A Dataflow Object Detection System for FPGA-based Smart Camera

Cédric Bourrasset, Luca Maggiani, Jocelyn Sérot, François Berry

A Dataflow Object Detection System for FPGA-based Smart Camera

Cédric Bourrasset *, Luca Maggiani *†, Jocelyn Sérot*, François Berry*
* Institut Pascal, Universite Blaise Pascal, Clermont Ferrand, France
† TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

Embedded computer vision based smart systems raise challenging issues in many research fields, including real-time vision processing, communication protocols or distributed algorithms. The amount of data generated by cameras using high resolution image sensors requires powerful computing systems to be processed at digital video frame rates. Consequently, the design of efficient and flexible *smart cameras*, with on-board processing capabilities, has become a key issue for the expansion of smart vision systems relying on decentralized processing at the image sensor node level.

In this context, FPGA-based platforms, supporting massive data parallelism, offer large opportunities to match real-time processing constraints compared to platforms based on general purpose processors. In this paper, we describe the implementation, on such a platform, of a configurable object detection application, reformulated according to the dataflow model of computation. The application relies on the computation of the histogram of oriented gradients (HOG) and a linear SVM-based classification. It is described using the CAPH programming language, allowing efficient hardware descriptions to be generated automatically from high level dataflow specifications without prior knowledge of hardware description languages such as VHDL or Verilog. Results show that the performance of the generated code does not suffer from a significant overhead compared to handwritten HDL code.

I. Introduction

Traditional computer visions system often operate in a centralized manner, even for multi-camera applications, where the sequences of frames output by each camera are sent to a central computing unit. This central unit gathers information from all the available cameras and processes it in order to extract significant features. However as the number of source nodes increases, such a centralized approach quickly becomes infeasible because the central node becomes a bottleneck. This is specially true when high resolution cameras with high acquisition rates are deployed, for instance in object detection applications. In this context, and in the current state of network technology, the necessity to meet real-time processing constraints rules out any kind of centralized approach.

As a result, in the last years, many distributed video systems have been proposed. They aim at overcoming the above-mentioned bottleneck issue by distributing the computational intensive tasks on the camera nodes. Such nodes are generally called *Smart Cameras* (SC). Image processing capability is added by embedding processing units such as general purpose processors (GPP), specialized processors (DSP) or field programmable gate arrays (FPGA). The latter solution has drawn a lot of attention in the past years because it offers large opportunities for exploiting the fine grain, regular, parallelism that most of image processing applications exhibit at the lowest levels of processing. However, programming FPGA-based platforms is traditionally done using hardware description languages (HDLs) – see figure 1 – and therefore requires expertise in digital design. This, in practice, hinders the applicability of FPGA-based solutions.

As a response, a lot of work has been devoted in the past decade to the design and development of high-level languages and tools, aiming at allowing FPGAs to be used by programmers who are not experts in digital design, such as Catapult-C [1], Stream-C [2] or Impulse-C [3]. Most of these tools propose a direct conversion of C or C++ code into HDL (VHDL or Verilog). While attractive, this approach suffers from several drawbacks. First, C programs often rely on features which are difficult, if not impossible, to implement in hardware (dynamic memory allocation for instance). This means that code frequently has to be rewritten to be accepted by the compilers. Practically, this rewriting cannot be carried out without understanding why certain constructs have to be avoided and how to replace them by "hardware-compatible" equivalents. So a minimum knowledge of hardware design principles is actually required. Second, C is intrinsically sequential whereas hardware is truly parallel. In the current state-of-the-art, this cannot be done in a fully automatic way and the programmer is required to put annotations (pragmas) in the code to help the compiler, which

