Real-time covariance tracking algorithm for embedded systems

Andrés Romero Mier y Terán[1] and Lionel Lacassagne[1]
Laboratoire de Recherche en Informatique
Bat 650, University Paris Sud
Email: andres.romero@lri.fr
lionel.lacassagne@lri.fr

Ali Hassan Zahraee[2]
[2] Institut d'Electronique Fondamentale
Bat 220, Univertsity Paris Sud
Email: ali.hassan@u-psud.fr

Michèle Gouiffès[3]
[3] Laboratoire d'Informatique
pour la Mécanique
et les Sciences de l'Ingénieur
Bat 508, University Paris Sud
Email: michele.gouiffes@u-psud.fr

***Abstract*—Covariance matching techniques have recently grown in interest due to their good performances for object retrieval, detection and tracking. By mixing color and texture information in a compact representation, it can be applied to various kinds of objects (textured or not, rigid or not). Unfortunately, the original version requires heavy computations, and is difficult to execute in real-time on embedded systems. This article presents a review on different versions of the algorithm and its various applications. Then, a comprehensive study is made to reach higher level of performance on multi-core CPU architectures, by comparing different ways to structure the information, using SIMD instructions and advanced loop transformations. The execution time is reduced significantly on two dual-core CPU architectures for embedded computing: PandaBoard with ARM Cortex-A9 and an Intel Ultra Low voltage U9300. According to our experiments on Covariance Tracking (CT), it is possible to reach a speedup of $\times$2.2 on ARM Cortex-A9 and $\times$2.8 on Intel U9300, when compared to the original algorithm, leading to real-time execution.**

I. Introduction

Tracking consists in estimating the evolution in state (*e.g.*, location, size, orientation) of a moving target over time. This process is often subdivided into two other subproblems: detection and matching. Detection deals with the difficulties of generic object recognition *i.e.*, finding instances from a particular object class or semantic category (*e.g.*, humans, faces, vehicles) registered in digital images and videos. On the other hand, matching methods provide the location which maximizes the similarity with the objects previously detected in the sequence. Generic object recognition requires models that cope with the diversity of instances appearances and shapes. This is generally made by learning techniques and classification. Conversely, matching algorithms analyze particular information and construct discriminative models that allow to disambiguate different instances from the same category and avoid confusions.

The main difficulty of tracking is to trace target trajectories and adapt to changes of appearance, pose, orientation, scale and shape. Since the beginnings of computer vision a diversity of tracking methods have been proposed, some of them construct path and state evolution estimations using a Bayesian framework (*e.g.*, particle filters, hidden Markov models), others measure the perceived optical flow in order to determine object displacements and scale changes (median flow) [7]. Exhaustive appearance-based methods compare a dense set of overlapping candidate locations to detect the one that fits best with some kind of template or model. When *a priori* information about the target location and its dynamics (*e.g.*, speed and acceleration) is available, the number of comparisons can be reduced enormously by giving preference to the more likely target regions. Other accelerations can be achieved using local searches that are based on gradient-descent algorithms able to handle small target displacements and geometrical changes. Among these approaches, feature points tracking techniques are very popular [9] since points can be extracted in most scenes, contrary to lines or other geometric features. Because they represent very local patterns, their motion models can be assumed as rigid and be estimated in a very efficient way. This method, as well as block matching, are raw-pixel methods, since the target is directly represented by its pixels matrix.

In order to deal with non-rigid motion, kernel-based methods such as Mean-Shift (*MS*) [2] [4] use a representation based on color or texture distribution.

Covariance tracking (*CT*) [13] is a very interesting and elegant alternative which offers a compact target representation based on the spatial correlation of different features computed at each pixel in the target bounding box. Very satisfying tracking performances have been observed for diverse kinds of objects (*e.g.* with rigid motion or not, with texture or not) . *CT* has been studied extensively, and many feature configurations and arrays of covariance descriptors have been proposed to improve target discrimination [8], [21] , [5], [11], [1] and [14]. Smoother trajectories can be obtained by considering target dynamics, therefore they increase tracking accuracy and reduce the search space [17], [18]. Genetic algorithms [20] can also be used to accelerate the convergence towards the optimal solution of the best candidate position, considering a search in a large image. But, to our knowledge, little work has been done to analyze the computational demands of *CT* and its portability to embedded systems. The goal of this article is to fill this gap, analyze the algorithm's computational behavior for different implementations and measure their load on embedded architectures.

The article is structured as follows: the covariance tracking *CT* is explained in Section II, details about the *CT* optimizations proposed to achieve a higher level of performance by accelerating the kernel of the algorithm are discussed in Section III. Experiments and details about the algorithm implementation are presented in Section IV and finally our conclusions are given in Section V.

II. Covariance matrices as image region descriptors

Let I represent a luminance (grayscale) or a color image with three channels and consider a rectangular region of size $W \times H$ (it can be the bounding box of the target to be tracked for example). Let F be the $W \times H \times d$ dimensional feature image extracted from I

$$F_{uv} = F(\boldsymbol{p}_{uv}) = \phi(I, \boldsymbol{p}_{uv}) \text{ with } \boldsymbol{p}_{uv} = (x_u, y_v) \quad (1)$$

where ϕ is any d-dimensional mapping forming a feature vector for each pixel of the bounding box. The features can be spatial coordinates $\boldsymbol{p}_{uv}$, intensity, color (in any color space), gradients, filter responses, or any possible set of images obtained from I. Now, let $\{\mathbf{z}_k\}_{k=1\cdots n}$ be a set of d-dimensional feature vectors inside the rectangular region $R \subset F$ of n pixels. Concerning notations, $\boldsymbol{p}_{uv}$ stands for the pixel at u^{th} raw and v^{th} column.

