Implementation of an Affine-Covariant Feature Detector in Field-Programmable Gate Arrays

Cristina Cabani & W. James MacLean

Department of Electrical and Computer Engineering, University of Toronto
{cabani|maclean}@eecg.utoronto.ca

Abstract. This article describes an FPGA-based implementation of the Harris-Affine feature detector introduced by Mikolajczyk and Schmid [1, 2]. The system is implemented on the Transmogrifier-4, a prototyping platform that includes four Altera Stratix S80 FPGAs and NTSC/VGA video interfaces. The system achieves a speed of 90–9000 times the speed of an equivalent software implementation, allowing it to process standard video (640×480 pixels) at 30 frames per second.

1 Introduction & Previous Work

Feature detectors are algorithms that can locate and describe points or regions of interest in an image. The idea is that if a structure in an image can be described by a limited set of features, and this set is robust to changes in viewing conditions, then the features can be used to identify and match the structure across different images. Most modern feature detectors can trace their roots back to the Harris corner detector [3], which selects image regions that have sufficient image-gradient energy in orthogonal directions. While these features are not scale invariant, subsequent advances such as SIFT features [4, 5], scale-space representations [6] and affine-covariant approaches [2] search for maximal responses over different scales to provide scale invariance. Rotational invariance is usually achieved with a *signature* (or *descriptor*) that identifies the features and normalises it with respect to a dominant orientation.

As the complexity of the algorithm increases, so does the amount of time and computer resources required to perform it. For this reason, feature detection algorithms are being ported to integrated circuits such as Field-Programmable Gate Arrays (FPGAs), where the inherent parallelism of the image processing operations can be exploited to increase the computational speed. Among hardware circuits, FPGAs are attractive because they can be reprogrammed to implement multiple applications. This allows designers to test the system under real conditions and implement modifications easily, considerably shortening the development time of the hardware system.

A number of hardware approaches have been reported. There are several FPGA-based Harris corner detectors [7, 8], and an FPGA-based implementation of SIFT features is indicated in [9], although the implementation details are not public. These detection algorithms are amenable to hardware implementation due to the lack of an iterative process. Our design addresses the implementation of affine-covariant detectors and the issue of iteration.

2 The Harris-Affine Feature Detector

The Harris-Affine detector combines a multiscale version of the Harris corner detector with an iterative procedure for computing the affine shape of an image region in the neighbourhood of the features. This combination results in features that adapt to significant affine transformations caused by changes in viewpoint. The multiscale Harris detector computes the scale-normalized second moment matrix $\mu(\mathbf{x}, \sigma_I, \sigma_D)$ at each pixel in the image to describe the variance of the image intensities around the pixel at location $\mathbf{x}$:

$$\mu(\mathbf{x}, \sigma_I, \sigma_D) = \sigma_D^2 \; g(\sigma_I) * \begin{bmatrix} L_x^2(\mathbf{x}, \sigma_D) & L_x L_y(\mathbf{x}, \sigma_D) \\ L_x L_y(\mathbf{x}, \sigma_D) & L_y^2(\mathbf{x}, \sigma_D) \end{bmatrix} \tag{1}$$

where σ_D and σ_I are the differentiation and integration scales, and $L_x(\mathbf{x}, \sigma_D)$ and $L_y(\mathbf{x}, \sigma_D)$ are the first derivatives of the image at $\mathbf{x}$ in the x- and y-directions computed with a Gaussian derivative at scale σ_D.

The algorithm finds points at which the second moment matrix has two large eigenvalues. The explicit computation of the eigenvalues is avoided by using the trace and determinant of $\mu(\mathbf{x}, \sigma_I, \sigma_D)$ [3] to evaluate the *cornerness* function,

$$\textit{cornerness} = \det(\mu(\mathbf{x}, \sigma_I, \sigma_D)) - \alpha \, \mathrm{trace}^2(\mu(\mathbf{x}, \sigma_I, \sigma_D)), \tag{2}$$

where α is a positive parameter usually in the range $0 \leq \alpha \leq 0.25$.

