K. Pauwels, M. Tomasi, J. Diaz Alonso, E. Ros and M. M. Van Hulle, "A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features," in *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 999-1012, July 2012. doi: 10.1109/TC.2011.120
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5936059&isnumber=6204246

A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features

Karl Pauwels, Matteo Tomasi, Javier Díaz, Eduardo Ros, and Marc M. Van Hulle, *Senior Member*, *IEEE*

Abstract—Low-level computer vision algorithms have extreme computational requirements. In this work, we compare two real-time architectures developed using FPGA and GPU devices for the computation of phase-based optical flow, stereo, and local image features (energy, orientation, and phase). The presented approach requires a massive degree of parallelism to achieve real-time performance and allows us to compare FPGA and GPU design strategies and trade-offs in a much more complex scenario than previous contributions. Based on this analysis, we provide suggestions to real-time system designers for selecting the most suitable technology, and for optimizing system development on this platform, for a number of diverse applications.

Index Terms—Reconfigurable hardware, graphics processors, real-time systems, computer vision, motion, stereo.

1 Introduction

Low-level vision engines constitute a crucial intermediate step toward a fully symbolic interpretation of the visual environment by transforming pixel-based intensity values into a more meaningful description, such as correspondences between images. In most current vision systems, the low-level component summarizes the visual signal into a sparse set of interesting features [1] (e.g., corners and/or edges) and restricts further processing, such as correspondence finding, to this condensed representation. Such an approach ignores large parts of the visual signal (e.g., textured regions) and is often motivated by computational resource limitations. Recent advances in massively parallel hardware now make it feasible to instead establish reliable correspondences for most pixels in real time by processing the signal in its entirety. In this work, we compare two such real-time vision architectures, one developed on a Field-Programmable Gate Array (FPGA) and the other on a Graphics Processing Unit (GPU). Both architectures extract dense optical flow (OF), dense stereo, and local image features (energy, orientation, and phase) on the basis of a Gabor wavelet decomposition [2]. These engines have numerous applications in robot vision [3], [4], [5], motion analysis [6], [7], and image (sequence) processing [8], [9].

- *K. Pauwels and M.M. Van Hulle are with the Laboratorium voor Neuro- en Psychofysiologie, K.U.Leuven, Afdeling Neurofysiologie, O&N II Herestraat 49 - bus 1021, Leuven 3000, Belgium. E-mail: {karl.pauwels, marc.vanhulle}@med.kuleuven.be.*
- *M. Tomasi, J. Díaz, and E. Ros are with the Computer Architecture and Technology Department, University of Granada, E.T.S.I. Informática y Telecomunicación, CITIC, C/ Periodista Daniel Saucedo, s/n, Granada 18071, Spain. E-mail: {mtomasi, jdiaz, eduardo}@atc.ugr.es.*

Manuscript received 7 Jan. 2011; revised 7 June 2011; accepted 13 June 2011; published online 22 June 2011.
Recommended for acceptance by T. El-Ghazawi.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-01-0012.
Digital Object Identifier no. 10.1109/TC.2011.120.

1.1 Related Real-Time Approaches

Due to the abundance of optical flow and dense stereo estimation methods, we only review a selection of recent real-time implementations. Both for optical flow and stereo, one can, broadly speaking, distinguish between local and global methods. The former only use image information in a small region surrounding the pixel whereas the latter enforce additional constraints on the estimates (such as spatial or spatiotemporal smoothness) [10], [11]. Local methods are easier to implement efficiently in parallel architectures and real-time implementations exist on a variety of platforms (PC (CPU) [12], [13], FPGA [14], [15], and GPU [16], [17]). Although they are more accurate, global methods are more difficult to parallelize and real-time performance can only be achieved at low resolutions or through a variety of algorithmic simplifications [18], [19], [20]. A multiscale coarse-to-fine control scheme [21] is commonly applied in real-time implementations to efficiently extend the dynamic range of both local and global methods. This work considers local coarse-to-fine phase-based methods that exhibit an increased robustness compared to other real-time local methods (for a detailed discussion, see [2]).

1.2 Related Architecture Comparisons

Platform selection is a crucial stage in system development and many studies aim to facilitate this process (see [22] for a review). In an early study [23], the GPU is outperformed by the FPGA in terms of required clock cycles on three different applications: Gaussian Elimination, Data Encryption Standard, and Needleman-Wunsch sequence alignment. Biological sequence alignment is also considered in [24], but here the focus is mainly on a Network-on-Chip implementation that greatly outperforms the other architectures. In [25], CPU, GPU, and FPGA are compared on 3D tomography computation. The GPU obtains the highest absolute performance, but the FPGA has again smaller clock cycle requirements. Another study concludes that

each platform requires a different approach for random number generation [26]. The FPGA outperforms the GPU three times in absolute performance, and even by an order of magnitude when power consumption is considered. Note however that combining random number generation with, e.g., a Monte-Carlo application can greatly increase the computation-to-memory ratio, which is favorable for the GPU. In [27], Monte-Carlo simulation, FFT, and weighted sum operations are evaluated on FPGA, GPU, and Playstation 2 using a unified source description based on stream compilers. This work shows the high performance of the GPU and the low-level tuning required to achieve high performance on the FPGA. A similar conclusion has been reached more recently on the basis of a Quantum Monte-Carlo study [28]. Another contribution [29] uses NVIDIA's Compute Unified Device Architecture (CUDA) paradigm [30] as a starting point for FPGA design. With this unified language and design flow (labeled FCUDA), the authors obtain similar performance in FPGAs and GPUs.

In the framework of image processing, Asano et al. [31] found that FPGAs deliver the most performance in complex applications (local window-shifting stereo and k-means clustering) and GPUs in simple computations (2D convolution). Using a (simple) optical flow algorithm, Chase et al. [32] obtained similar performance with FPGA and GPU, but the FPGA required a 12 times longer design time. Finally in [33], a systematic approach is presented for the comparison of GPU and FPGA using a variety of image processing algorithms: color correction, 2D convolution, video frame resizing, histogram equalization, and motion vector estimation. As compared to the present study, the work in [33] uses simpler algorithms, older technology, and does not consider accuracy, resource usage, or development effort. The complexity of the algorithms employed here and the more extensive evaluation allow us to provide new suggestions and more sophisticated design choices for the on-chip implementation of highly complex computer vision models.

1.3 Novelty

Most previous comparison studies focus on performance alone and provide only qualitative descriptions of supported arithmetic representation, design time, system cost, and target applications. Our work goes beyond previous comparisons (such as [33]) and provides a number of original contributions.

1. We have developed both systems using languages and tools that provide a high level of abstraction. For the GPU, we have used CUDA as opposed to low-level description languages such as Parallel Thread Execution. For the FPGA, we have used Handel-C from Mentor [34], except for critical low-level system (PCIe) or memory controllers for which VHDL and specific IP modules enable better control.
2. Previous comparison studies examine different algorithms in isolation. The use of high-level languages in this work has not only enabled the development of much more complex vision modules, but also their integration into a single low-level vision engine. This allows for a much deeper exploration of the design trade-offs of each platform. The optical flow module's multiscale architecture is of a complexity that is rarely seen in FPGA implementations [35]. We have now replicated and integrated this module with other low-level modules to arrive at a novel hardware architecture that achieves a very high throughput as compared to previous contributions [36]. The integration of different vision modules has been addressed before in FPGAs [37], but considering much simpler algorithms. The GPU architecture originates from [3] but has been extensively described, analyzed, and optimized in this work, with significantly improved performance as a result.
3. In addition to design time, system cost, and power consumption, our evaluation also includes quantitative evaluations of external and on-chip memory bandwidth and size requirements, arithmetic complexity and representation, final system accuracy and speed, as well as a discussion of embedded capabilities and certification issues. Furthermore, we also consider the scheduling of processing units, techniques for hardware sharing, and accuracy versus resource utilization trade-offs. These topics are seldom addressed in previous studies and significantly bias the design toward one or the other technology.
4. We provide a set of guidelines for selecting the most suitable target technology for various computer vision algorithms as well as for code optimization and possible algorithmic modifications required by each device. Our main focus is on absolute (unnormalized) system performance (for cost, speed, development time, etc.). In previous studies, performance values have often been normalized by clock frequency, power consumption, die area, etc. [23], [26], [33]. Although we will also address this issue and we understand its importance as an attempt to abstract architectural and technological issues, we feel that for real-world applications, absolute values provide the proper metric for comparison.

1.4 Overview

The different modules of the vision engine are discussed step by step, in order of increasing (model) complexity: Gabor filtering in Section 2, local image features in Section 3, stereo in Section 4, and optical flow in Section 5. Each of these sections first explains the algorithms and then discusses the GPU and FPGA implementations and (possible) algorithmic simplifications. Due to space constraints, the model descriptions are kept brief. However, MATLAB implementations are provided in the supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2011.120. The reader is encouraged to consult [2] for more in-depth discussions and motivations. The integration of the different modules is discussed in Section 6. Next, Section 7 addresses the comparison of both architectures. The results are discussed in Section 8 and the paper is concluded in Section 9.