The region R is represented with the $d \times d$ covariance matrix

$$\mathbf{C}_R = \frac{1}{n-1} \sum_{k=1}^{n} (\mathbf{z}_k - \boldsymbol{\mu})(\mathbf{z}_k - \boldsymbol{\mu})^T \quad (2)$$

where $\boldsymbol{\mu}$ is the mean feature vector computed on the n points.

The covariance matrix is a $d \times d$ matrix which fuses multiple features naturally by measuring their correlations. The diagonal terms represent the variance of each feature, while elements outside this diagonal are the correlations. Thanks to the averaging in the covariance computation, noisy pixels are largely filtered out, which is an interesting advantage when compared to raw-pixel methods. Covariance matrices are more compact than most classical object descriptors. Indeed, due to symmetry, $\mathbf{C}_R$ has only $(d^2 + d)/2$ different values whatever the size of the target. To some extent, it is robust against scale changes, because all values are normalized by the size of the object, and against rotation when the locations coordinates $\boldsymbol{p}_{u,v}$ are replaced by the distance to the center of the bounding box.

The covariance descriptor ceases to be rotationally invariant when orientation information is introduced in the feature vector such as the norm of gradients with respect to x and y directions. The information considered by the covariance descriptor should be adapted to problem at hand, because they depend on the application, as described in next paragraph.

A. Covariance descriptor feature spaces

Covariance descriptors have been used in computer vision for object detection [16], re-identification [1], [14] and tracking [13]. The recommended set of features to use depends significantly on the application and the nature of the object: tracking faces is different than tracking pedestrians because faces are somehow more rigid than pedestrians that have more articulations. Color is an important hint for pedestrian or vehicle tracking/re-identification because of their clothes or bodywork color. But color is less significant to re-identify or track faces because the set of colors they exhibit is relatively limited.

Table I displays a summary of the more common feature combinations used by covariance descriptors in computer vision. The most obvious ones are the components from different color spaces such as RGB and HSV. Pixel brightness in the gray-scale image I and its local directional gradients as absolute values $|I_x|$ and $|I_y|$, gradient magnitude $\sqrt{I_x^2 + I_y^2}$ and its angle calculated as $\arctan \frac{|I_x|}{|I_y|}$. Foreground images $\mathbf{G}$ resulting from background subtraction methods and its gradients $\mathbf{G}_x$ and $\mathbf{G}_y$. Features $g_{00}(x, y)$ to $g_{74}(x, y)$ represent the 2-D Gabor kernel as a product of an elliptical Gaussian and a complex plane wave [11].

Some texture analysis and tracking methods use local binary patterns (*LBP*) in the place of Gabor filters because it is more simple and economical. Values Var_{LBP}, LBP_{θ_0} and LBP_{θ_1} in Table I represent local binary pattern variance (which is a classical property of the *LBP* operator [12]) and the angles defined by them, as detailed in [15] respectively. This version of the feature vector has shown very good performances for tracking, both in terms of robustness and computation times, and requires a shorter vector when compared to Gabor filters. In the rest of the paper, for the algorithmic optimization, a vector of seven features is considered.

Now, let us detail the computation of the covariance descriptor.

B. Covariance descriptor computation

From (2), the (i, j)-th element of the covariance matrix is

$$\mathbf{C}_R(i, j) = \frac{1}{n-1} \sum_{k=1}^{n} (z_k(i) - \mu(i))(z_k(j) - \mu(j)). \quad (3)$$

Expanding the means and rearranging the terms we have

$$\mathbf{C}_R(i, j) = \frac{1}{n-1} \left[\sum_{k=1}^{n} z_k(i) z_k(j) - \frac{1}{n} \sum_{k=1}^{n} z_k(i) \sum_{k=1}^{n} z_k(j) \right]. \quad (4)$$

The covariance in a given region depends on the sum of each feature dimension $z(i)_{i=1\cdots n}$, as well as the sum of the multiplications of any pair of features $z(i)z(j)_{i,j=1\cdots n}$, requiring in total $d + d^2$ integral images, one for each feature dimension $z(i)$ and one for the multiplication of any pair of feature dimensions $z(i)z(j)$.

Let A be a $W \times H \times d$ tensor of the integral images of each feature dimension

$$A_{uv}(i) = \sum_{\boldsymbol{p} \in R(11,uv)} F_{uv}(i) \text{ for } i = i \cdots d, \quad (5)$$

where $R(11, uv)$ is the region bounded by the top-left image corner $\boldsymbol{p}_{11} = (1, 1)$ and any other point in the image $\boldsymbol{p}_{uv} =$

Application	Feature set $\phi(I, \boldsymbol{p})$ with $\boldsymbol{p} = (x, y)$
Face tracking and recognition [11]	$\left[\begin{array}{ccccccc} x & y & \lvert I_x\rvert & \lvert I_y\rvert & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & I & \lvert I_x\rvert & \lvert I_y\rvert & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert & \theta(x,y) \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & I & g_{00}(x,y) & g_{01}(x,y) & \cdots & g_{74}(x,y) \end{array}\right]$
Pedestrian detection [16], [19]	$\left[\begin{array}{ccccccc} x & y & \lvert I_x\rvert & \lvert I_y\rvert & \sqrt{I_x^2+I_y^2} & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert & arctan\frac{\lvert I_x\rvert}{\lvert I_y\rvert} \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & \lvert I_x\rvert & \lvert I_y\rvert & \sqrt{I_x^2+I_y^2} & arctan\frac{\lvert I_x\rvert}{\lvert I_y\rvert} & \mathbf{G} & \sqrt{\mathbf{G}_x^2+\mathbf{G}_y^2} \end{array}\right]$
Pedestrian tracking [1], [13], [14], [16] and [15]	$\left[\begin{array}{ccccccc} x & y & R & G & B & \lvert I_x\rvert & \lvert I_y\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & R & G & B & \lvert I_x\rvert & \lvert I_y\rvert & \lvert I_{xx}\rvert & \lvert I_{yy}\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & H & S & V & \lvert I_x\rvert & \lvert I_y\rvert \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & R & G & B & \mathrm{Var}_{LBP} \end{array}\right]$
	$\left[\begin{array}{ccccccc} x & y & I & \sin(LBP_{\theta_0}) & \cos(LBP_{\theta_0}) & \sin(LBP_{\theta_1}) & \cos(LBP_{\theta_1}) \end{array}\right]$