At each scale (σ_I, σ_D), a point is selected as a corner if its *cornerness* is a maximum in its 8-point neighbourhood. The algorithm then selects the *characteristic scale* of the feature as the scale that achieves a maximum value of the Laplacian function

$$|\mathrm{LoG}(\mathbf{x}, \sigma_I)| = \sigma_I^2 \, |L_{xx}(\mathbf{x}, \sigma_I) + L_{yy}(\mathbf{x}, \sigma_I)|. \tag{3}$$

The preliminary features obtained with this Harris-Laplace detector [10] are then refined to account for affine transformations in the image. In this iterative process, image regions around the features are normalized (warped) until the second moment matrix computed at the feature location has equal eigenvalues. The scale and location of each feature is recomputed at each iteration based on the normalized region instead of the original image.

2.1 Algorithm

The Harris-Affine algorithm is initialized with points extracted with the multiscale Harris detector. For each preliminary feature at location $\mathbf{x}^{(0)}$ and scale ${\sigma_I}^{(0)}$ the algorithm applies the following procedure:

1. Initialize the *shape adaptation matrix* U to the identity matrix: $U^{(0)} = I$.
2. Normalize an image region $W(\mathbf{x}_w)$ centred at $\mathbf{x}_w^{(k-1)} = {U^{(k-1)}}^{-1} \mathbf{x}^{(k-1)}$, where the superscript (k) denotes the kth iteration.
3. Select the *integration scale* ${\sigma_I}^{(k)}$ at the point $\mathbf{x}^{(k-1)}$ that maximizes the absolute value of the Laplacian (Equation 3).

4. Select the *differentiation scale* ${\sigma_D}^{(k)} = s\,{\sigma_I}^{(k)}$ that maximizes the isotropy ratio $Q = \frac{\lambda_{min}(\mu)}{\lambda_{max}(\mu)}$, where $s \in [0.5, \ldots, 0.75]$ and $\mu = \mu(\mathbf{x}_w^{(k-1)}, {\sigma_I}^{(k)}, {\sigma_D}^{(k)})$ is the second moment matrix at the point $\mathbf{x}_w^{(k-1)}$ computed from the image region W.
5. Detect the *spatial localization* $\mathbf{x}_w^{(k)}$ of a maximum of the Harris *cornerness* function (Equation 2) around $\mathbf{x}_w^{(k-1)}$ and compute the location of the feature in the original image domain $\mathbf{x}^{(k)}$:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + U^{(k-1)}\,(\mathbf{x}_w^{(k)} - \mathbf{x}_w^{(k-1)}) \tag{4}$$

6. Compute the inverse square root of the second moment matrix at $\mathbf{x}_w^{(k)}$: $\mu_i^{(k)} = \mu^{-\frac{1}{2}}(\mathbf{x}_w^{(k)}, {\sigma_I}^{(k)}, {\sigma_D}^{(k)})$.
7. Update the shape adaptation matrix to $U^{(k)} = \mu_i^{(k)} \cdot U^{(k-1)}$ and normalize $U^{(k)}$ so that $\lambda_{max}(U^{(k)}) = 1$.
8. Evaluate convergence and divergence ratios,

$$\text{convergence ratio} = 1 - \frac{\lambda_{min}(\mu)}{\lambda_{max}(\mu)} < \epsilon_C \tag{5}$$

$$\text{divergence ratio} = \frac{\lambda_{max}(U)}{\lambda_{min}(U)} < \epsilon_D. \tag{6}$$

The process converges when the matrix μ is sufficiently close to a rotation matrix, or equivalently, when its eigenvalues $\lambda_{max}(\mu)$ and $\lambda_{min}(\mu)$ are almost equal ($\epsilon_C \approx 0.05$). Divergence can be measured from the eigenvalues of U to discard features with deformations that are too eccentric (e.g., $\epsilon_D \approx 6$).