2 Gabor Filtering

All the algorithms of the low-level vision engine operate on the responses of a filterbank of quadrature pair Gabor filters

tuned to different orientations and different scales. We use the same filterbank as described in [2]. This filterbank consists of $N = 8$ oriented complex Gabor filters. The different orientations, θ_q, are evenly distributed and equal to $\frac{q\pi}{N}$, with q ranging from 0 to $N-1$. For a specific orientation θ_q, the 2D complex Gabor filter at pixel location $\mathbf{x} = (x, y)^{\mathrm{T}}$ equals

$$f_q(\mathbf{x}) = e^{-\frac{x^2+y^2}{2\sigma^2}} e^{j\omega_0(x\cos\theta_q + y\sin\theta_q)}, \quad (1)$$

with peak frequency ω_0 and spatial extension σ. The filterbank has been designed with efficiency in mind and relies on 11×11 separable spatial filters that are applied to an image pyramid [38]. The peak frequency is doubled from one scale to the next. At the highest frequency, we use a 4-pixel period. The filters are separable and by exploiting symmetry considerations, all 16 responses can be obtained on the basis of only 24 1D convolutions with 11 tap filters [2] (see also Section 2.1). The filter responses, obtained by convolving the image, $I(\mathbf{x})$, with the oriented filter (1) can be written as

$$Q_q(\mathbf{x}) = (I * f_q)(\mathbf{x}) = \rho_q(\mathbf{x}) e^{j\,\phi_q(\mathbf{x})} = C_q(\mathbf{x}) + j\,S_q(\mathbf{x}). \quad (2)$$

Here, $\rho_q(\mathbf{x}) = \sqrt{C_q(\mathbf{x})^2 + S_q(\mathbf{x})^2}$ and $\phi_q(\mathbf{x}) = \mathrm{atan2}(S_q(\mathbf{x}), C_q(\mathbf{x}))$ are the amplitude and phase components, and $C_q(\mathbf{x})$ and $S_q(\mathbf{x})$ are the real and imaginary responses of the quadrature filter pair. The $*$ operator depicts convolution. Note that the use of $\mathrm{atan2}$ as opposed to atan doubles the range of the phase angle. As a result, correspondences can be found over larger distances.

2.1 GPU Implementation

The image pyramid is constructed by smoothing with a five-tap low-pass filter and subsampling [2]. A GPU thread is launched for each output pixel. The smoothing filter is separable, but higher performance is achieved by filtering directly with the 2D filter since separable filtering requires intermediate external memory storage. The image data are accessed through the texture units, which provide a caching mechanism for efficient reuse of neighboring elements.

The 11-tap separable Gabor filters are larger than the low-pass filter and in this case, two stage separable filtering is much more efficient (three to four times faster). At each scale, the filterbank responses are obtained using the two GPU kernels[1] shown in Fig. 1 (g_x and g_y are horizontal and vertical 1D Gaussians, respectively). Kernel A performs all column convolutions and kernel B performs all row convolutions and combines the convolution outputs. In kernel A, the combination of multiple convolutions that operate on the same image data dramatically increases the computation-to-memory ratio. As before, both kernels read data from (cached) texture memory. Note that we perform one more convolution than mentioned in Section 2. To save this additional convolution, g_x would have to be performed in kernel A, but the latter only supports column convolutions. After combining the row convolution results, kernel B interleaves the even and odd responses in external memory. By storing filter responses rather than phase, phase wrap-around problems can be avoided in the subsequent warping operations.

1. A GPU kernel is a function executed on the GPU device by many threads running on different processors of the multiprocessors.

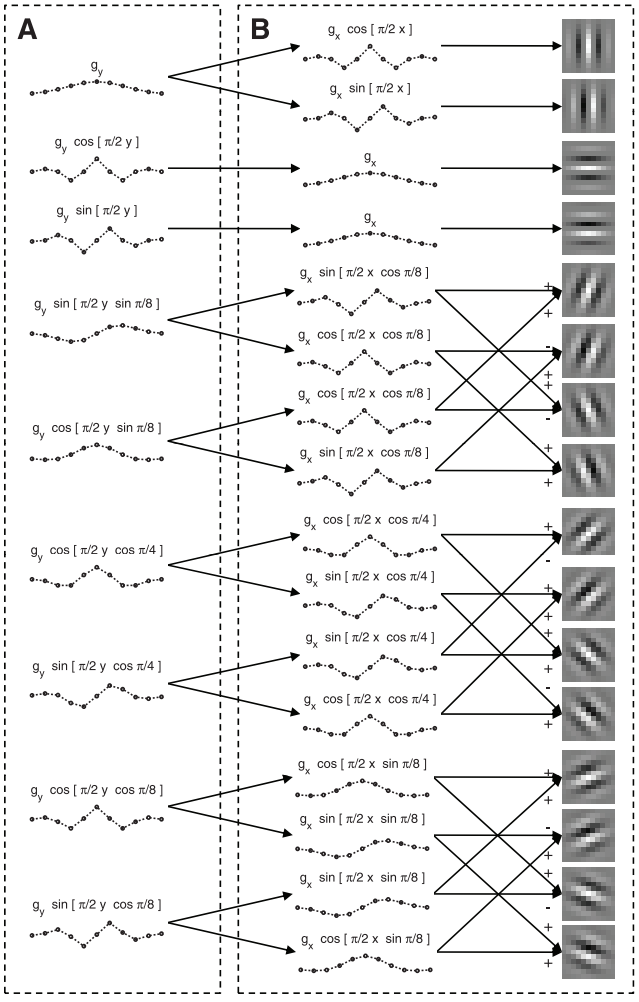

Fig. 1. GPU kernels used to obtain the (single scale) Gabor filterbank responses. (A) Column filter. (B) Row filter.

2.2 FPGA Implementation

The hardware description in reconfigurable devices remains a problematic issue for complex mathematical algorithms, but modern FPGAs incorporate embedded resources that facilitate architecture design and optimization. The embedded multipliers and adders in high-performance DSPs enable great speedups in convolutions, and small internal RAMs can be used as circular buffers to cache image rows for local processing. The presence of many parallel processing units and the low delays for paths between them are some of the major benefits of FPGAs. All these embedded and parallel resources are exploited in the convolutions of the Gabor filtering stage. The image pyramid is built first by an iterative reduction circuit. Gabor filtering is applied after this operation has completed and just before further processing on each scale. Contrary to the GPU, the FPGA stores the images rather than the filter outputs. This requires replicating the filtering module for each input image. As we will see later, we process a left and right image, and a three-frame sequence, so five cores are required in total. Since the filters are 11×11 pixels in size, 11 embedded multiport RAMs are used to store the 11 image rows that have been most recently used in the convolution. The filters themselves are stored in wired

memory at initialization time. After a first latency equal to 5.5 rows, the module produces 1 pixel per clock cycle.

Instead of optimized floating point units and because of the complex logic required to manage large bit widths, we have chosen a constrained fixed-point arithmetic. As shown in [33], over a 12-fold increase in logic density is required for the FPGA to be comparable with the GPU and support floating point arithmetic. Therefore, all variables are limited in their fractional part from the Gabor filtering stage and onward.

3 Local Image Features

Using a tensor-based method, the Gabor filter responses across orientation can be combined into the local energy, E_{local}, orientation, θ_{local}, and phase at this orientation, ϕ_{local} [2]:

$$E_{\text{local}} = \sum_{q=0}^{N-1} \rho_q^2, \tag{3}$$

$$\theta_{\text{local}} = \frac{1}{2}\arg\left(\sum_{q=0}^{N-1} \rho_q e^{2j\theta_q}\right), \tag{4}$$

$$\phi_{\text{local}} = \operatorname{atan2}(S, C), \tag{5}$$

where

$$S = \sum_{q=0}^{N-1} S_q \rho_q^2 \cos(\theta_q - \theta_{\text{local}}), \tag{6}$$

$$C = \sum_{q=0}^{N-1} C_q \rho_q^2 |\cos(\theta_q - \theta_{\text{local}})|. \tag{7}$$

The energy measure provides an indication of where interesting features (lines or corners) are situated and the orientation and phase measures describe and identify the type of feature.

3.1 GPU Implementation

In this stage, each GPU thread operates on a single pixel and processes the different orientations sequentially. Not all local image features can be computed in a single run across orientation because the weighted filter responses in (6) and (7) depend on the local orientation (4). We do however have sufficient register space to store $S_q\rho_q^2$ and $C_q\rho_q^2$ (for all q) during a first run across orientation. After this first run also the local energy (3) and orientation (4) become available, which allows for computing S and C using (6) and (7) and finally also the local phase (5). Each scale is processed sequentially by repeating the same kernel.

3.2 FPGA Implementation

The data dependency resulting from the response weighting in (6) and (7) is removed in the FPGA implementation by computing these terms as $S = \sum_{q=0}^{N-1} S_q$ and $C = \sum_{q=0}^{N-1} C_q$ [39]. Each Gabor filter output is sent along three different paths: local energy, orientation, and phase. These paths are synchronized through a specific retiming mechanism that delays faster processes. IP cores provided by the CORDIC implementation of the Xilinx Core Generator [40] are used for square root and arctangent calculations. The local features (LF) are calculated for each scale and stored in external RAM. Each word in memory contains the information of four different pixels using a fixed-point format with 9 bits per feature. The local features circuit adds latency to the Gabor filtering stage, but does not affect throughput.