TABLE I
FEATURES CONSIDERED BY THE COVARIANCE DESCRIPTOR DEPENDING ON THE APPLICATION.

(x_u, y_v). In a general way, let $R(uv, u'v')$ be the rectangular region defined by the top-left point $\boldsymbol{p}_{uv}$ and the right-bottom point $\boldsymbol{p}_{u'v'}$.

Similarly, the tensor containing the feature product-pair integral images is denoted as

$$B_{uv}(i,j) = \sum_{\boldsymbol{p} \in R(11,uv)} F_{uv}(i) F_{uv}(j) \text{ for } i,j = i \cdots d. \quad (6)$$

Now, for any point $\boldsymbol{p}_{uv}$, let $\mathbf{A}_{uv}$ be a d dimensional vector and $\mathbf{B}$ a $d \times d$ dimensional matrix such as

$$\begin{aligned} \mathbf{A}_{uv} &= [A_{uv}(1) \cdots A_{uv}(d)]^T \\ \mathbf{B}_{uv} &= \begin{pmatrix} B_{uv}(1,1) & \cdots & B_{uv}(1,d) \\ & \vdots & \\ B_{uv}(d,1) & \cdots & B_{uv}(d,d) \end{pmatrix}, \end{aligned} \quad (7)$$

The covariance of the region bounded by $(1,1)$ and $\boldsymbol{p}_{uv}$ is

$$\mathbf{C}_R(11, uv) = \frac{1}{n-1}\left[\mathbf{B}_{uv} - \frac{1}{n}\mathbf{A}_{uv}\mathbf{A}_{uv}^T\right], \quad (8)$$

where n is the number of pixels in the R under investigation. Similarly and after some algebraic manipulations, the covariance of the region $R(uv, u'v')$ is

$$\begin{aligned} \mathbf{C}_{R(uv,u'v')} = \tfrac{1}{n-1}\Big[&(\mathbf{B}_{u'v'} + \mathbf{B}_{uv} - \mathbf{B}_{u'v} - \mathbf{B}_{uv'}) \\ &-\tfrac{1}{n}\left(\mathbf{A}_{u'v'} + \mathbf{A}_{uv} - \mathbf{A}_{uv'} - \mathbf{A}_{u'v}\right) \cdot \\ &\left(\mathbf{A}_{u'v'} + \mathbf{A}_{uv} - \mathbf{A}_{uv'} - \mathbf{A}_{u'v}\right)^T\Big], \end{aligned} \quad (9)$$

After constructing the integral images the covariance of any rectangular region can be computed in $O(d^2)$ time regardless of the size of the region $R(uv, u'v')$. The complete process is represented graphically in Figure 1.

Next paragraph explains the covariance matching process.

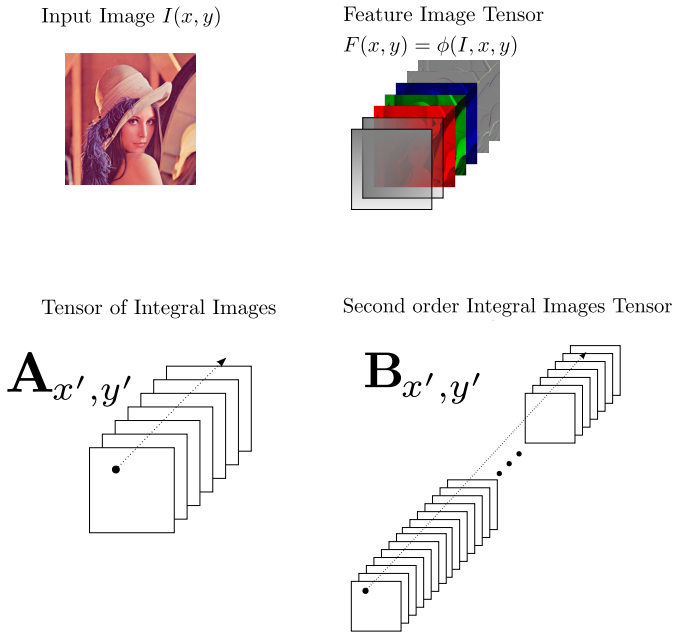

Fig. 1. Covariance descriptor computation: the image is first decomposed into an array of feature images (feature image tensor) applying the feature map $F_{uv} = \phi(I, \boldsymbol{p}_{uv})$. Then the crossed-products of these features are computed, using these arrays, the tensor integral images $A_{u'v'}(i)$ and the second order integral images tensor $B_{u'v'}(i,j)$ are computed.