3 Hardware Design

The feature detector was implemented on the Transmogrifier-4 (TM-4), a development platform developed at the University of Toronto that contains four Stratix S80 FPGAs [11]. Figure 1 shows the high-level architecture of the detector.

At the front end, the Video Input module receives NTSC signals from a CCD camera, performs frame de-interlacing and color conversion, and forwards pixels corresponding to a 640×480 grayscale image to the processing stages of the detector. The core of the detector consists of the same two main stages present in the algorithm. The first stage is a Multiscale Harris corner detector that provides candidate feature points at three scales specified by the user. The second stage refines the location, scale and shape matrix of the candidate features, and is equivalent to the iterative portion of the algorithm, with the exception that the loops have been unrolled into individual identical modules (One Iteration) to provide greater throughput.

First-in-first-out (FIFO) buffers store information about candidate feature points after the Multiscale Harris module and after each iteration. Because preliminary feature points are discarded along the process, the FIFO buffers store

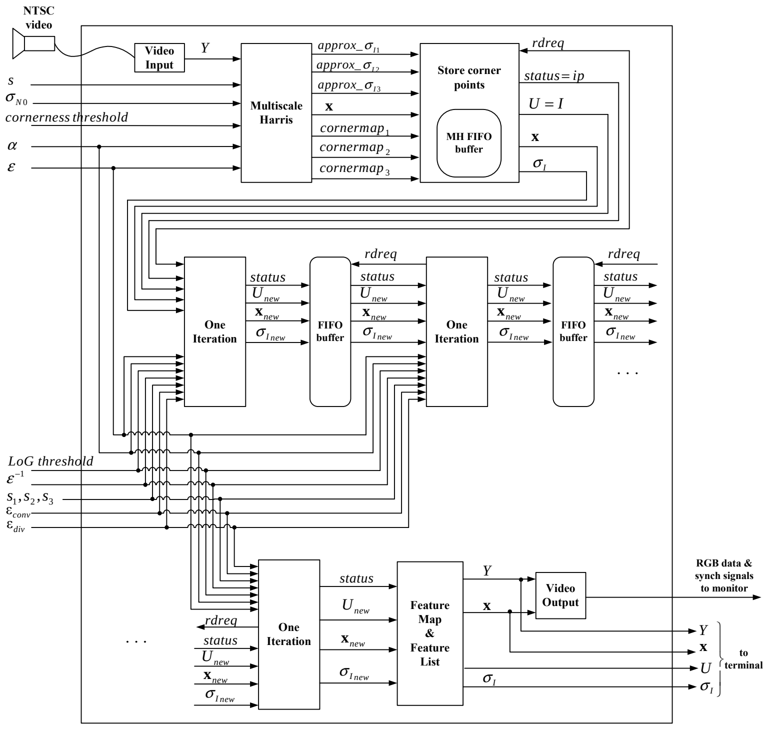


Fig. 1. High-level architecture of the feature detector

only the information that corresponds to features that are either still being analyzed or have been accepted as final feature points. This shortens the processing latency of a frame by allowing the One Iteration modules to operate only on non-discarded feature points. In addition, the MH FIFO buffer separates the clock domain of the Multiscale Harris stage which runs at 12Mhz from the clock domain of the pipeline of iterations, which runs at 60Mhz.

The location, scale and shape transformation matrix produced by the last iteration block are buffered in a Feature List which can be accessed from a Unix terminal. The locations are also marked in a binary Feature Map, so that the features can be overlaid on the video stream and displayed on a monitor via the Video Output module.