4 Phase-Based Stereo

Stereo disparity (D) estimates can be obtained efficiently from the phase difference between the left and right images [2]. For oriented filters, the phase difference has to be projected on the epipolar line. Since we assume rectified images, this is equal to the horizontal. For a filter at orientation θ_q, a disparity estimate is then obtained as follows:

$$\delta_q(\mathbf{x}) = \frac{\left[\phi_q^L(\mathbf{x}) - \phi_q^R(\mathbf{x})\right]_{2\pi}}{\omega_0 \cos\theta_q}, \tag{8}$$

where the $[\,]_{2\pi}$ operator depicts reduction to the $]-\pi;\pi]$ interval. These different estimates are robustly combined using the median. To reduce noise, a subsequent 3×3 median filtering is performed that outputs the median if the majority of its inputs are valid; otherwise, it signals an invalid estimate. Due to phase periodicity, the phase difference approach can only detect shifts up to half the filter wavelength. To compute larger disparities, the estimates obtained at the different pyramid levels are integrated by means of a coarse-to-fine control strategy [21]. A disparity map $\delta^k(\mathbf{x})$ is first computed at the coarsest level k. To be compatible with the next level, it is upsampled, using an expansion operator $\mathcal{X}$, and multiplied by two:

$$d^k(\mathbf{x}) = 2 \cdot \mathcal{X}\big(\delta^k(\mathbf{x})\big). \tag{9}$$

This map is then used to reduce the disparity at level $k+1$, by warping the right filter responses before computing the phase difference:

$$\delta_q^{k+1}(\mathbf{x}) = \frac{\left[\phi_q^L(\mathbf{x}) - \phi_q^R(\mathbf{x}')\right]_{2\pi}}{\omega_0 \cos\theta_q} + d^k(\mathbf{x}), \tag{10}$$

where

$$\mathbf{x}' = (x + d^k(\mathbf{x}),\ y)^{\mathrm{T}}. \tag{11}$$

In this way, the remaining disparity is guaranteed to lie within the filter range. This procedure is repeated until the finest level is reached. Note that the median filter is applied at each scale.

4.1 GPU Implementation

As in the local features stage, each thread in the stereo implementation operates on a single pixel and processes orientations sequentially. The kernel is repeatedly applied at each scale, starting from the coarsest. The texture hardware is used to upsample the previous scale stereo estimates (9). This estimate provides the coordinates (11) for the texture fetch of the right frame filter responses (10). Although this warping transformation cannot be

predicted (it depends on the image contents), spatial locality is high, and the texture cache enables a high bandwidth in this data-intensive stage of the algorithm. In addition, the texture units also provide low precision bilinear interpolation to resolve noninteger coordinates and increase the precision of the warping operations at no additional cost. After completing a scale, the stereo estimates are assigned to the texture units and processed in a median filter kernel. Due to the possibility of invalid estimates, a variable number of estimates need to be sorted in this kernel. To simplify this process, we first replace invalid estimates alternately with a very large and very small number and continue with a standard sorting routine. In this way, missing values have a negligible effect on the estimated median and the kernel remains simple. After median filtering, the texture references are mapped to the Gabor filter responses at the next scale, and the GPU kernel is repeated. Shared memory is not used in this implementation.

4.2 FPGA Implementation

The FPGA also processes the different scales sequentially, starting at the coarsest level. Unlike in the GPU, filter response warping is infeasible due to the smaller number of memory banks, and the much slower clocked memory. Instead, the images are warped before the Gabor filtering. This greatly reduces memory access. Furthermore, since the warping is 1D in the stereo case, the rows can be stored in embedded multiport RAMs, and external memory access can be avoided entirely. The single scale stereo core consists of three stages that are reused to process all scales: Gabor filtering (shared with the local features stage), phase difference calculation, and median estimation. In the phase difference computation, we again use the arctangent cores from Xilinx, while a tree-based architecture is used for sorting in the median circuit. The regularizing 3×3 spatial median filtering is performed at the end of each scale. The final stereo estimates are stored in a fixed-point format with 8 and 4 bits for the integer and fractional parts, respectively. Further details of the stereo architecture can be found in [41].

5 Phase-Based Optical Flow

In a similar fashion as stereo disparity can be obtained from the phase difference between left and right images, optical flow can be obtained from the evolution of phase in time [42]. We use the algorithm by Gautama and Van Hulle [43], which can exploit multiple image frames.

Points on an equiphase contour satisfy $\phi(\mathbf{x}, t) = c$, with c a constant. Differentiation with respect to time yields:

$$\nabla\phi \cdot \mathbf{v} + \psi = 0, \tag{12}$$

where $\nabla\phi = (\delta\phi/\delta x\ ,\ \delta\phi/\delta y)^{\mathrm{T}}$ is the spatial phase gradient, $\mathbf{v} = (v_x, v_y)^{\mathrm{T}}$ the optical flow vector, and $\psi = \delta\phi/\delta t$ the temporal phase gradient. Due to the aperture problem, only the velocity component along the spatial phase gradient can be computed (normal flow). Under a linear phase model, the spatial phase gradient can be substituted by the radial frequency vector, $\omega_0(\cos\theta_q\ ,\ \sin\theta_q)$. In this way, the component velocity, $\mathbf{c}_q(\mathbf{x})$, can be estimated directly from the temporal phase gradient, $\psi_q(\mathbf{x})$:

$$\mathbf{c}_q(\mathbf{x}) = -\frac{\psi_q(\mathbf{x})}{\omega_0}\ (\cos\theta_q\ ,\ \sin\theta_q). \tag{13}$$

At each location, the temporal phase gradient is obtained from a linear least-squares fit to the model:

$$\hat{\phi}_q(\mathbf{x}, t) = a + \psi_q(\mathbf{x})t, \tag{14}$$

where $\hat{\phi}_q(\mathbf{x}, t)$ is the unwrapped phase. We typically use five frames in this estimation. The intercept of (14) is discarded and the reliability of each component velocity is measured by the mean squared error (MSE) of the linear fit. Each component velocity $\mathbf{c}_q(\mathbf{x})$ provides the linear constraint (12) on the full velocity:

$$v_x(\mathbf{x}) \cdot \omega_0 \cos\theta_q + v_y(\mathbf{x}) \cdot \omega_0 \sin\theta_q + \psi_q(\mathbf{x}) = 0. \tag{15}$$

The constraints provided by several component velocities need to be combined to estimate the full velocity. Provided a minimal number of component velocities at pixel $\mathbf{x}$ are reliable (their MSE is below a threshold, τ_l, the phase linearity threshold), they are integrated into a full velocity by solving the overdetermined system of (15) in the least-squares sense. As in Section 4, a 3×3 spatial median filter is applied (separately to each optical flow component) to regularize the estimates. Next, a coarse-to-fine control scheme is used to integrate the estimates over the different pyramid levels [9]. Starting from the coarsest level k, the optical flow field $\mathbf{v}^k(\mathbf{x})$ is computed, median filtered, expanded, and used to warp the phase at the next level, $\phi^{k+1}(\mathbf{x}', t)$, as follows:

$$\mathbf{x}' = \mathbf{x} - 2 \cdot \mathbf{v}^k(\mathbf{x}) \cdot (3 - t). \tag{16}$$

This effectively warps all pixels in the five-frame sequence to their respective locations in the center frame (frame 3).

5.1 GPU Implementation

The Gabor pyramid is traversed in the same way as in the stereo algorithm, but the previous scale optical flow estimates are now used to warp the Gabor filter responses of the two frames before and the two frames after the center frame in the buffer (16). Only a very small amount of temporary storage is required to solve the linear least-squares systems and the use of shared memory can again be avoided. A median filter kernel similar to the one discussed in Section 4.1 is applied to the estimates before proceeding to the next scale.

5.2 FPGA Implementation

As in the stereo module, the warping operates on images rather than phase or filter outputs. In addition, the temporal sequence is reduced from five to three frames to save two Gabor filter modules and additional external memory requirements. Unlike in the stereo case, the image warping is now 2D and requires random external memory access. Due to the sequential nature of this memory, a throughput of 1 pixel per clock in the warping stage can only be guaranteed by storing a 2×2 window for each pixel, four times increasing the memory requirements. An iterative median filter regularization is performed for this circuit as

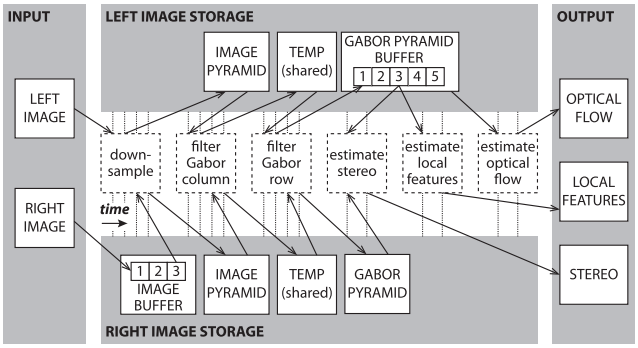

Fig. 2. GPU system overview showing the (sequential) processing stages (dashed boxes) and their interaction with GPU memory (gray).

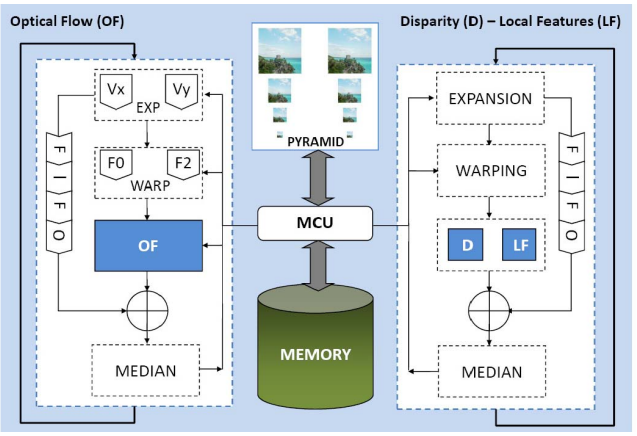

Fig. 3. FPGA system overview showing the main processing cores (blue): optical flow, disparity, and local features.

well. The final estimates are again stored in a fixed-point format with 8 and 4 bits for, respectively, the integer and fractional parts.