C. Covariance matching and tracking

Covariance models and instances can be compared and matched using a simple nearest neighbor approach *i.e.* by finding the covariance descriptors that best resembles a model. The problem is that covariance matrices (*SPD* matrices in general) do not lie on the Euclidean space and many common and widely known operations in Euclidean spaces are not applicable or require to be adapted (*e.g.*, a *SPD* matrix multiplied

by a negative scalar is no longer a valid *SPD* matrix). A $n \times n$ *SPD* matrix only has $n \times (n+1)/2$ different elements, while it is possible to vectorize them and perform element-by-element subtraction, this approach provides very poor results as it fails to analyze the correlations between variables and the patterns stored in them. A solution to this problem is proposed in [3] where a dissimilarity measure between two covariance matrices is given as

$$\rho(\mathbf{C}_1, \mathbf{C}_2) = \sqrt{\sum_{i=1}^{n} \ln^2 \lambda_i(\mathbf{C}_1, \mathbf{C}_2)} \tag{10}$$

where $\{\lambda_i(\mathbf{C}_1, \mathbf{C}_2)\}_{i=1,\cdots,n}$ are the generalized eigenvalues of $\mathbf{C}_1$ and $\mathbf{C}_2$ computed from

$$\lambda_i \mathbf{C}_1 \mathbf{x}_i - \mathbf{C}_2 \mathbf{x}_i = 0 \qquad i = 1, \cdots, d. \tag{11}$$

In (11), $\mathbf{x}_i \neq 0$ are the generalized eigenvectors. Distance measure (10) satisfies the metric axioms for *SPD* matrices $\mathbf{C}_1$ and $\mathbf{C}_2$

1. $\rho(\mathbf{C}_1, \mathbf{C}_2) \geq 0$ and $\rho(\mathbf{C}_1, \mathbf{C}_2) = 0$ only if $\mathbf{C}_1 = \mathbf{C}_2$,
2. $\rho(\mathbf{C}_1, \mathbf{C}_2) = \rho(\mathbf{C}_2, \mathbf{C}_1)$
3. $\rho(\mathbf{C}_1, \mathbf{C}_2) + \rho(\mathbf{C}_1, \mathbf{C}_3) \geq \rho(\mathbf{C}_2, \mathbf{C}_3)$.

(12)

The tracking starts in the first frame of the sequence, by computing the covariance matrix $\mathbf{C}_1$ in the bounding box of the target under consideration. The initial detection is not detailed in this paper, since it can be made in various ways, by object recognition or background subtraction for example. Then, the tracking consists in finding the new location of the target in the successive frames, by finding the covariance matrix $\mathbf{C}_2$ minimizing the dissimilarity (10).

In the remainder of the paper, the search in made exhaustively in a large area around the previous location of the object. There are two main reasons for that. First, the computation times are more predictable than the steepest descent method, and the results of next section concerning execution times will correspond to the worst case. Second, the exhaustive search allows the matching of a target undergoing brutal motion, or can retrieve the target after occlusion.

III. Covariance Tracking Algorithm Analysis and Optimizations

A REFAIRE Three strategies are studied to optimize the covariance tracking CT on multi-core CPUs. The first one is based on SoA→AoS transformation. The second one consists in architectural optimizations: either multi-threading the SoA version with OpenMP middleware or using SIMD instructions (SSE for Intel, Neon on ARM) for the AoS version. The third one is the Loop-Fusion transform.

A. SoA→AoS

The goal of *SoA→AoS* transform (Structure of Arrays to Array of Structures) consists in transforming a set of independent arrays into one array, where each cell is a structure combining the elements of each independent array. The contribution of such a transform is to leverage the cache performance by enforcing spatial and temporal cache locality. Let us define the following notations:

- h and w the height and width of the image
- n_F the number of features,
- n_P, the number of products of features, that is $n_P = n_F(n_F+1)/2$,
- F a cube (*SoA*) or matrix (*AoS*) of features,
- P a cube (*SoA*) or matrix (*AoS*) of feature products,
- I_F and I_P two cubes (or matrices) of integral images (from F or P),

Here we want to optimize the locality of the features (or the product of features) of a given point of coordinates (i, j). In *SoA* version, we have two cubes F_{SoA} of size $n_F \times h \times w$ and P_{SoA} of size $n_P \times h \times w$. In *AoS* we have two matrices F_{AoS} and P_{AoS} of size $h \times (w \cdot n_F)$ and $h \times (w \cdot n_P)$.

In our case, the *SoA→AoS* transform consists in swapping the loop nests and changing the addressing computations from a 3D-form like $cube[k][i][j]$ into a 2D-form like $matrix[i][j \times n + k]$, where n is the structure cardinal (here n_F or n_P).

The covariance tracking algorithm is composed of three stages:

1) point-to-point products computation of all features,
2) the integral image computation of features,
3) the integral image computation of products.

The product of features and its transformation are described in algorithms 1 and 2. Thanks to commutativity of the multiplication, only half of the products have to be computed (the loop on k_2 starts at k_1, line 3). As the two last stages are similar, we only present a generic version of integral image computation (Algo. 3) and its transformation (Algo. 4).

Algorithm 1: product of features - *SoA* version

```
1 k ← 0
2 foreach k1 ∈ [0..nF − 1] do
3     foreach k2 ∈ [k1..nF − 1] do
4         foreach i ∈ [0..h − 1] do
5             foreach j ∈ [0..w − 1] do
6                 P[k][i][j] ← F[k1][i][j] × F[k2][i][j]
7                 k ← k + 1
```

Algorithm 2: product of features - *AoS* version

```
1 foreach i ∈ [0..h − 1] do
2     foreach j ∈ [0..w − 1] do
3         k ← 0
4         foreach k1 ∈ [0..nF − 1] do
5             foreach k2 ∈ [k1..nF − 1] do
6                 P[i][j × nP + k] ←
                  F[i][j × nP + k] × F[i][j × nP + k]
7                 k ← k + 1
```

