here, efficient denoising methods are interesting because the small image sensors and the simple camera optics are prone to noise, especially in low light conditions. Moreover, using optic flow algorithms allows to port challenging computational photography applications like panorama stitching or high dynamic range imaging to mobile platforms.

Acknowledgements. The authors gratefully acknowledge partial funding by the cluster of excellence 'Multimodal Computing and Interaction' (MMCI).

References

1. Perona, P., Malik, J.: Scale space and edge detection using anisotropic diffusion. IEEE Transactions on Pattern Analysis and Machine Intelligence 12, 629–639 (1990)
2. Bruhn, A., Weickert, J., Schnörr, C.: Lucas/Kanade meets Horn/Schunck: Combining local and global optic flow methods. International Journal of Computer Vision 61, 211–231 (2005)

3. Horn, B., Schunck, B.: Determining optical flow. Artificial Intelligence 17, 185–203 (1981)
4. Wells, W.M.: Efficient synthesis of Gaussian filters by cascaded uniform filters. IEEE Transactions on Pattern Analysis and Machine Intelligence 8, 234–239 (1986)
5. Grewenig, S., Weickert, J., Bruhn, A.: From box filtering to fast explicit diffusion. In: Goesele, M., Roth, S., Kuijper, A., Schiele, B., Schindler, K. (eds.) DAGM 2010. LNCS, vol. 6376, pp. 533–542. Springer, Heidelberg (2010)
6. Wells, M.T.: Mobile image processing on the Google phone with the Android operating system (2009), http://www.3programmers.com/mwells/documents/pdf/Final (retrieved 2011-01-06)
7. GMA3: Moon filter (2010), http://itunes.apple.com/en/app/moon-filter/id387317833, (retrieved 2011-01-06)
8. Gogolok, R., Steinel, A.: Shockmypic (2009), http://www.shockmypic.com/iphone/ (retrieved 2011-01-06)
9. Bradski, G., Kaehler, A.: Learning OpenCV: computer vision with the OpenCV library. O'Reilly, Sebastopol (2008)
10. Bouguet, J.Y.: Pyramidal implementation of the Lucas Kanade feature tracker – description of the algorithm (2000), http://trac.assembla.com/dilz_mgr/export/272/doc/ktl-tracking/algo_tracking.pdf(retrieved 2011-01-06)
11. Harmat, A.: Variational optic flow (2010), http://sourceforge.net/projects/varflow/ (retrieved 2011-01-06)
12. Bay, H., Tuytelaars, T., Van Gool, L.: SURF: Speeded up robust features. In: Bischof, H., Leonardis, A., Pinz, A. (eds.) ECCV 2006, Part I. LNCS, vol. 3951, pp. 404–417. Springer, Heidelberg (2006)
13. Olsson, S., Åkesson, P.: Distributed mobile computer vision and applications on the Android platform. Master's thesis, Faculty of Engineering, Lund University, Sweden (2009)
14. Ballagas, R., Rohs, M., Sheridan, J.G.: Mobile phones as pointing devices. In: Rukzio, E., Hakkila, J., Spasojevic, M., Mäntyjärvi, J. (eds.) Proc. 2005 Pervasive Mobile Interaction Devices, Munich, Germany, vol. 6, pp. 1–4 (2005)
15. Wagner, D., Mulloni, A., Langlotz, T., Schmalstieg, D.: Real-time panoramic mapping and tracking on mobile phones. In: Lok, B., Klinker, G., Nakatsu, R. (eds.) Proc. IEEE Virtual Reality Conference 2010, Waltham, MA, pp. 211–218 (2010)
16. Wagner, D., Schmalstieg, D., Bischof, H.: Multiple target detection and tracking with guaranteed framerates on mobile phones. In: Proc. of IEEE Int. Symposium on Mixed and Augmented Reality 2009, Orlando, FL (2009)
17. Iijima, T.: Basic theory of pattern observation. In: Papers of Technical Group on Automata and Automatic Control. IECE, Japan (1959) (in Japanese)
18. Catté, F., Lions, P.L., Morel, J.M., Coll, T.: Image selective smoothing and edge detection by nonlinear diffusion. SIAM Journal on Numerical Analysis 32, 1895–1909 (1992)
19. Weickert, J.: Anisotropic Diffusion in Image Processing. Teubner, Stuttgart (1998)
20. Liang, S.: Java Native Interface: Programmer's Guide and Reference. Addison–Wesley, Boston (1999)
21. Meier, R.: Professional Android 2 Application Development. Wrox Press Ltd., Birmingham (2010)
22. Dupuis, E.: Optimizing YUV–RGB color space conversion using Intel's SIMD technology (2003), http://lestourtereaux.free.fr/papers/data/yuvrgb.pdf, (retrieved 2011-01-07)