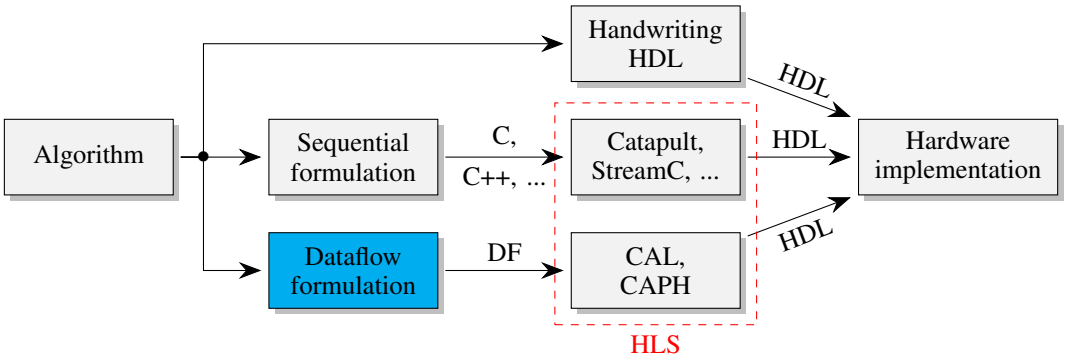


Fig. 1: Hardware design methods

adds to the burden. Finally, the code generally has to undergo various optimizations and transformations before the actual HDL generation. With most of the existing tools these transformations and optimizations require inputs from the programmer [4], who therefore must have a rather good knowledge in digital design.

As a result, theres still a gap between what can be described with a general purpose, Turing-complete, language and what can be efficiently and automatically implemented on an FPGA.

In this context, there has been recently a regained attention on approaches relying on *domain specific languages*, such as Caltrop Actor language (CAL) [5]. The idea is that the aforementioned gap can be reduced by departing from the classical, sequential and imperative model of computation underpinning C or C++ formulations of algorithms. The *dataflow* model of computation, in particular, has nice properties which makes it a good candidate for harnessing the complexity of FPGA programming. This has been noticed by Najjar *et al.* in [6] who, in 1999, already argued that the two major characteristics of this model - all data are values and all operations are purely functional - nicely fit the execution model of FPGAs. A few years before, Sérot *et al.* had already demonstrated in [7], [8] that dataflow could be used to exploit massively parallel computers dedicated to real-time image processing. The key idea was to describe an application as a graph of dataflow operators and then physically map this graph on a network of data-driven processing elements (DDPs). But at the time of these experiments, the FPGA technology was still in its infancy and building a complete reconfigurable computing system required either resorting to GPPs or to ASICs .The dramatic improvements in FPGA technology during the past decade have provided an unique opportunity to bring this old idea up to date.

The main focus of this paper is to support this claim, in other words to show how a purely dataflow reformulation of an algorithm allows it to be implemented very efficiently on a FPGA without resorting either to low-level hardware description languages or complex and inefficient C-based HLS languages. The proposed approach relies on the CAPH [9]–[11] programming language, which is able to produce efficient FPGA designs from high level dataflow descriptions.

It will be demonstrated by describing the implementation, on a FPGA-based smart camera platform, of a real-time object detection application. This application, re-formulated here according to the dataflow Model of Computation (MoC), associates a feature extraction step (computation of histograms of oriented gradients, HOGs) and a linear classification step in order to predict the presence of predefined object (such as pedestrians or vehicles) in videos.

The remainder of the paper is organized as follows. The proposed application is presented in Sec. II and its dataflow reformulation is given in Sec. III. Sec. IV described the implementation, using the CAPH language obtained from this reformulation. Results are presented and discussed in Sec. V. Sec. VI concludes and gives perspectives for future work.

II. Detection system

Figure 2 gives an overview of the proposed application. It involves two main steps : feature extraction and classification. The feature extraction (outlined in the dashed box) is composed of three modules (Gradient Extraction, Histogram, Block Normalization) and processes the sequence of images coming from the video

source. For each image, the extracted features are compared with a specific object model by the classification module to tell whether such objects are present or not in the image.

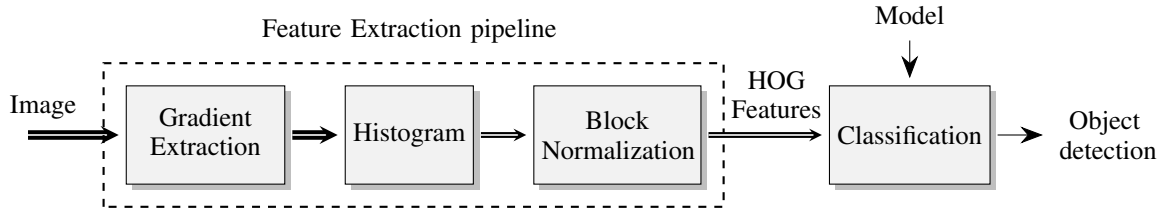


Fig. 2: Overview of the object detection application

A. Feature Extraction

The descriptor extracted from each image is an Histogram of Oriented Gradients (HOG), computed by evaluating well-normalized local histograms of image gradient orientations in a dense grid. It has been shown in [12] that this kind of descriptors provides excellent performances for visual object recognition. The computation of the HOG descriptors is sketched in Fig. 2. It consists in three steps.