Algorithm 3: integral image - *SoA* version, $n \in \{n_F, n_P\}$

```
1 foreach k ∈ [0..n − 1] do
2     foreach i ∈ [0..h − 1] do
3         foreach j ∈ [0..w − 1] do
4             I[k][i][j] ← I[k][i][j] + I[k][i][j − 1] + I[k][i −
              1][j] − I[k][i − 1][j − 1]
```

Algorithm 4: integral image - *AoS* version, $n \in \{n_F, n_P\}$

```
1 foreach i ∈ [0..h − 1] do
2     foreach j ∈ [0..w − 1] do
3         foreach k ∈ [0..n − 1] do
4             I[i][j×n+k] ← I[i][j×n+k]+I[i][(j−1)×n+
              k]+I[i−1][j×n+k]−I[k][i−1][(j−1)×n+k]
```

B. SIMD and OpenMP

Once this transform is done, one can also apply SIMD to the different parts of the algorithm. For the product part, the two internal loops on k_1 and k_2 are fully unrolled in order to show the list of all multiplications and the list of vectors to construct through permutation instructions (*e.g.*, `_mm_shuffle_ps` in SSE). For example, for a typical value of $n_F = 7$, there are $n_P = 28$ products. The associated vectors are (the numbers are the feature indexes):

$$\begin{aligned}
[P_0, P_1, P_2, P_3] &= [F_0, F_0, F_0, F_0] \times [F_0, F_1, F_2, F_3] \\
[P_4, P_5, P_6, P_7] &= [F_0, F_0, F_0, F_1] \times [F_4, F_5, F_6, F_1] \\
[P_8, P_9, P_{10}, P_{11}] &= [F_1, F_1, F_1, F_1] \times [F_2, F_3, F_4, F_5] \\
[P_{12}, P_{13}, P_{14}, P_{15}] &= [F_1, F_2, F_2, F_2] \times [F_6, F_2, F_3, F_4] \\
[P_{16}, P_{17}, P_{18}, P_{19}] &= [F_2, F_2, F_3, F_3] \times [F_5, F_6, F_3, F_4] \\
[P_{20}, P_{21}, P_{22}, P_{23}] &= [F_3, F_3, F_4, F_4] \times [F_5, F_6, F_4, F_5] \\
[P_{24}, P_{25}, P_{26}, P_{27}] &= [F_4, F_5, F_5, F_6] \times [F_6, F_5, F_6, F_6]
\end{aligned}$$

In that case, the 7th vector is 100% filled, but it will become sub-optimal if n_P is not divisible by the cardinal of the SIMD register (4 with SSE and Neon). In SSE, some permutations can be achieved using only one instruction, the other need a maximum of two instructions. Because some permutations can be re-used to perform other permutations, it is possible to achieve a factorization over all the required permutations. For example with $n_F = 7$, fifteen shuffles are required. In Neon it is more complex. If some permutations can be done into 128-bits registers – that is with a parallelism of 4 – other permutations require instructions only available with 64-bit registers, like the *look-up table* instruction named `vtbl`. So in Neon, 128-bit float registers should be: 1) split into 64-bit registers, 2) type-casted into 64-bit integer registers, 3) permuted with `vtbl` instructions 4) type-casted into 64-bit float registers and 5) combined into 128-bit float registers. Finally it requires 48 SIMD Neon instructions to create the seven pairs of products.

The table II provides the algorithmic complexity and the amount of memory accesses for both scalar and SIMD (SSE and Neon) versions. It also provides the arithmetic intensity (AI) – popularized by Nvidia – that is the ratio between the number of operations (including the number of permutations for SIMD version) and the number of memory accesses.

Note that the scalar version has a low AI of 0.5 as the number of memory accesses is twice the number of operations. It can also be noticed that the SIMD version has $\times 2.7$ less operations for SSE (respectively 1.6 for Neon) and $\times 4.8$ less memory accesses, thus the AI ratios reache 0.9 for SSE and 1.5 for Neon.

Concerning OpenMP, the point is to evaluate *SOA+OpenMP* versus *AoS+SIMD*. Indeed, because for a common 4-core General Purpose Processor (GPP) the degree of parallelism with a multi-threaded version and with a SIMDized version is the same, *i.e.* four. Results are provided in cycles per point (*cpp*) *versus* the data amount (image size). The *cpp* is a normalized metric that help to detect *cache overflow* (when data do not fit in the cache): the curve of *cpp* increases significantly.

The three versions (SoA+OpenMP, AoS, AoS+SIMD) have been benchmarked on three generations of Intel processors: Penryn, Nehalem and SandyBridge for image size varying from 128×128 up to 1024×1024. It appears (Fig. 2) that a 4-threaded version is always slower than a 1-threaded SIMD version. Eight threads are required on the Nehalem to be faster. The reason is the low AI inducing a high stress on the architecture's buses and also because manipulating SOA requires $n_P = 28$ active references in the cache, that is more than the usual L2 or L3 associativity (24 on the Intel processor). In the next steps of this article, SIMDization is the only architectural optimization being considered as realistic.