3.1 One Iteration Module

The One Iteration module (Figure 2) is the most complex of the two processing stages. The Normalize Window module performs the affine normalisation process. A 25×25 region around the feature at location $\mathbf{x}$ is warped by an affine transformation U. Since in general the transformation gives non-integer coordinates, the grayscale values for the warped image are computed by bilinear interpolation of the grayscale values in the original image. This interpolation creates a bottleneck in the system since four pixels have to be retrieved from a memory

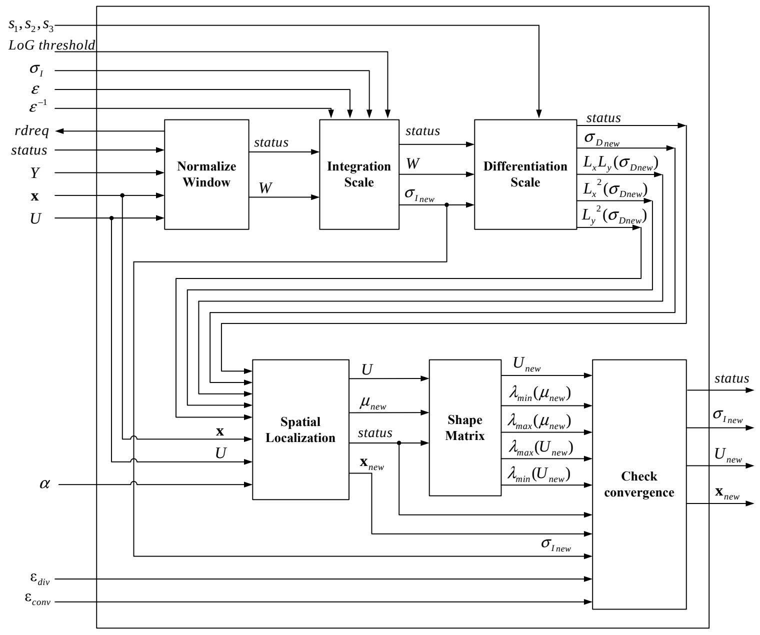


Fig. 2. One Iteration module

buffer for each of the 625 pixels that form the warped image. For this purpose, the original image is de-interlaced along the rows and columns and stored into four different buffers, so that the four pixels required for the interpolation can be accessed in a single clock cycle.

The resulting warped image W is forwarded to the next two modules which compute the integration and differentiation scales. The Integration Scale module evaluates the Laplacian function (Equation 3) at three scales: the current integration scale and the two adjacent scales, and selects the one that results in the maximum value of the Laplacian.

The Differentiation Scale module computes the eigenvalues of the second moment matrix μ (Equation 1) at three scales. The selected differentiation scale is the one that produces an isotropy ratio $Q = \frac{\lambda_{min}(\mu)}{\lambda_{max}(\mu)}$ that is closest to 1.

The Spatial Localization module computes the value of the cornerness function (Equation 2) at each location in the 8-point neighbourhood of the current feature and updates the location $\mathbf{x}$ to the location of the point that achieves the largest cornerness value.

The Shape Matrix module evaluates the inverse square root of the second moment matrix and uses it to update the shape adaptation matrix U. Fortunately, μ is a 2×2 symmetric matrix, which simplifies the computation of the inverse square root considerably.

The final step in the iteration is to evaluate the convergence and divergence of the feature. The feature is said to have converged if the ratio of the minimum

Table 1. Processing times for the hardware and MATLAB implementations for 2 iterations

Cornerness threshold	# of preliminary features	Time (s) software	Time (s) hardware	Ratio of software time to hardware time
500	11307	1.1057×10^3	0.1232	8.8765×10^3
1000	10768	1.0394×10^3	0.1173	8.8583×10^3
2000	9828	0.9558×10^3	0.1071	8.9200×10^3
3000	9016	0.8824×10^3	0.0983	8.9715×10^3
4000	9371	0.8267×10^3	0.0914	9.0480×10^3
5000	7821	0.7782×10^3	0.0854	9.1114×10^3
6000	7332	0.7373×10^3	0.0801	9.2034×10^3
7000	6905	0.7056×10^3	0.0755	9.3469×10^3
8000	6577	0.6738×10^3	0.0719	9.3666×10^3
9000	6280	0.6461×10^3	0.0687	9.4015×10^3
10000	6009	0.6230×10^3	0.0658	9.4705×10^3
12000	5588	0.5876×10^3	0.0612	9.5970×10^3
14000	5230	0.5562×10^3	0.0573	9.6994×10^3

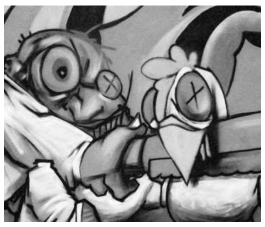
to maximum eigenvalues of μ is close to 1 and it is said to have diverged if the ratio of the eigenvalues of U is larger than a user-specified threshold.