The **first step** computes spatial derivatives G_x and G_y in x and y direction for each pixel (x, y) within image I. Gradient magnitude $G(x, y)$ and angle $\theta(x, y)$ are then obtained as :

$$G(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \tag{1}$$

$$tan\big(\theta(x,y)\big) = \frac{G_y(x,y)}{G_x(x,y)} \tag{2}$$

with $G_x(x,y) = \frac{\partial I}{\partial x}$ and $G_y(x,y) = \frac{\partial I}{\partial y}$.

In the **second step**, the histogram of gradient directions is calculated by accumulating the gradient magnitude $G(x, y)$ on a set of *bins*, where each bin covers a predefined segment of the full $0 \ldots 180^o$ orientation range. Accumulation is carried out over local regions called *cells* (see Fig. 3).

In the **third step**, a local normalization of the histograms is performed for improving the descriptor invariance to illumination and contrast changes. It is here carried out by grouping cells into larger entities, called *blocks*, each block grouping 2×2 adjacent cells, and by normalizing each block separately.

For each position of the detection window, the final descriptor is built by concatenating the normalized histograms obtained on each block within the corresponding window[1].

It must be noticed that, with this approach, detection of objects in the image is carried out by shifting the detection window over the entire image and, for each position of the window, passing the descriptor computed on this window to the classifier. This so-called *sliding* detection technique has often been considered unpractical in a real-time execution context due to its heavy computing requirements, especially with high resolution images. In the sequel, we will show how its reformulation under the *dataflow* model of computation actually allows it to be implemented so that it can be computed on the fly, independently of the respective sizes of the input images and detection windows.

B. Classification

The classification stage (second module in Fig. 2) uses the descriptor produced by the previous stage to tell whether an object is present in the corresponding detection window. For this, a SVM-based (Support Vector Machine) algorithm is used. This algorithm compares the input descriptor with a reference model, produced by a supervised learning (training) step, as proposed by Vapnik et al. [13]. The training step – which is carried out offline – builds the reference model by assigning a great number of pre-defined examples

[1]The size of the detection window is supposed to be a multiple of the size of the block.

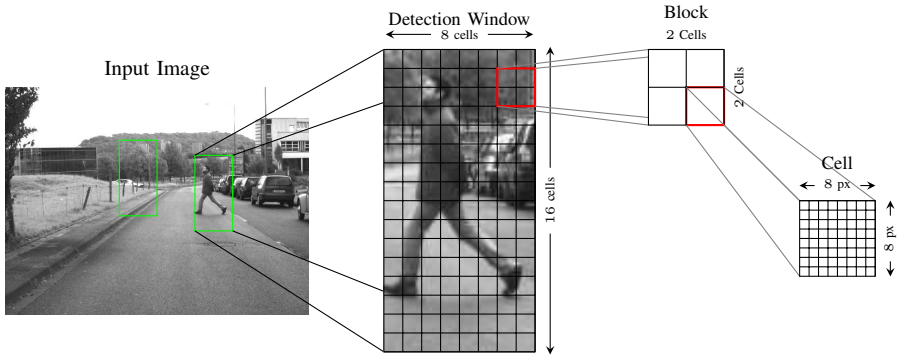


Fig. 3: Illustration of detection window, block and cell

to a predefined set of *classes* (two here). The reference model is then used online to map each descriptor to one of the predefined classes (non-probabilistic binary classification).

More formally, given a set of l data elements $\mathbf{x_1}, ..., \mathbf{x_l}$ and their corresponding classes $y_1 = y(\mathbf{x_1}), ..., y_l = (\mathbf{x_l})$ where $\mathbf{x_i} \in \mathbb{R}^n$ and $y_i = \pm 1$, the classification function can be expressed as follows:

$$y(\mathbf{x}) = sgn\left(\sum_{i=1}^{N_{SV}} y_i \alpha_i K(\mathbf{x_i}, \mathbf{x}) + b\right), 0 \leq \alpha_i \leq C \tag{3}$$

where $K(\mathbf{x_i}, \mathbf{x})$ is a kernel function, C is a regularization constant, α_i and b are parameters given by the learning phase. N_{SV} is the number of the reference features, called Support Vectors (SVs). In our case $\mathbf{x_1}, ..., \mathbf{x_l}$ are database images and y correspond to the presence or not of the targeted objects in images. When linear kernels are used for binary classification, the classification function can be expressed as a simple dot product [14] :

$$y(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} + b \tag{4}$$

The weight vector $\mathbf{w}$ and the bias b are determined by the training phase for each object we want to detect.