The single scale optical flow core is the most complicated part of the system and contains over 900 parallel processing units and 80 pipeline stages. It is described in more detail in [35].

6 System Integration

6.1 GPU System Description

The GPU system has been implemented on an NVIDIA Geforce GTX 280 [30]. It is a massively parallel processor equipped with large amounts of high bandwidth (but large latency) memory. Programming and debugging is facilitated by the extended C language provided by the CUDA programming framework that exposes the GPU's streaming processors and texture hardware.

6.1.1 System Overview

Fig. 2 provides an overview of the sequential processing stages (dashed boxes) and their interaction with GPU memory (gray). The downsampling and filtering stages first process the left and then the right image. The temporary storage used in between the Gabor column and row filtering operations is reused for the right image. A buffer is used to store the Gabor pyramid responses for the five most recent frames of the left sequence. Each of these pyramids is used five times by the optical flow algorithm. An image buffer delays the right images to synchronize the input to the stereo algorithm. Since the left Gabor pyramid of the third frame is required by the local features, stereo and optical flow stages, memory access could be reduced by combining these stages into a single GPU kernel. This does not improve performance though since, as we will see in Sections 7.1 and 7.2, all these kernels are compute- rather than memory bound.

6.1.2 Algorithm Modifications

No algorithm simplifications are required in the GPU implementation. Some minor differences in the precision of the estimates with respect to a CPU implementation do occur (see Section 7.4) due to the single precision floating point representation, and the use of the texture units for bilinear interpolation in the warping operations. These texture units use only 8 bits fractional value for the interpolation weights [44]. The image rather than filter response warping simplification adopted in the FPGA architecture is not beneficial for the GPU, since it requires filtering the images multiple times and the filtering stage requires a substantial amount of time (see Section 7.1). Therefore, the GPU implementation addresses a more complex model and this will have a slight impact on the final system accuracy of the GPU approach versus the simpler FPGA approach.

6.2 FPGA System Description

Every processing module of the FPGA system has been developed and debugged independently on a XircaV4 platform from SevenSolutions [45] equipped with a Virtex 4 xc4vfx100 and four external SDRAM banks that allow parallel access and thus a higher bandwidth. The Virtex 4 system has limited resources and can only fit the entire system if additional simplifications or resource sharing are adopted. Therefore, the fully parallel architecture (which can process 1 pixel per clock along scales) has been synthesized for a Virtex 5 architecture. This architecture is of the same hardware generation as the GPU used. All subsequent performance analyses have been performed with this synthesis. Handel-C by Mentor Graphics [34], a C-like hardware description language, has been used, except for critical parts such as interfaces and a memory controller that have been described with VHDL. A C-like language such as Handel-C allows one to reduce development time without significantly affecting hardware requirements or performance as compared to VHDL [46]. By using the remapping option to distribute combinational logic across flip-flop boundaries, the clock frequency can be significantly improved. To achieve a high performance with such high-level description languages, advanced design techniques such as superscalar and deeply pipelined circuit architectures are required [15], [39], [41]. These are supported thanks to Handel-C's built-in parallel statement. By combining these design methodologies with this language, a good trade-off can be found between time-to-market, design resources, and system performance. This is one of the challenges of current circuit design technologies.

6.2.1 System Overview

Fig. 3 contains the final architecture with its three main modules: the single scale optical flow core, the single scale

TABLE 1
Accuracy Loss for FPGA Simplifications (Simulation)

simplification	optical flow (*yosemite*)			
	AAE (°)	STD (°)	dens (%)	AE<5° (%)
original	2.10	3.06	82	84
image warping	3.50	3.91	69	58
three frames	5.16	7.89	96	68
fixed point	2.13	3.94	79	84
all simplifications	8.73	12.83	85	40

stereo core (which also computes the local features), and the main multiscale hierarchical processing units. This last block contains a reduction block for pyramid construction, an expansion block with bilinear interpolation, and a warping circuit (1D in the stereo case). The different processing units are synchronized by means of circular FIFO buffers or blocking channels, and external memory access is greatly reduced in the warping circuit by operating on images rather than filter responses. Each block of Fig. 3 represents a complex design. All system modules have been described in detail in [35], [41] and implemented separately in the XircaV4. A detailed description of the integration of all the circuits and an evaluation of the architecture's scalability is provided in [36].

6.2.2 Algorithm Modifications

As explained in previous sections, we have adopted the following hardware friendly simplifications: fixed-point arithmetic, reduced temporal support, and image warping. In the local features stage, the hardware cost has been reduced further by simplifying the computation of the S and C-components. The accuracy loss related to these simplifications has been studied with a MATLAB model of the hardware circuit. This preliminary study is very useful for the hardware designer as it allows adjusting the bit width. The accuracy loss associated with the final design of the most complex module, the optical flow, is reported in Table 1 for the well-known *yosemite* sequence [47]. The accuracy is evaluated in terms of the standard measures average angular error (AAE), deviation of the AAE (STD), and density of valid estimates. To facilitate comparison, we have also included an additional column that shows the percentage of flow vectors obtained with an angular error (AE) smaller than five degrees when the phase linearity threshold is set very high. All tests were performed without the median filter regularization. The last row contains the results when all simplifications are combined. These differ from those in Section 7.4 due to differences in the rounding model and the threshold operation adopted by the hardware. We can see from Table 1 that the largest accuracy loss results from the image warping simplification. Unfortunately, filter response warping requires 16 times more hardware resources. The reduction of the number of frames from five to three further reduces accuracy (and increases density) but is mandatory to reduce memory access. The fixed-point notation has only a negligible effect on accuracy, but it can reduce the density of the estimates.

We can conclude from this section that complex FPGA system implementations require many algorithmic modifications to map the algorithms onto the hardware logic. In particular if fixed-point arithmetic is employed, as in the system considered here, a systematic methodology is required to evaluate different modifications. This is much less in GPU system development since these devices are more oriented to directly execute algorithm descriptions.

TABLE 2
FPGA Module Complexity in Terms of Processing Units

module	# units	module	# units
pyramid	24	stereo/local features core	873
expansion	18	optical flow core	1,257
warping	102	median	39
merge	12	Gabor (1 bank)	318

6.2.3 Memory Controller Unit (MCU)

The final system output requires 63 bits storage for each pixel: 12 for disparity, 24 for optical flow, 9 for magnitude, 9 for local orientation, and 9 for phase. Memory management is very complicated since external memory addresses use $32 + 4$ bit words. In the optimal case, an external memory address can be accessed each clock cycle, but this performance can degrade significantly as a result of erroneous memory management scheduling. To manage the huge quantity of data accesses, it is therefore crucial to use a special Memory Controller Unit. We employ an MCU with different Abstract Access Ports (AAP) [48]. Due to its critical importance, this circuit is described in VHDL. The MCU operates at a higher frequency than the processing engine to enable temporal multiplexing of the different memory accesses. Blocking FIFOs link the different clock domains and a round robin priority system manages petitions coming from the AAPs. The MCU is also essential in the multiscale motion compensation (warping) and in synchronizing the different scales through memory operations.

7 Architecture Comparison

In this section, we compare the GPU and FPGA implementations in terms of arithmetic complexity, external and on-chip memory access, data dependency, accuracy, speed, power consumption, cost, and design time.

7.1 Arithmetic Complexity

The arithmetic complexity has been reduced in the FPGA through a variety of algorithmic simplifications. The GPU on the other hand requires fewer filtering operations. The FPGA outputs a pixel every 2.7 clock cycles and performs over 2,300 operations each cycle (estimation based on the approximate number of custom elementary processing units excluding pipeline synchronization). Table 2 shows a breakdown of this number according to the different vision modules. These modules are reused in the multiscale processing. Table 3 reports on the arithmetic complexity (and memory access) of the different GPU kernels when processing $1{,}280 \times 1{,}024$ images at six scales. The per pixel measures in this table refer to all pixels across all scales ($1{,}280 \times 1{,}024 \times \sum_{s=1}^{6} 2^{-2(s-1)} = 1{,}747{,}200$ in this example). The number of instructions executed and computation times were obtained with the CUDA Visual Profiler. The relative throughput is computed with respect

TABLE 3
GPU Kernel Arithmetic Complexity and External Memory Access ($1{,}280 \times 1{,}024$)

processing stage	time/frame (in ms)	Arithmetic Complexity # instructions (per pixel)	Arithmetic Complexity relative throughput	External Memory Access # inputs (per pixel)	External Memory Access # outputs (per pixel)	External Memory Access total (in MB)	External Memory Access throughput (in GB/s)
downsample	1.1	70	0.35	50	1/2	337	294
Gabor column	2.4	401	0.93	22	18	267	108
Gabor row	7.9	946	0.67	198	32	1,533	189
local features	1.3	244	1.07	16	3	127	96
stereo	2.6	507	1.08	29	1	200	74
optical flow	12.7	2,548	1.13	82	2	560	43
median filter	1.5	294	1.09	27	3	200	129

to the maximal single-issue instruction throughput (in certain occasions, the GPU can dual-issue instructions). From this column, we can infer that all kernels, except for downsampling and Gabor row filtering, are most likely compute bound.