It is worth noting that a single One Iteration module could be used to repeatedly process features by feeding the results of one pass through the module back into the module. This limits the throughput of the system, but allows for scalability in cases where there are not enough FPGA resources to pipeline several iteration blocks.

4 Results

The hardware system was compared to a MATLAB implementation of the algorithm in terms of speed and accuracy of the results. Table 1 shows the processing times for the hardware system and the floating-point MATLAB implementation for the case of the graffiti image 3(a). The times are quoted beside the corresponding value of the *cornerness threshold* parameter used to obtain them, and the number of preliminary features detected in the Multiscale Harris stage for these thresholds. The processing times for the floating-point implementation and the number of preliminary features were measured directly from MATLAB running on a 2.66 GHz Pentium IV processor with 4 GB of memory. The processing times for the hardware system were computed for clock rates of 12 MHz (for the Multiscale Harris module) and 60 MHz (for the iteration blocks).

Table 1 shows that the hardware implementation achieves a speed that is on average 9.2×10^3 times faster than the speed of the MATLAB version. In general, MATLAB implementations are not the most efficient in terms of speed, however the majority of operations involved in the feature detector algorithm are matrix operations, which are highly optimized in MATLAB. For this reason, it is expected that software implementations in other languages, like C, would improve the performance of the MATLAB version by up to a factor of 100.


(a) graffiti

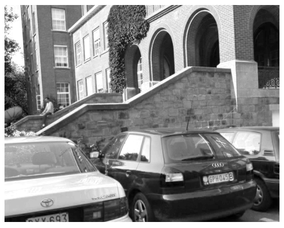

(b) cars

Fig. 3. Test images

Taking this improvement into account, the hardware implementation achieves an increase in speed of at least two orders of magnitude.

The current software implementation made available by Mikolajczyk [12] can process 1299 preliminary points (identified as “cgood” in the output of the executable file) in 3.55 seconds, to produce 1033 features. In this test, we used the same image and computing platform as those used to obtain the results in Table 1. Unfortunately, the number of iterations and other implementation details are not available to draw a precise comparison between this software implementation and the hardware system. However, a rough estimate is that the hardware implementation is 100 times faster than this software version for the same number of detected points.

The fixed-point representation and other algorithmic simplifications used in the hardware implementation of the feature detector introduce errors in the location, scale and shape of the detected features. To provide a quantitative measure of these errors, the results obtained from ModelSim simulations of two test images, shown in Figure 3, were compared against the results obtained from processing the same images with the floating-point MATLAB implementation. Each image was analyzed at three scales: $\sigma_{N0} = 1$ pixel, $\sigma_{N0} = 2$ pixels and $\sigma_{N0} = 4$ pixels.

Figure 4 shows the distribution of Euclidian distances between the features detected in hardware and the MATLAB features at the closest spatial location. The majority of hardware features are located within 3 pixels of a software feature. The tails of the histograms correspond to hardware features that did not have a match among the software features.

Figure 5 presents the distribution of the absolute value of the difference between the scales of corresponding features, for each test case. The histograms show that the errors in scale are small, in particular when compared to the the step between scales of $\varepsilon = 1.125$, and can be reasonably attributed to quantization errors in the hardware implementation.

Table 2 lists the total resource utilization per FPGA. The numbers in parenthesis indicate the percentage of the total resources of that kind available in one Stratix S80 FPGA. For reference, a Stratix S80 FPGA includes 79040 logic elements, 176 9-bit DSP elements and 7427520 bits of memory.