C. Loop fusion

We have tested three versions with *loop-fusion* in order to increase the AI ratio by reducing the amount of memory accesses. The first one is a scalar parametric version (with n_F) that fuses the external `i`-loop and keeps the three `j`-loops unchanged. The second one is a specialized version with $n_F = 7$ where the three internal loops are fused together. The third one is the SIMDized version of the second one. The internal loop fusion allows to save the LOAD/STORE instructions in order to write a product of features into memory and to read it afterwards to compute the integral image of products. The Loop-Fusion has been done by hand, but some tools like PIPS [10] can do such kind of transformation automatically [6]. The complexity of scalar and SIMD versions are provided in table III

D. Embedded systems

Let us now focus on more *embedded* processors like the Intel ULV (Ultra Low Voltage) Penryn U9300 – that belongs to the Penryn family – and the ARM Cortex-A9. Their average power consumption (TDP) are respectively about 10 W and 1 W while running at close frequencies: 1.2 GHz and 1.0 Ghz.

instructions	MUL	ADD	LOAD	STORE	AI
AoS scalar version with 3 loops					
product of features	n_P	0	$2n_P$	n_P	-
integral of features	0	$3n_F$	$4n_F$	n_F	-
integral of products	0	$3n_P$	$4n_P$	n_P	-
total	n_P	$3(n_P + n_F)$	$6n_P + 4n_F$	$2n_P + n_F$	-
total with $n_P = n_F(n_F + 1)/2$	$2n_F^2 + 5n_F$		$4n_F^2 + 9n_F$		-
total with $n_F = 7$	133		259		0.5
AoS SIMD (with $n_F = 7$) version with 3 loops					
product of features	7	0	2	7	-
integral of features	0	21	28	7	-
integral of products	0	6	2	2	-
total SSE (+ 15 PERM)	49		54		0.9
total Neon (+ 48 PERM)	82		54		1.5

TABLE II
COMPLEXITY AND ARITHMETIC INTENSITY OF SCALAR AND SIMD VERSIONS WITHOUT LOOP-FUSION

instructions	MUL	ADD	LOAD	STORE	AI
AoS scalar version + Loop Fusion					
integral of features	0	$2n_F$	$2n_F$	n_F	-
integral product of features	n_P	$2n_P$	n_P	n_P	-
total	n_P	$2(n_P + n_F)$	$n_P + 2n_F$	$n_P + n_F$	-
total with $n_P = n_F(n_F + 1)/2$	$1.5n_F^2 + 3.5n_F$		$n_F^2 + 4n_F$		-
total with $n_F = 7$	98		77		1.3
AoS SIMD (with $n_F = 7$) version + Loop Fusion					
integral of features	0	4	4	2	-
integral product of features	7	14	7	7	-
total SSE (+ 15 PERM)	40		20		2.0
total Neon (+ 48 PERM)	73		20		3.7

TABLE III
COMPLEXITY AND ARITHMETIC INTENSITY OF SCALAR AND SIMD VERSIONS WITH LOOP-FUSION

The figure 3 provides the *cpp* for SoA, AoS, AoS+SIMD and also AoS+T and AoS+T+SIMD for the Intel U9300 and ARM Cortex-A9 (where T stands for Loop-Fusion Transform).

First, for both processors, the SoA version is very inefficient, compared to the best one (AoS+T+SIMD). Secondly, there are two big differences between them. The first one is impact of the optimizations. For the Cortex-A9, for images larger than 100×100, the only efficient optimization is the Loop-Fusion transform. The impact of Neon instructions is minor: AoS+SIMD is similar to AoS version, and AoS+T+SIMD is similar to AoS+T. It comes from the memory hierarchy of the Cortex and the number of extra instructions used to perform the permutations (48 for Neon *versus* only 15 for SSE). The second difference is *cpp* values: the Intel *cpp*'s are around $\times 4.5$ smaller than ARM ones, that comes from higher latency instructions.

If we now focus on the impact of Loop-Fusion (Tab. IV), we can see that the speedup with AoS version is about $\times 2$ for Intel and $\times 3.3$ for ARM, while the total speedups are $\times 5.3$ and $\times 3.4$. This optimization (combined with n_F specialization for loop-unwinding) is mandatory for this algorithm.

Concerning power efficiency, there is a factor 4 of speed between the Intel U9300 and the Cortex-A9. However, as the power consumption (TDP) of the Cortex is approximatively 10 times slower than the U9300, the Cortex-A9 is twice power efficient than the U9300.

	Intel U9300	ARM Cortex-A9
AoS / AoS+T	$\times 1.8$	$\times 3.3$
AoS+SIMD / AoS+T+SIMD	$\times 2.2$	$\times 3.2$
SoA / AoS+T+SIMD	$\times \mathbf{5.3}$	$\times \mathbf{3.4}$

TABLE IV
IMPACT OF LOOP-FUSION TRANSFORM: SPEEDUPS FOR U9300 AND CORTEX-A9

IV. ALGORITHM IMPLEMENTATION

Two sequences have been evaluated: Panda and Pedxing for which the robustness of the algorithm have been evaluated in [14] and [15]. for both of them, the execution times are given in *cpp* for each version of the algorithm: SoA is the basic version, and AoS++ stands for AoS transform + SIMDization + Loop-Fusion transform.

Two counter-intuitive results can be noticed. The first one is the features computation *cpp*: it is lower for SoA. The reason is obviously the memory layout of SoA (versus AoS) when computing the features and storing them into a cube or a matrix. The second counter-intuitive result is more interesting: it is the tracking part of the algorithm. The tracking is based