C. Algorithm

The algorithm for the object detection application, detailing feature extraction and the classification step introduced in the previous section, is given in listing 1, in pseudo code.

The first step (labeled HOG) corresponds to the feature extraction described in Sec. II-A. It computes the HOG descriptor for each cell in the source image. The magnitude and the orientation angle of the gradient are computed for every pixel using the pixel-wise operations `gradx`, `grady`, *square root* and `arctan`. Then, the angle is discretized over eight uniformly spaced bins[2]. In listing 1, this discretization is performed by the function `findbin`. The function `getcell` returns the cell to which the current pixel belongs. Finally, for each cell, the descriptor is computed by first grouping cells in blocks and normalizing each histogram on the grouped cells using statistics computed on the block (functions `createblock` and `norm`).

The second step (labeled SVM) corresponds to the classification described in Sec. II-B. For each position of the detection window, the descriptors computed on the enclosed cells are concatenated (function `gatherHOG`) to build a global descriptor and this descriptor is passed to the classification function `svm`, which computes the dot product described by equation 4 and outputs the classification result for this window (object present or not).

[2]Since spatial information are often unrelated to vertical orientations, only the upper semi-circumference is considered regardless of the full turn [12]

```
// HOG
for all pixel (i,j) in image I
    gradx(i,j) = I(i,j+1) - I(i,j-1)
    grady(i,j) = I(i+1,j) - I(i-1,j)
    magnitude = sqrt(gradx^2 + grady^2)
    angle = arctan(grady(i,j),gradx(i,j))

    // find the histogram bin for this orientation
    bin = findbin(angle)

    // find the cell to which the current pixel belongs
    cell = getcell(i,j)
    histogram (cell, bin) =   histogram(cell, bin) + magnitude
end

for all cells c in image I
    // gather cells into block
    descriptor(c) = createblock(histogram)
    descriptor(c) = norm(descriptor(c))
end

// SVM
for all detection windows w in image I
    finaldesc = gatherHOG(descriptor, w)
    y = svm(finaldesc, model)
end
```

Listing 1: Algorithm

III. DATAFLOW REFORMULATION

In this section, we will reformulate the algorithm described in Sec. II according to a dataflow model of computation (MoC). This reformulation will be the key for its efficient implementation on a FPGA, as demonstrated in Sec. IV.

The dataflow MoC we use is the one initially introduced by Dennis in [15] : applications are described as a collection of computing units (called *actors*) exchanging streams of tokens through unidirectional FIFO-like channels. Execution occurs as tokens litteraly flow through channels, into and out of actors, according to a set of firing rules. These firing rules specify that an actor becomes active whenever tokens are available on all of its input channels and token(s) can be written on its output channel(s). When this occurs, input tokens are consumed, and result(s) are computed and produced on the output channel(s).

We extend the original model with a mechanism allowing *structured* values, such as images, windows or histograms, to be represented as sequential streams of tokens. With this mechanism, the different operations involved in a given algorithm can be described as actors operating "on the fly" on the corresponding streams. For this, we will actually distinguish two categories of tokens : *data* tokens and *control* tokens. Data tokens will be carrying *values* (such as pixels, or histogram values). Control tokens will be used to *structure* the streams.

This is illustrated in Fig. 4 with an actor computing the horizontal derivative of images. For each pixel $P(i,j)$ of the input image, the corresponding pixel in the output image is either $P(i,j) - P(i,j-1)$ (for $j > 0$) or $P(i,j)$ (for $j = 0$). Each image is represented as a list of lists, the control token "<" (resp. ">") denoting the start (resp. end) of a list[3]. For legibility, the size of the images is 3×3 in Fig. 4. The behavior of the `DX` actor is purely data driven : whenever a token is available on its input, it is consumed and a corresponding result token is produced on the input. Informally, this behavior can be specified as follows[4] :

- when waiting for a new image and reading a "<" control token, write the same token on output and start waiting for a new line,
- when waiting for a new line and reading a "<" control token, write the same token on output, set local variable `z` to 0 and start waiting for pixels,
- when waiting for pixels and reading a data token `p`, write data token `p-z` and set variable `z` to `p`,

[3]In Fig. 4, the input of output sequences of tokens should be read from left to right; in other words, the input (resp. output) tokens are fed (resp. produced) to (resp. by) the actor in this order : first "<", then "<", then `0`, then `1`, *etc.* (resp. "<", "<", `0`, `1`, *etc.*).