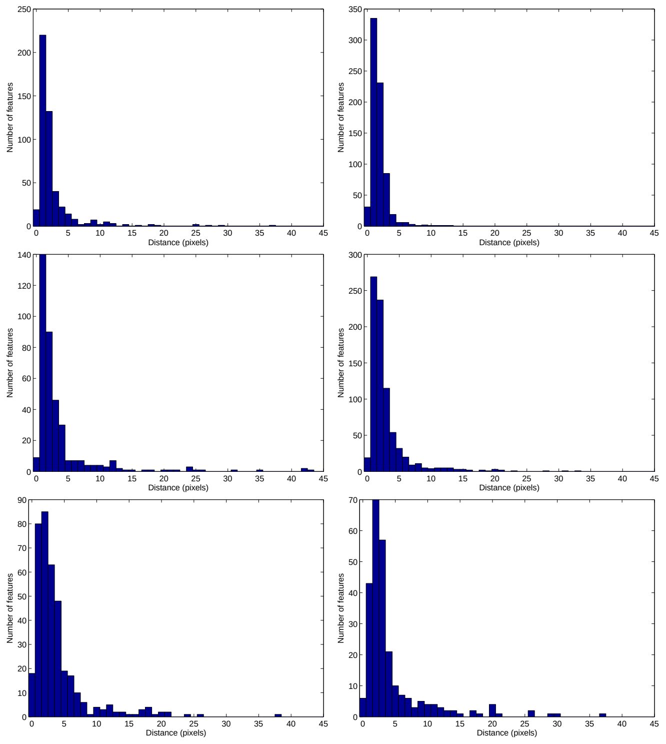


Fig. 4. Distribution of Euclidian distances between matching hardware and software features in the Graffiti (left) and Cars (right) image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

5 Conclusions

Implementations of image-processing tasks in integrated circuits can often achieve significantly faster processing times than implementations of the same tasks in general-purpose processors, even when clocked at much lower rates. The FPGA-based feature detector presented in this paper, for example, achieves processing speeds 90-9000 times the speed of an equivalent software implementation. Among integrated circuits, FPGAs offer a good compromise between area, speed and flexibility, since they can be easily reprogrammable.

The Harris-Affine algorithm relies to a great extent on the accuracy of intermediate results to achieve convergence of the location, scale and shape matrix of a feature point. Therefore, the main challenge when implementing the algorithm in hardware is to balance the need for numerical precision with an efficient use of the available hardware resources. Another important consideration in the design of the system is that the amount of data that needs to be processed by the iterative portion of the algorithm can vary significantly depending on the nature of the images being processed and the value of the input parameters. This poses

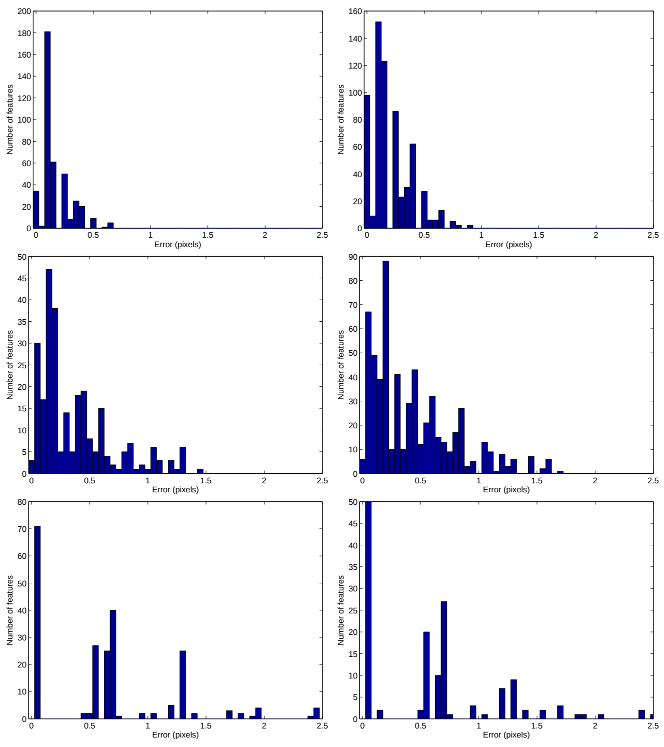


Fig. 5. Distribution of errors in scale between corresponding hardware and software features in the Graffiti (left) and Cars (right) images, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

a challenge from the hardware perspective because the system has to be capable of handling both very long and very short delays.