When we compare Table 2 with the *#instructions* column in Table 3, we see that the (relative) arithmetic complexity of the optical flow module is much higher in the GPU than in the FPGA. Note however that in the FPGA, the stereo and optical flow cores also contain multiple Gabor filtering cores. If we also include the filtering stages in the GPU instruction counts, we obtain 2,098 instructions for *stereo+features* ($1{,}347 + 244 + 507$) and 3,221 for *optical flow* ($1{,}347/2 + 2{,}548$). Interestingly, although very different approaches were taken, the relative complexity is very similar in both architectures.

7.2 External and On-Chip Memory Access

Real-time computer vision systems depend to a large extent on high clock frequencies and high bandwidth external memory. FPGA technology severely limits these two aspects. Since GPUs do not suffer from the resultant logic delays, they can employ specific silicon logic that runs at high frequencies and can interface the latest memory technologies (see also Table 8). For these reasons, a specialized MCU has been developed that optimizes and schedules the external memory accesses in the FPGA (see Section 6.2.3). Fig. 4 summarizes the memory accesses to each of the FPGA's SRAM banks: banks B0 and B1 provide double buffer communication with the PC, bank B2 supports the optical flow computation, and bank B3 supports the stereo (and local features) computation. Fig. 4 also shows the number of accesses per bank, which differs for each processing path (stereo and optical flow). The cores communicate with memory through the MCU and cache previously used image rows in embedded BRAMs (on-chip memories) to optimize local accesses.

The external memory access requirements of the different GPU kernels are summarized in Table 3. In this context, external memory refers to the GPU's high bandwidth, large latency memory banks (as opposed to CPU memory). The throughput in the *downsample* and *Gabor row* kernels greatly exceeds the device bandwidth (measured at 113 GB/s on the GeForce GTX 280) due to the effective use of the texture cache. Considering also the small relative instruction throughput, it is clear that these two kernels are memory bound. The optical flow kernel on the other hand achieves the lowest memory and highest arithmetic throughput. The median filter kernel is most effective in exploiting all the GPU's resources by achieving both high memory and arithmetic throughput. Due to the sequential nature of the different processing stages, a pool of temporary storage can be reused often and the total storage requirements are equal to 193 and 772 MB for the 640×512 and $1{,}280 \times 1{,}024$ examples, respectively. This large requirement results from the $5 + 1$ frames Gabor pyramid buffer and could become problematic at higher image resolutions. With regard to internal (on-chip) memory, the GPU implementation extensively uses constant and texture (both automatically cached) rather than shared memory. Recent GPU optimization studies also encourage the use of register space over shared memory whenever possible [49]. Shared memory could be used to further enhance the algorithms. The communication between the CPU and the GPU is performed asynchronously concurrently with computation and thus only slightly increases latency without affecting throughput.

Note that one of the reasons for the high computing power of the GPU is the parallel access to a large number of memory chips clocked at a very high frequency. Depending on the package, FPGAs can also exploit a larger number of memory chips, but at a much lower clock frequency. Solutions such as the built-in memory controllers presented in Spartan-6 devices can help future FPGA devices to overcome this technology bottleneck.

7.3 Data Dependency

Although the algorithms presented in the previous sections are highly parallelizable, important data dependencies exist

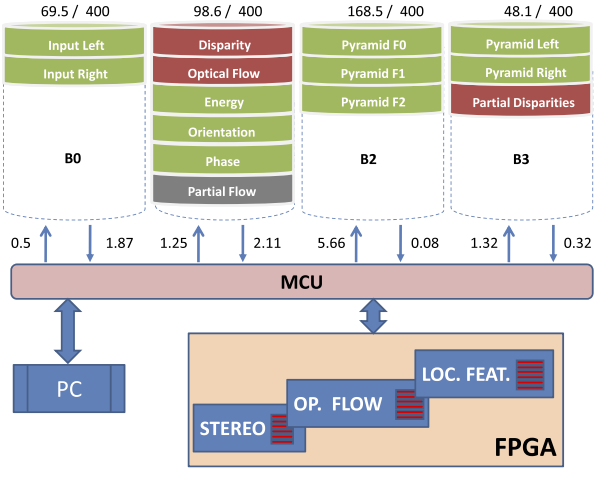

Fig. 4. Number of read and write memory accesses per pixel in the different FPGA banks. The bandwidth used versus available is shown above each bank in MB/s.

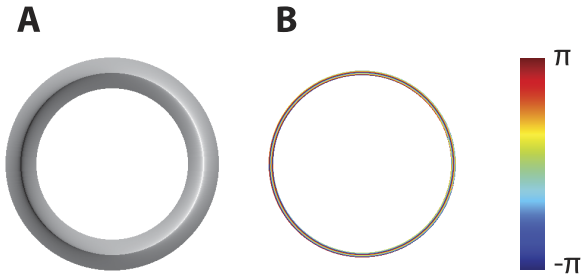

Fig. 5. (A) Test image with continuously changing phase in the center of the circular manifold. (B) Local phase estimated on this image (GPU).

that affect both architectures in a different way. The FPGA architecture is most affected by the multiscale nature of the algorithms. Since the image pyramid construction is inherently sequential, it must be completed before the main processing can start. The coarse-to-fine control scheme introduces additional dependencies that increase the number of pixels, operations (image warping), and external data storage requirements. These dependencies are less severe for the GPU architecture. The warping operations are efficiently handled by the texture units (and thus essentially free) and the external memory access is not a limiting factor since the feature extraction kernels are compute- rather than memory bound. The downsampling stage is memory bound but only represents a small proportion of the total processing time (see Table 3). Much more severe for the GPU architecture is the data dependency that results from the separable Gabor filtering, where external memory access cannot be avoided in between column and row filtering. The memory-bound row filtering kernel takes up a large proportion of the total processing time. The only alternatives here are either performing nonseparable convolution, which we found to be slower (results not shown), or retaining the column filtering results on chip, which requires more resources than available. The FPGA does not have this problem since the column filtering results can be stored in embedded BRAMs. The Gabor filtering core is even replicated multiple times in the disparity and optical flow modules.

All the spatial interactions occur through the Gabor and median filtering, and through the coarse-to-fine control scheme. A very large amount of parallelism can be exploited within each scale since each pixel is processed independently. It is here that the FPGA's increased flexibility enables a more fine-grained parallelism. The FPGA processes different pixels sequentially but performs all tasks in parallel by over 3,000 custom processing units. The GPU on the other hand processes a large number of pixels in parallel, but performs the different operations sequentially.

TABLE 4
Local Features Accuracy

	localization		orientation		phase	
	AVG (pixel)	STD (pixel)	AVG (°)	STD (°)	AVG (°)	STD (°)
FPGA	0.13	1.24	0.43	2.61	7.22	69.26
GPU	0.07	0.20	0.28	3.56	6.55	11.50
CPU	0.07	0.20	0.28	3.56	6.55	11.50

7.4 Accuracy

We adopt a similar methodology as described in [2] to evaluate the accuracy of both systems. The local feature estimation is evaluated using a synthetic image (Fig. 5A) in which the feature type in the center of a circular manifold changes from a step edge to a line. For stereo and optical flow, we use benchmark sequences from the Middlebury data set [47] for which ground truth is available.

The average and standard deviation of the errors in feature localization, orientation and phase (evaluated at 500 points), are summarized for both architectures and a reference CPU implementation in Table 4. The phase estimated on the GPU is shown in Fig. 5B. The results are very accurate and comparable for both the GPU and FPGA, but the GPU results are nearly identical to the CPU results. As a result of the low precision arctangent core, the phase estimation is significantly less precise in the FPGA.

Table 5 summarizes the stereo results for the FPGA, GPU, and CPU in terms of the mean absolute error (MAE), the standard deviation of the error (STD), the density of valid estimates, and the proportion of pixels with an absolute error smaller than 0.5 pixels. Once again, the GPU and CPU results cannot be discerned from this table. As expected, the FPGA results are less accurate (mainly due to image warping) and less dense (mainly due to the fixed-point arithmetic). The stereo estimates for the *cones* sequence are visualized in Fig. 6. For illustrative purposes, the rightmost image in Fig. 6 also shows that a simple left/right consistency check can remove most of the erroneous estimates on the GPU [2].