[4]Formally, and as described in the next section, the CAPH language encodes this behavior as a set of *transition rules*, where each rule is described as a combination of pattern matching on the input(s) and local variable(s) and of actions on the output(s).

- when waiting for pixel and reading a ">" control token, write the same token on output and start waiting for a new line,
- when waiting for a new line and reading a ">" control token, write the same token on output and start waiting for a new image.

Note that this style of description naturally supports a pipelined execution scheme; processing of a line, for example, can start as soon as the first pixel is read without having to wait for the entire structure to be received; this feature, which effectively allows concurrent circulation of successive "waves" of tokens through the network of actors is of course crucial for on-the-fly processing.

Fig. 4: Dataflow processing example

In the rest of this section, we propose a complete reformulation of the algorithm described in Sec. II according the dataflow model described above, starting with a dataflow graph showing all involved actors and data dependencies between these actors and continuing with a description of the behaviour of the most significant actors.

The complete dataflow graph of the algorithm listed in listing 1, reformulated according to our dataflow model, is given in Fig. 5. In this figure, each gray box correspond to an actor and channels are drawn as black arrows connecting theses boxes. Images enter the graph as structured streams of pixels and results are output as binary images (also represented as structured stream of pixels) indicating whether object has been detected at each position of the source image.

All definitions and notations in the sequel refer to those used in Fig. 5.

A. Feature extraction

This part corresponds to the upper part of the graph (outlined in a red box in Fig. 5).

1 Gradient Computation

Gradient computation is a classical image processing operation. However, when targeting FPGAs, the main challenge is the implementation of the square root and $arctan$ functions. The square root is often approximated as the sum of the absolute values of the gradient components but the computation of the $arctan$ function is more tricky. Classical solutions based on look-up tables (LUTs), such as described in [16] for example, either requires large amounts of memory or multi-cycle operations which significantly reduces data throughput. Several approaches, such as [17], [18], have proposed to rely on the binning strategy to accumulate the histogram without explicitly computing $arctan(G_y/G_x)$ but their sequential formulation does not fit well with the dataflow model of computation.

We propose to use a set of eight 3×3 convolution kernels $C_{0...7}$ not to directly bin the gradient orientation but instead to calculate the contribution of each corresponding direction to each histogram bin. With this approach, the components of the final histogram can be computed in parallel.

For this, a set of eight actors $conv$, each parametrised by a 3×3 kernel C_k, each accepts an image p and produces another image q in which non-null pixels are those which have to be accumulated in the k^{th} bin of the target histogram.

More formally, given an $n \times m$ image p, encoded as a structured streams of pixel

$$p = << p_{1,1} \dots p_{1,m} > \dots < p_{n,1} \dots p_{n,m} >> \tag{5}$$

the behavior of actor $conv$, parametrised by kernel C_k, can defined in a purely functional manner with the following equation :

$$conv_{C_k}(<< p_{1,1} \dots p_{1,m} > \dots < p_{n,1} \dots p_{n,m} >>) \; = << q_{1,1} \dots q_{1,m} > \dots < q_{n,1} \dots q_{n,m} >>$$