One of the goals in the design was to allow for scalability. Whenever possible the bit-widths of the parameters and intermediate signals have been defined as generic parameters, to allow for easier modification in case a different precision is required. As well, modules that operate on several scales in parallel consist of identical structures that are "stacked", so that more scales can be added with as few modifications as possible if more hardware resources become available. Similarly, the design can be easily modified to process images of different resolutions, since only the line buffers in the Multi-scale Harris module depend on the image width. This translates to processing times that depend on the image size as *O(N)*, where N is the number of pixels in the image. More importantly, this design addresses the implementation of an iterative process and presents a highly-reprogramable functional block (One Iteration) that can be repeated in a pipelined architecture or used in isolation within a feedback loop depending on the available hardware resources.

Table 2. Resource utilization per FPGA

FPGA	Logic elements	Memory bits	DSP elements
FPGA 0	38256 (48%)	3527389(47%)	126 (72 %)
FPGA 1	55028 (70 %)	5045677 (68%)	112 (64%)
FPGA 2	37036 (47 %)	4159329 (56%)	25 (14%)
FPGA 3	70159179 (89 %)	4437738 (60%)	117 (66%)

To the authors' knowledge, an implementation of a complex iterative algorithm in reprogrammable hardware has not been presented before in the computer vision literature. As such, this contribution aims to expand the understanding of the capabilities of reprogrammable hardware in real-world computer vision applications, where the size, speed and clock rates of a general-purpose processor may not be convenient.

The authors would like to thank the Canadian Foundation for Innovation, Ontario Centres of Excellence and Altera Corporation for financial support.

References

1. Mikolajczyk, K., Schmid, C.: An affine invariant interest point detector. In: European Conference on Computer Vision. Volume 4. (2002) 128–142
2. Mikolajczyk, K., Schmid, C.: Scale & affine invariant interest point detectors. International Journal of Computer Vision **60**(1) (2004) 63–86
3. Harris, C., Stephens, M.: A combined corner and edge detector. In: Proceedings Fourth Alvey Vision Conference, Manchester, United Kingdom (1988) 147–151
4. Lowe, D.G.: Object recognition from local scale-invariant features. In: Proceedings of the Seventh International Conference on Computer Vision, Kerkyra, Greece (1999) 1150–1157
5. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision **60** (2004) 90–110
6. Lindeberg, T.: Feature detection with automatic scale selection. International Journal of Computer Vision **30**(2) (1998) 79–116
7. Benedetti, A., Perona, P.: Real-time 2-D feature detection on a reconfigurable computer. In: Proceedings of the IEEE Conference on Computer Vision & Pattern Recognition. (1998)
8. Giacon, P., Saggin, S., Tomasi, G., Busti, M.: Implementing DSP algorithms using Spartan-3 FPGAs. Xcell Journal **Issue 53** (2005) 22–25
9. Se, S., Barfoot, T., Jasiobedzki, P.: Visual motion estimation and terrain modeling for planetary rovers. In: 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space, Munich (2005)
10. Mikolajczyk, K., Schmid, C.: Indexing based on scale invariant interest points. In: IEEE International Conference on Computer Vision. (2001) 525–531
11. Fender, J.: The Transmogrifier-4 Project (2005) [Online]. Available http://www.eecg.toronto.edu/˜tm4.
12. Mikolajczyk, K.: Harris-Affine and Hessian-Affine detector (2007) [Online]. Available http://www.robots.ox.ac.uk/ vsg/research/affine/detectors.html#binaries.