The optical flow is evaluated using the measures from Section 6.2.2 and the results are summarized in Table 6. The same phase linearity threshold ($\tau_l = 0.5$) was used for all sequences and architectures. The differences between the GPU and CPU are still small but, due to the increased arithmetic complexity, they are now visible in the table. The FPGA again obtains a lower accuracy as a result of the different algorithmic simplifications that had to be adopted. The optical flow estimates obtained on the *rubberwhale* sequence are visualized in Fig. 7. In addition to the medium

TABLE 5
Stereo Accuracy

sequence	FPGA				GPU				CPU			
	MAE (pixel)	STD (pixel)	dens (%)	AE<0.5 (%)	MAE (pixel)	STD (pixel)	dens (%)	AE<0.5 (%)	MAE (pixel)	STD (pixel)	dens (%)	AE<0.5 (%)
venus	0.74	1.38	87	52	0.41	0.99	98	84	0.41	0.99	98	84
tsukuba	0.84	1.44	92	58	0.67	1.33	100	76	0.67	1.33	100	76
cones	2.52	6.25	85	53	1.19	2.91	88	69	1.19	2.91	88	69
teddy	2.97	5.89	68	34	1.99	3.99	91	62	1.99	3.99	91	62

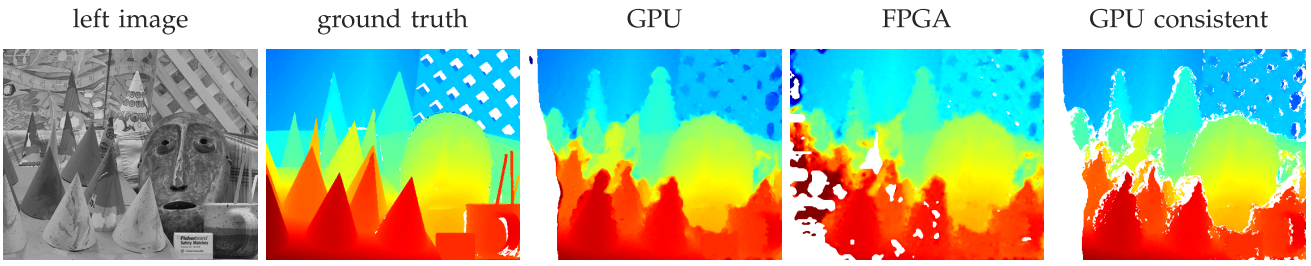

Fig. 6. Left image, ground truth, and estimated disparity for the *cones* image pair.

TABLE 6
Optical Flow Accuracy

sequence	FPGA AAE (°)	FPGA STD (°)	FPGA dens (%)	FPGA AE<5° (%)	GPU AAE (°)	GPU STD (°)	GPU dens (%)	GPU AE<5° (%)	CPU AAE (°)	CPU STD (°)	CPU dens (%)	CPU AE<5° (%)
yosemite	5.20	8.20	92	64	2.70 (1.77)	5.42 (2.25)	99 (83)	87	2.67 (1.78)	5.09 (2.23)	99 (84)	87
divtree	4.00	3.65	86	65	1.82 (1.69)	1.60 (1.33)	100 (97)	97	1.80 (1.69)	1.46 (1.31)	100 (97)	97
rubberwhale	11.20	20.60	79	46	6.00 (3.87)	14.61 (8.14)	99 (88)	79	6.03 (3.85)	14.71 (8.17)	99 (87)	79
grove3	12.08	19.34	93	47	6.06 (2.49)	15.23 (5.05)	88 (44)	67	5.86 (2.47)	14.44 (5.39)	88 (46)	67

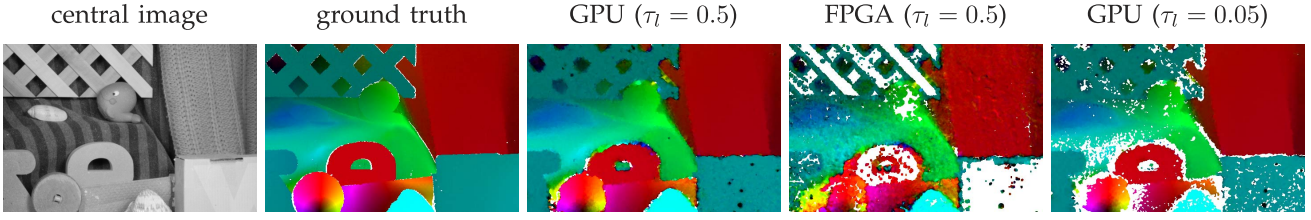

Fig. 7. Central image, ground truth, and estimated optical flow for the *rubberwhale* sequence.

accuracy result, we have also included results obtained at a higher accuracy setting ($\tau_l = 0.05$) in Table 6 (in brackets) and Fig. 7 (rightmost image). Due to the lower precision on the FPGA resulting from the fixed-point arithmetic, such high precision estimates can only be obtained on the GPU and CPU.

7.5 Speed

The total processing time, frame rate, throughput, and cycles/pixel obtained with the particular GPU and FPGA systems used here are summarized in Table 7 for medium and high image resolutions and for the single and multiscale cases. The GPU's throughput increases at the high image resolution due to increased occupancy of the cores. The FPGA's speed more than doubles in the single scale case since the critical image pyramid data dependency is avoided. As a result, the system can process 1 pixel per clock cycle. The FPGA's lower clock frequency (one order of magnitude less than the GPU) is the main cause of the lower frame rate and throughput. If we normalize performance by clock frequency, the FPGA requires about 20 and 10 times fewer cycles/pixel than the GPU in the single and multiscale cases, respectively.

7.6 Power Consumption, Cost, and Design Time

Table 8 summarizes the power consumption, cost [50], [51], time-to-market (estimated), and required human resources (postdoctoral level) for different GPU and FPGA generations. The FPGA design time also includes the analysis of the fixed-point representation and the evaluation of the algorithmic simplifications. Developing this complex low-level vision engine in a constrained amount of time required the use of high-level languages to (greatly) simplify the design process. Although a low-level system description (closely mapping FPGA circuitry and manual placement) can significantly increase performance, time-to-market deadlines often favor a fast development. For this reason, FPGA designers usually work at a higher level of abstraction, using for instance Verilog or VHDL languages at RT level or high-level description languages (such as Handel-C) as we did here. In the GPU, a working system can be obtained in a matter of days if the model is understood and C- or MATLAB-code is available. Optimization to the levels of performance reported here however requires a considerable amount of additional time, expertise, and a great deal of experimental tuning. Table 8 also summarizes the computational power of the platforms in terms of absolute and normalized (by power consumption) number of operations per second (OPS). The latter is

TABLE 7
Speed

	640×512 GPU	640×512 FPGA	1280×1024 GPU	1280×1024 FPGA
single scale				
proc. time (msec/frame)	5.4	6.1	20.5	24.7
frame rate (Hz)	185	161	49	40
throughput (MP/s)	61	53	64	53
cycles/pixel	21.4	1	20.3	1
multiscale				
proc. time (msec/frame)	9.0	16.6	30.3	66.6
frame rate (Hz)	111	60	33	15
throughput (MP/s)	36	20	43	20
cycles/pixel	35.6	2.7	30.0	2.7

TABLE 8
Platform Characteristics

platform	fab (in nm)	power (in W)	cost (in USD)	time-to-market (in months)	proc. clock (in MHz)	mem. clock (in MHz)	abs. perf. (in G(FL)OPS)	norm. perf. (in G(FL)OPS/W)
GeForce GTX 280	65	236.0	-	2 (1 person)	1,296	2,214 (GDDR3)	933	4.0
GeForce GTX 580	40	244.0	499	2 (1 person)	1,544	2,004 (GDDR5)	1,581	6.5
Virtex4 xc4vfx100	90	7.2	2,084	15 (2 persons)	42	100 (SRAM)	92	12.8
Virtex5 xc5vlx330t	65	5.5	12,651	12 (2 persons)	53	100 (SRAM)	165	30.0

expressed in fixed-point operations per second for the FPGA and floating point operations per second (FLOPS) for the GPU. The GPU provides an order of magnitude more computational resources, but is outperformed by the FPGA when power consumption is taken into account.

8 Discussion

A comparison between FPGA and GPU architectures is complicated by continuous changes in technology and marketing strategies of the two platforms, which both substantially affect the computational resources available. The introduction of embedded processing cores or new memory interfaces can significantly affect an architecture. Speed and power consumption can be improved in both architectures simply by adopting the latest technology. The synthesis tool facilitates this adaptation in the FPGA although some interfaces to external devices may have to be redesigned. The FPGA's performance can be increased by utilizing DDR-based memory, which can be clocked much higher than SSRAM. Unfortunately, most prototyping platforms contain only one such memory bank, which is insufficient for the current system. We expect that a custom-designed board with four DDR memory banks would allow for at least a 50 percent performance increase, but the evaluation of this issue is beyond the scope of the present contribution. In addition to an increased number of processing cores and available memory, the GPU typically acquires new capabilities in each generation. Such changes may require modifications in the configuration of the kernel calls or even the kernel code itself to achieve optimal performance. The C-like programming interface greatly facilitates such tuning. A transfer to devices with fewer resources is straightforward on the GPU, but requires significant architectural changes on the FPGA. Two main strategies can be adopted here: hardware sharing and accuracy reduction [36]. The first sacrifices throughput to retain precision and vice versa for the second. Both strategies were required in the Virtex 4 implementation (Table 8) in order to fit the complete architecture in such a low cost device (hence the increased time-to-market).

We have summarized the major results of our comparison in Table 9, by assigning the components of the vision engine and the comparison to the most suitable target platform. The Gabor filterbank is more suited to the FPGA since the GPU requires intermediate storage for the separable filtering, which results in a memory-bounded kernel. On the other hand, the central use of phase as image descriptor is more interesting for the GPU due to the availability of the special function units and the floating point nature of the device. The FPGA suffers here from the fixed-point representation which affects thresholds. The phase warping also exceeds the external memory available to the FPGA, and needs to be replaced by image warping. The local features computation requires additional time on the GPU whereas it is merged with the disparity computation on the FPGA. The image pyramid introduces a critical data dependency in the FPGA implementation that causes a significant overhead. The median filtering is very efficient on the GPU, but still represents more overhead than in the FPGA case. The warping operations are much better suited for the GPU due to the availability of the texture units, which facilitate irregular memory access and interpolation. The high arithmetic intensity of the optical flow module can be better handled through the operation level parallelism available on the FPGA. On the other hand, large temporal support (five frames) requires a large amount of memory, which is only available on the GPU. The FPGA implementation is many times more power efficient than the GPU, although some accuracy has to be compromised. The FPGA requires an order of magnitude fewer cycles per pixel, but only achieves around half the absolute speed of the GPU (in the multiscale case). A GPU design strongly depends on the architecture provided by the manufacturer, whereas an FPGA design leaves more choices to the engineer. This flexibility of the FPGA (optical flow operation level parallelism, multiple convolution cores, etc.) comes at the cost of a much larger design time than the GPU.