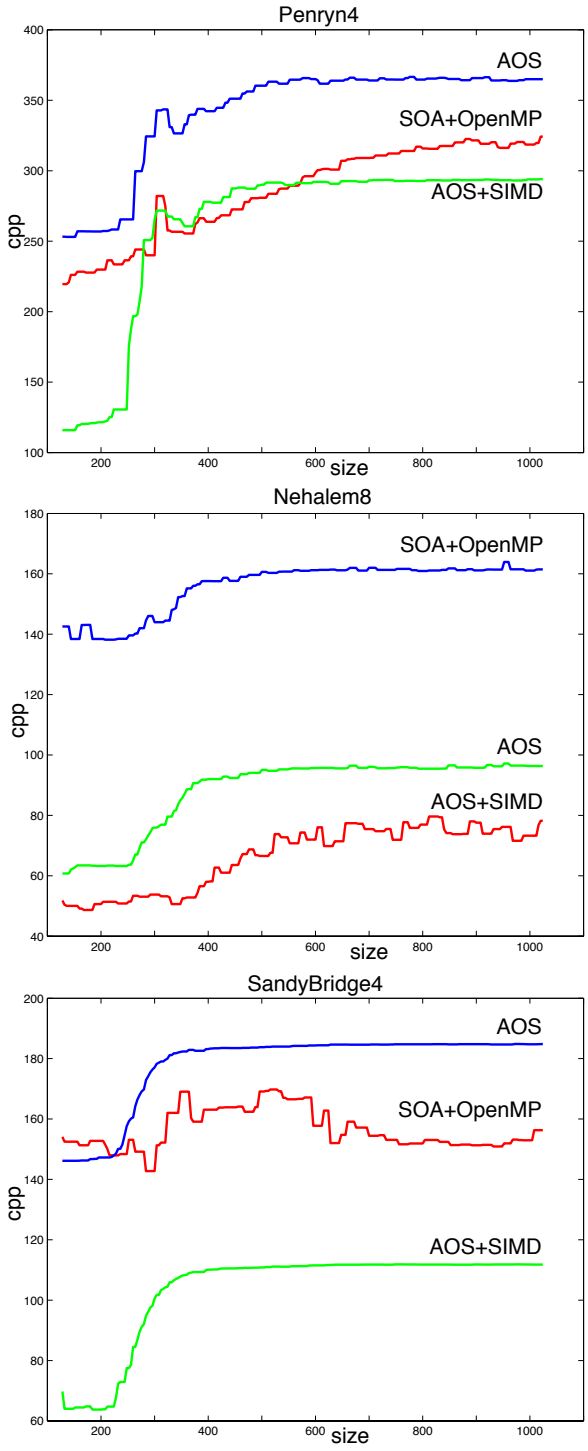

Fig. 2. Performance in *cpp* of a 1×4-core Penryn (top), 2×4-core Nehalem (middle), 1×4-core SandyBridge (bottom) for image sizes $\in [128..1024]$.

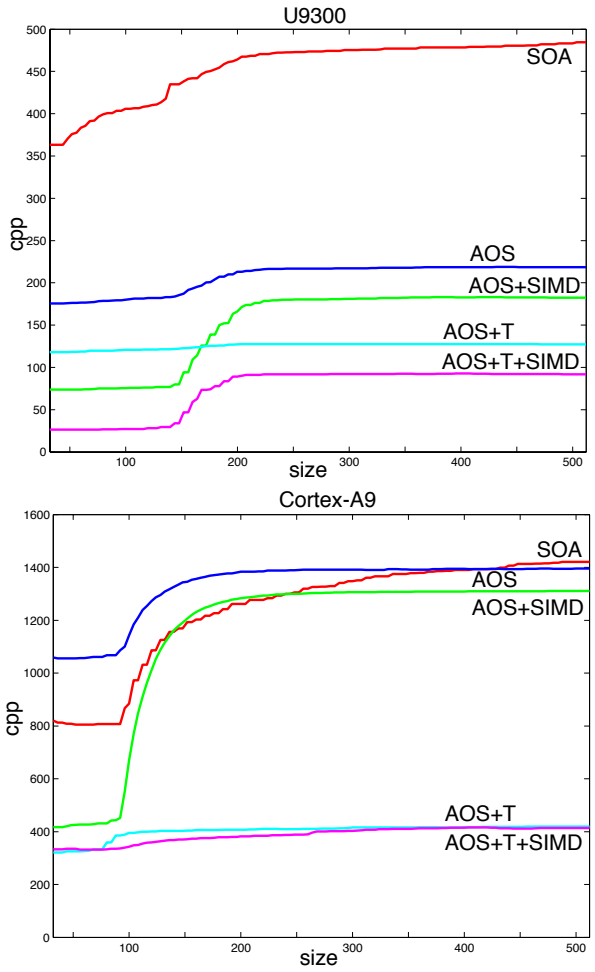

Fig. 3. Performance in *cpp* of an Intel ULV U9300 and an ARM Cortex-A9 for image sizes $\in [32..512]$

sequence	Panda		Pedxing	
size	312×233		640×480	
algorithm version	SoA	AoS++	SoA	AoS++
features computation (*cpp*)	128	150	128	150
kernel computation (*cpp*)	599	87	618	91
tracking (*cpp*)	23	23	11	11
total (*cpp*)	738	248	769	264
kernel / total	81 %	35 %	80 %	34 %
total speedup	$\times 2.9$		$\times 2.8$	
1-C execution time (ms)	45	15	197	68
2-C execution time (ms)	36	9	158	38

TABLE V
cpp AND EXECUTION TIME FOR INTEL U9300 ON 1 CORE AND 2 CORES

on the computation of a similarity criterion that requires the computation of the generalized eigenvalues, inversions and matrix logarithms (10). In order to have the same behavior we use Gnu Scientific Library to perform these computations on both plateforms, but we can also use Intel MKL or Eigen libraries. The future position is chosen by evaluating forty (in our case, but it is parameterizable) random positions in a research window, so matrix operations represent a high percentage of the tracking part. It appears that the features used for tracking lead to a "more" ill-conditioned matrix requiring more computations for Panda than for Pedxing sequence.