The results presented here only consider a subset of all possible problem types. For example, it is well understood that the FPGA's representation flexibility makes it better suited for problems involving a large number of bitwise operations (e.g., biological sequence alignment). Many problems, on the other hand, require a much larger dynamic range than the application considered in this work, and then a fixed-point representation is no longer suitable. It is also well known that the GPU excels on computationally intense, high throughput applications with large data sets, but is not suitable for applications that require very short latency responses.

TABLE 9
Most Suitable Target Platform

FPGA	GPU
Gabor filterbank	phase as image feature
local features	image pyramid
median filtering	warping
arithmetically intense optical flow	multi-frame optical flow
power consumption	accuracy
normalized speed	absolute speed
flexibility	design time
embedded platforms	high-performance computing
certification capabilities	low cost

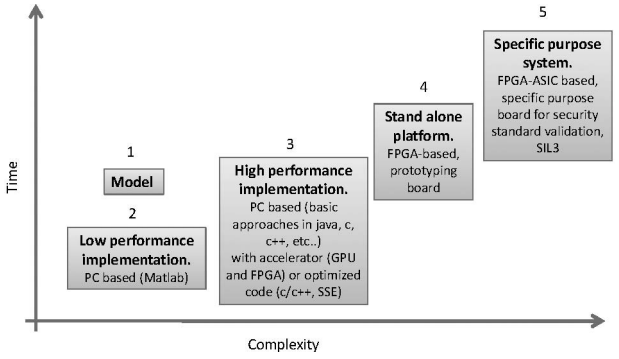

Fig. 8. System development stages and target applications as a function of time and complexity.

8.1 Target Scenarios and Products

The smaller clock frequency and power requirements render the FPGA platform very suitable for embedded applications, especially in the latest devices such as Xilinx' Virtex 6 and 7 families [40]. Traditionally, GPUs have not been considered suitable for this, but preliminary systems are starting to appear (e.g., GE's Intelligent Platforms [52]). Another important strength of the FPGA (e.g., the Virtex 5) is its compliance with certification standards such as DO-254 [53] which enable its application in critical aerospace and defense applications. The GPU on the other hand benefits from the large market penetration (gaming) that greatly reduces its cost for general purpose computation. The FPGA's high cost can only be reduced for very-high-volume applications by means of an ASIC implementation of the FPGA prototype.

The development of a complex vision system requires different stages depending on the target application. In this work, we have focused on GPU and FPGA implementations. In Fig. 8, we have situated these and other platforms in a time-complexity plot. For real-time purposes, we can either choose a stand-alone platform, based mainly on FPGA technology, or a standard PC accelerated through optimized code or a coprocessing board such as an FPGA or GPU. If we want to move toward an industrial product, we need to complete all the different stages and use technologies such as DSPs and FPGA-ASICs. We have not addressed this issue in this work. While FPGAs can cover a wide spread of applications including industrial, robotics, automotive, aerospace, and military, GPUs are still limited in this respect by the restricted certification capabilities and the high power consumption (but see [52]). For this reason, FPGAs are present in three different stages of Fig. 8 whereas GPUs only occur in one.

Important efforts are being invested in applying GPUs in high-performance computing (as opposed to embedded systems). Of particular relevance here is NVIDIA's Tesla line of professional products, which are more durable and have special features, such as ECC data protection.

9 Conclusion

Platform comparisons such as the one performed in this work strongly depend on the specific hardware used, but general conclusions can still be drawn. The work in [33] examined five relatively simple image processing algorithms implemented on a Xilinx Virtex 4 FPGA and a GeForce GTX 7900 GPU. In accordance with previous comparative work, the authors found that in the presence of data dependencies, a customized FPGA data path exceeds GPU performance. For such simple models, FPGA technology seems to be more suitable. We have extended the state of the art by comparing both platforms using medium to highly complex vision algorithms that stretch the FPGA to its limits. The complex vision modules and their integration considered here, provide a general idea of the two platforms' behavior when both intensive (and locally random) external memory access and high arithmetic complexity are involved. Our contribution demonstrates that especially the high bandwidth provided through newer GPU memory interfaces enables the GPU to overcome the FPGA in terms of absolute performance, when complex models are implemented. The GPU's exceptional performance here is also to a large extent due to the iterative nature of the multiscale algorithms. It was stated in [33] that the FPGA's embedded memory bandwidth enables it to overcome the GPU for local and deterministic processing, but as we have shown here, external and random (but with 2D locality) memory accesses are more suitable for the newer GPU architectures and their texture units. On the other hand, the cycles per pixel requirements show that a customized architecture is much more efficient than a generic one. An ASIC conversion of the FPGA is expected to outperform the GPU since it can reduce the GPU's clock frequency advantage and it requires fewer simplifications due to the increased number of resources. Both for simple and complex models, the development time is higher when addressing FPGA technology (and even more so for an ASIC conversion) due to the large amount of choices given to the designer, as opposed to GPU approaches in which a base computing architecture is already defined and taking full advantage of it is the goal. Although GPUs are not suitable for many embedded applications and none of the critical ones, as a coprocessing board, the GPU surpasses the FPGA in most of our comparisons.

Acknowledgments

This work was supported by research grants received from the program Financing program (PFV/10/008) and the CREA Financing program (CREA/07/027) of the K.U.Leuven, the Belgian Fund for Scientific Research—Flanders (G.0588.09), the Interuniversity Attraction Poles Programme—Belgian Science Policy (IUAP P6/29), the Flemish Regional Ministry of Education (Belgium) (GOA 10/019), the SWIFT prize of the King Baudouin Foundation of Belgium, the European Commission (IST-2007-217077), and ARC-VISION (TEC2010-15396) and CEI BioTIC GENIL (CEB09-0010) of the MICINN CEI program.

References

[1] D. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *Int'l J. Computer Vision*, vol. 60, no. 2, pp. 91-110, 2004.

[2] S. Sabatini, G. Gastaldi, F. Solari, K. Pauwels, M. Van Hulle, J. Díaz, E. Ros, N. Pugeault, and N. Krüger, "A Compact Harmonic Code for Early Vision Based on Anisotropic Frequency Channels," *Computer Vision and Image Understanding*, vol. 114, no. 6, pp. 681-699, 2010.

[3] K. Pauwels, N. Krüger, M. Lappe, F. Wörgötter, and M.M. Van Hulle, "A Cortical Architecture on Parallel Hardware for Motion Processing in Real Time," *J. Vision,* vol. 10, no. 10, p. article 18, 2010. http://www.journalofvision.org/content/10/10/18.abstract.

[4] R. Nelson and J. Aloimonos, "Obstacle Avoidance Using Flow Field Divergence," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 11, no. 10, pp. 1102-1106, Oct. 1989.

[5] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, "Robust Object Recognition with Cortex-Like Mechanisms," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 29, no. 3, pp. 411-426, Mar. 2007.

[6] T. Corpetti, E. Memin, and P. Perez, "Dense Estimation of Fluid Flows," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 24, no. 3, pp. 365-380, Mar. 2002.

[7] G. Papadopoulos, A. Briassouli, V. Mezaris, I. Kompatsiaris, and M. Strintzis, "Statistical Motion Information Extraction and Representation for Semantic Video Analysis," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 19, no. 10, pp. 1513-1528, Oct. 2009.

[8] P. Kovesi, "Phase Preserving Denoising of Images," *Proc. Fifth Int'l/Nat'l Biennial Conf. Digital Image Computing, Techniques, and Applications (DICTA '99),* pp. 212-217, 1999.

[9] K. Pauwels and M. Van Hulle, "Optic Flow from Unstable Sequences through Local Velocity Constancy Maximization," *Image and Vision Computing,* vol. 27, no. 5, pp. 579-587, 2009.

[10] A. Bruhn, J. Weickert, and C. Schnorr, "Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow Methods," *Int'l J. Computer Vision,* vol. 61, no. 3, pp. 211-231, 2005.

[11] D. Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," *Int'l J. Computer Vision,* vol. 47, nos. 1-3, pp. 7-42, 2002.

[12] M. Anguita, J. Díaz, E. Ros, and F. Fernandez-Baldomero, "Optimization Strategies for High-Performance Computing of Optical-Flow in General-Purpose Processors," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 19, no. 10, pp. 1475-1488, Oct. 2009.

[13] G. Bradski, "The OpenCV Library," *Dr. Dobb's J. Software Tools,* vol. 25, pp. 120-126, 2000.

[14] J. Díaz, E. Ros, F. Pelayo, E. Ortigosa, and S. Mota, "FPGA-Based Real-Time Optical-Flow System," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 16, no. 2, pp. 274-279, Feb. 2006.

[15] J. Díaz, E. Ros, R. Carrillo, and A. Prieto, "Real-Time System for High-Image Resolution Disparity Estimation," *IEEE Trans. Image Processing,* vol. 16, no. 1, pp. 280-285, Jan. 2007.