Concerning the acceleration, we can see (tables V and VI)

sequence	Panda		Pedxing	
size	312×233		640×480	
algorithm version	SoA	AoS++	SoA	AoS++
features computation (*cpp*)	461	461	486	486
kernel computation (*cpp*)	1491	395	1600	415
tracking (*cpp*)	96	96	19	19
total (*cpp*)	2048	952	2106	921
kernel / total	73 %	42 %	73 %	45 %
total speedup	$\times 2.2$		$\times 2.2$	
1-C execution time (ms)	149	69	647	283
2-C execution time (ms)	108	36	492	149

TABLE VI
cpp AND EXECUTION TIME FOR ARM CORTEX-A9 ON 1 CORE AND 2 CORES

that the optimization of the kernel provides a speedup of $\times 2.9$ for Intel and $\times 2.2$ for ARM that assets the need of all optimizations. As both processors have two cores, all the processing parts can be done either on one core (the execution time is the sum of all parts) or on two cores (the biggest part is on one core and the two other parts are on the second core). With such a coarse grain thread distribution, the Intel U9300 can track targets in real-time for 640×480 images, while the ARM Cortex-A9 can do it for image sizes up to 320×240. As said previously, there is a factor 10 for power consumption, and only a factor 4 in execution time. So for small images (up to 320×240) the Cortex-A9 is the best choice as it is real-time and more power efficient than the Intel. While for bigger sizes (up to 640×480), the Intel is the only choice for real-time implementation

V. CONCLUSION

We have presented the implementation of a robust covariance tracking algorithm, with a parameterizable complexity that can be adapted (number and nature of features) for trade-off between robustness and execution time. Classical software and hardware optimizations have been applied: SIMDization and Loop-Fusion transform combined with AoS-SoA transform to accelerate the kernel of the algorithm. These optimizations allow a real-time execution on *heavy* embedded systems like Intel U9300 and on *light* ones like the ARM Cortex-A9.

In the future we will also accelerate the features computation part of the algorithm and create a multi-threaded version of the algorithm in order to perform multi-target tracking. From a benchmarking point of view, we will evaluate Cortex-A15 and more power-efficient Intel processor. We will also focus on the choice of features and ill-conditioned matrix versus execution time. As far as we know, our implementation of the covariance tracking algorithm is the first real-time implementation for embedded systems, while being robust.

ACKNOWLEDGMENT

This article is supported by european project ITEA2 SPY (http://www.itea2-spy.org/).

REFERENCES

[1] S. Bak, E. Corvee, F. Bremond, and M. Thonnat. Multiple-shot human re-identification by mean riemannian covariance grid. In *Advanced Video and Signal-Based Surveillance (AVSS), 2011 8th IEEE International Conference on*, pages 179–184. IEEE, 2011.

[2] D. Comaniciu, V. Ramesh, and P. Meer. Kernel-based object tracking. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(5):564–577, 2003.

[3] W. Förstner and B. Moonen. A metric for covariance matrices. *Quo vadis geodesia*, pages 113–128, 1999.

[4] M. Gouiffès, F. Laguzet, and L. Lacassagne. Color connectedness degree for mean-shift tracking. In *Pattern Recognition (ICPR), 2010 20th International Conference on*, pages 4561–4564. IEEE, 2010.

[5] S. Guo and Q. Ruan. Facial expression recognition using local binary covariance matrices. In *Wireless, Mobile & Multimedia Networks (ICWMMN 2011), 4th IET International Conference on*, page 237–242, 2011.

[6] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *ICS*, pages 244–251, 1991.

[7] Z. Kalal, K. Mikolajczyk, and J. Matas. Tracking-learning-detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(7):1409–1422, 2012.

[8] P. Li and Q. Wang. Local log-euclidean covariance matrix (L2ECM) for image representation and its applications. In A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, editors, *Computer Vision – ECCV 2012*, volume 7574 of *Lecture Notes in Computer Science*, pages 469–482. Springer Berlin Heidelberg, 2012.

[9] B. Lucas, T. Kanade, et al. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence*, 1981.

[10] MINES-ParisTech. PIPS. http://pips4u.org, 1989–2009. Open source, under GPLv3.

[11] Y. Pang, Y. Yuan, and X. Li. Gabor-based region covariance matrices for face recognition. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(7):989 –993, july 2008.

[12] M. Pietikäinen. *Computer Vision Using Local Binary Patterns*. Computational Imaging and Vision. Springer London, 2011.

[13] F. Porikli, O. Tuzel, and P. Meer. Covariance tracking using model update based on lie algebra. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 728–735. IEEE, 2006.

[14] A. Romero, M. Gouiffès, and L. Lacassagne. Covariance descriptor multiple object tracking and re-identication with colorspace evaluation. In *Asian Conference on Computer Vision, 2012. ACCV 2012*, 2012.

[15] A. Romero, M. Gouiffès, and L. Lacassagne. Enhanced local binary covariance matrices ELBCM for texture analysis and object tracking. In *ACM International Conference Proceedings Series*. Association for Computing Machinery, 2013. To appear.

[16] O. Tuzel, F. Porikli, and P. Meer. Pedestrian detection via classification on riemannian manifolds. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(10):1713–1727, 2008.

[17] A. Tyagi, J. W. Davis, and G. Potamianos. Steepest descent for efficient covariance tracking. In *Motion and video Computing, 2008. WMVC 2008. IEEE Workshop on*, page 1–6, 2008.

[18] Y. Wu, B. Wu, J. Liu, and H. Lu. Probabilistic tracking on riemannian manifolds. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, page 1–4, 2008.

[19] J. Yao, J. Odobez, et al. Fast human detection from videos using covariance features. In *The Eighth International Workshop on Visual Surveillance-VS2008*, 2008.

[20] X. ZHANG, G. DAI, and N. XU. Genetic algorithms. a new optimization and search algorithms [j]. *CONTROL THEORY & APPLICATIONS*, 3, 1995.

[21] Y. Zhang and S. Li. Gabor-LBP based region covariance descriptor for person re-identification. In *Image and Graphics (ICIG), 2011 Sixth International Conference on*, page 368–371, 2011.