[16] J. Lu, S. Rogmans, G. Lafruit, and F. Catthoor, "Stream-Centric Stereo Matching and View Synthesis: A High-Speed Approach on GPUs," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 19, no. 11, pp. 1598-1611, Nov. 2009.

[17] J. Marzat, Y. Dumortier, and A. Ducrot, "Real-Time Dense and Accurate Parallel Optical Flow Using CUDA," *Proc. Int'l Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG),* Feb. 2009.

[18] A. Bruhn, J. Weickert, T. Kohlberger, and C. Schnoerr, "A Multigrid Platform for Real-Time Motion Computation with Discontinuity-Preserving Variational Methods," *Int'l J. Computer Vision,* vol. 70, no. 3, pp. 257-277, 2006.

[19] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers, and H. Bischof, "Anisotropic Huber-L1 Optical Flow," *Proc. British Machine Vision Conf. (BMVC),* Sept. 2009.

[20] W.J. MacLean, S. Sabihuddin, and J. Islam, "Leveraging Cost Matrix Structure for Hardware Implementation of Stereo Disparity Computation Using Dynamic Programming," *Computer Vision and Image Understanding,* vol. 114, no. 11, pp. 1126-1138, 2010.

[21] J. Bergen, P. Anandan, K. Hanna, and R. Hingorani, "Hierarchical Model-Based Motion Estimation," *Proc. European Conf. Computer Vision (ECCV),* pp. 237-252, 1992.

[22] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli, "State-of-the-Art in Heterogeneous Computing," *Scientific Programming,* vol. 18, pp. 1-33, 2010.

[23] S. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," *Proc. Symp. Application Specific Processors,* pp. 101-107, June 2008.

[24] S. Sarkar, G. Kulkarni, P. Pande, and A. Kalyanaraman, "Network-on-Chip Hardware Accelerators for Biological Sequence Alignment," *IEEE Trans. Computers,* vol. 59, no. 1, pp. 29-41, Jan. 2010.

[25] N. Gac, S. Mancini, M. Desvignes, and D. Houzet, "High Speed 3D Tomography on CPU, GPU, and FPGA," *EURASIP J. Embedded Systems,* vol. 2008, pp. 1-12, 2008.

[26] D.B. Thomas, L. Howes, and W. Luk, "A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation," *FPGA '09: Proc. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays,* pp. 63-72, 2009.

[27] L. Howes, P. Price, O. Mencer, O. Beckmann, and O. Pell, "Comparing FPGAs to Graphics Accelerators and the Playstation 2 Using a Unified Source Description," *Proc. Int'l Conf. Field Programmable Logic and Applications (FPL '06),* pp. 1-6, Aug. 2006.

[28] R. Weber, A. Gothandaraman, R.J. Hinde, and G.D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 1, pp. 58-68, Jan. 2011.

[29] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," *Proc. IEEE Seventh Symp. Application Specific Processors (SASP '09),* pp. 35-42, July 2009.

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, Mar. 2008.

[31] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance Comparison of FPGA, GPU and CPU in Image Processing," *Proc. Int'l Conf. Field Programmable Logic and Applications (FPL),* pp. 126-131, 2009.

[32] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, "Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study," *Proc. 16th Int'l Symp. Field-Programmable Custom Computing Machines,* pp. 173-182, Apr. 2008.

[33] B. Cope, P. Cheung, W. Luk, and L. Howes, "Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study," *IEEE Trans. Computers,* vol. 59, no. 4, pp. 433-448, Apr. 2010.

[34] "Handel-C Mentor Graphics," Website, http://www.mentor.com/products/fpga/handel-c/, 2010.

[35] M. Tomasi, M. Vanegas, F. Barranco, J. Díaz, and E. Ros, "High-Performance Optical-Flow Architecture Based on a Multiscale, Multi-Orientation Phase-Based Model," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 20, no. 12, pp. 1797-1807, Dec. 2010.

[36] M. Tomasi, "Pyramidal Architecture for Stereo Vision and Motion Estimation in Real-Time FPGA-Based Devices," PhD dissertation, Univ. of Granada, June 2010.

[37] E. Ros, J. Díaz, S.M.I. Odeh, and A. Cañas, "A Low Level Real-Time Vision System Using Specific Computing Architectures," *Proc. Sixth WSEAS Int'l Conf. Signal Processing, Computational Geometry and Artificial Vision,* pp. 192-197, 2006.

[38] P. Burt and E. Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Trans. Comm.,* vol. 31, no. 4, pp. 532-540, Apr. 1983.

[39] J. Díaz, E. Ros, S. Mota, and R. Carrillo, "Local Image Phase, Energy and Orientation Extraction Using FPGAs," *Int'l J. Electronics,* vol. 95, no. 7, pp. 743-760, 2008.

[40] "Xilinx," Website, http://www.xilinx.com, 2010.

[41] M. Tomasi, M. Vanegas, F. Barranco, J. Díaz, and E. Ros, "Real-Time Architecture for a Robust Multiscale Stereo Engine," *IEEE Trans. Very Large Scale Integration Systems,* vol. PP, no. 99, pp. 1-12 2010.

[42] D.J. Fleet and A.D. Jepson, "Computation of Component Image Velocity from Local Phase Information," *Int'l J. Computer Vision,* vol. 5, pp. 77-104, 1990.

[43] T. Gautama and M. Van Hulle, "A Phase-Based Approach to the Estimation of the Optical Flow Field Using Spatial Filtering," *IEEE Trans. Neural Networks,* vol. 13, no. 5, pp. 1127-1136, Sept. 2002.

[44] *NVIDIA CUDA Programming Guide,* NVIDIA, 2004.

[45] "Seven Solutions," Website, http://www.sevensols.com, 2010.

[46] E.M. Ortigosa, A. Canas, E. Ros, P.M. Ortigosa, S. Mota, and J. Diaz, "Hardware Description of Multi-Layer Perceptrons with Different Abstraction Levels," *Microprocessors and Microsystems,* vol. 30, no. 7, pp. 435-444, 2006.

[47] "Middlebury Computer Vision Pages," Website, http://vision.middlebury.edu, 2010.

[48] M. Vanegas, M. Tomasi, J. Díaz, and E. Ros, "Multi-Port Abstraction Layer for FPGA Intensive Memory Exploitation Applications," *J. Systems Architecture,* vol. 56, no. 9, pp. 442-451, 2010.

[49] V. Volkov, "Better Performance at Lower Occupancy," *Proc. GPU Technology Conf. (GTC)*, 2010.
[50] "Digikey," Website, http://www.digikey.com/, May 2010.
[51] "NVIDIA," Website, http://www.nvidia.com/, May 2010.
[52] "Many-Core Processors Report Ready for Duty," white paper, General Electric, http://www.ge-ip.com/gpgpu, 2010.
[53] "Xilinx Aerospace and Defense," Website, http://www.xilinx.com/esp/aerospace-defense/index.htm, 2011.

Karl Pauwels received the MSc degree in commercial engineering, the MSc degree in artificial intelligence, and the PhD degree in medical sciences from the Katholieke Universiteit Leuven, Belgium. He is currently a postdoc at the Laboratorium voor Neuro- en Psychofysiologie, Medical School, K.U.Leuven. His research interests include optical flow, stereo and camera motion estimation in the context of real-time computer vision.

Matteo Tomasi received the bachelor's degree in electronic engineering in 2006 from the University of Cagliari, Italy. He received the MSc degree in computer engineering in 2007 and the PhD degree in 2010 from the University of Granada. Since 2007, he has been involved in the University of Granada at the Department of Computer Architecture and Technology. He had involved in the European Projects DRIVSCO and Sensopac. His main research interests are HDL and hardware design, real-time systems, reconfigurable systems, and computer vision and image processing.

Javier Díaz received the MS degree in electronics engineering in 2002 and the PhD degree in electronics in 2006 both from the University of Granada. Currently, he is an assistant professor at the Department of Computer Architecture and Technology of the same university. His main research interests are cognitive vision systems, high-performance image processing architectures, and embedded systems based on reconfigurable devices. He is also interested in safety-critical systems, biomedical/bioinspired devices, and robotics.

Eduardo Ros received the PhD degree in 1997 from the University of Granada. He is currently a full professor at the Department of Computer Architecture and Technology of the same University. He is currently the responsible researcher at the University of Granada of two European projects related to bioinspired processing schemes and real-time image processing. His research interests include hardware implementation of digital circuits for real-time processing in embedded systems and high-performance computer vision.

Marc M. Van Hulle received the MSc degree in electrotechnical engineering and the PhD degree in applied sciences from the Katholieke Universiteit Leuven (K.U.Leuven), Belgium. He also received the BSc Econ and MBA degrees. He received the Doctor Technices degree from Queen Margrethe II of Denmark, in 2003, and an honorary doctoral degree from Brest State University, Belarus, in 2009. He is currently a full professor at the K.U.Leuven Medical School, where he heads the Computational Neuroscience Group of the Laboratorium voor Neuro- en Psychofysiologie. In 1992, he was with the Brain and Cognitive Sciences Department, Massachusetts Institute of Technology, Boston, as a postdoctoral scientist. He has authored a monograph titled *Faithful Representations and Topographic Maps: From Distortion- to Information-Based Self-Organization* (John Wiley, 2000; also translated into Japanese) and 200 technical publications. His current research interests include computational neuroscience, neural networks, computer vision, data mining, and signal processing. He is a senior member of the IEEE.