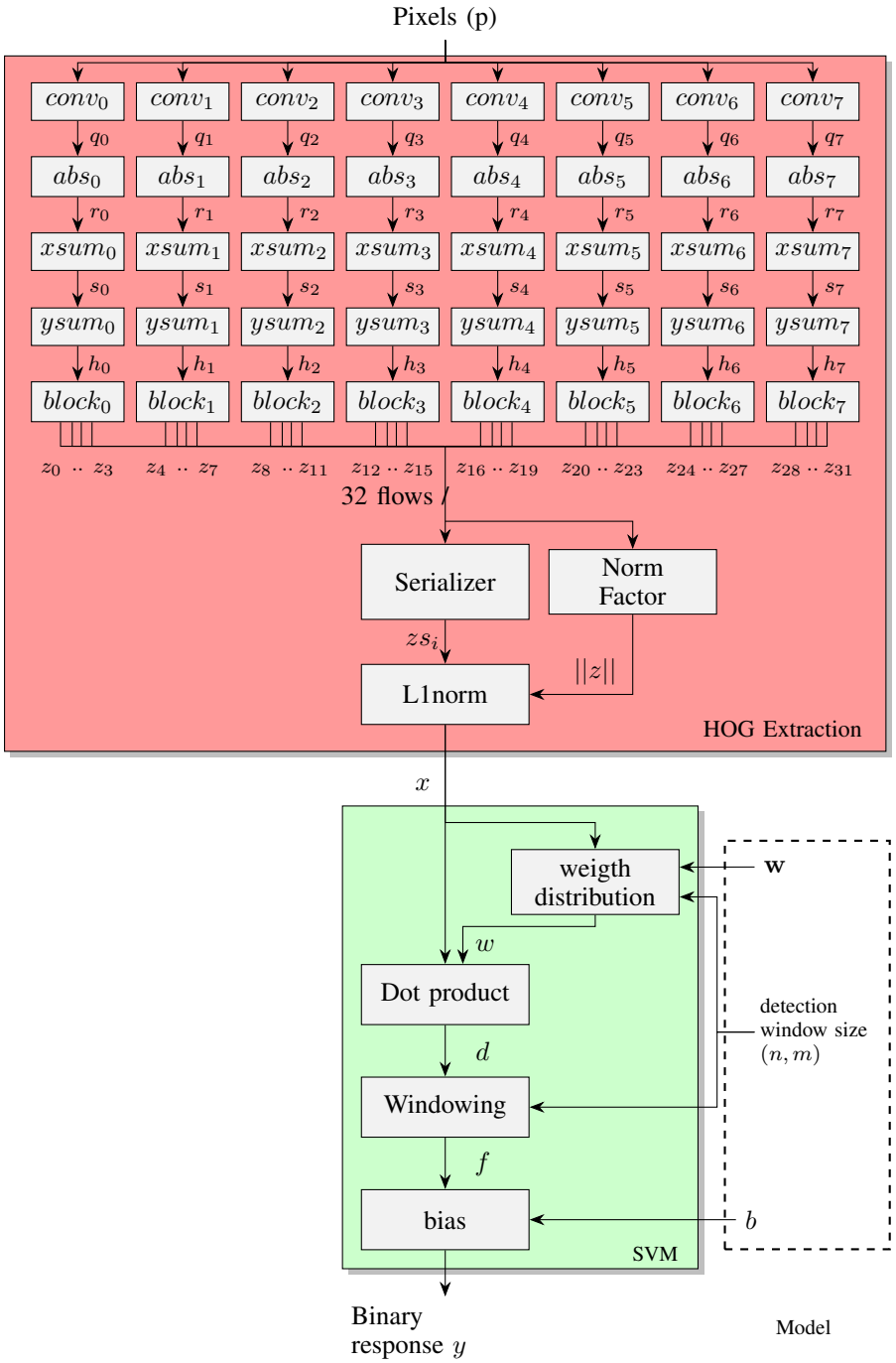


Fig. 5: Dataflow graph of the object detection application

where

$$q_{i,j} = \begin{cases} 0, & \text{if } i \in \{1, n\} \vee j \in \{1, m\} \\ \cos\theta_k \, p_{i,j+1} - \cos\theta_k \, p_{i,j-1} + \sin\theta_k \, p_{i+1,j} - \sin\theta_k \, p_{i-1,j}, & \text{otherwise} \end{cases}$$

and

$$\theta_k = \frac{k\pi}{8},\ k \in \{0, \ldots, 7\}$$

2 Histogram Computation

Histogram computation is the major processing bottleneck of all HOG-based detection applications and this is the reason why HOG descriptors has rarely been used (even when improved by Zhu et al. [19]). In [20], Porikli proposed to use so-called *integral histograms* to speed-up histogram computation. The technique allows fast evaluation of histogram values within rectangular regions regardless of their size. However, in our case, it would require to first generate an integral histogram for each histogram bin, and then compute the histograms of the target regions by intersection, which is not easily formulated under the dataflow MoC.

We propose, rather, to compute the k^{th} histogram entry by simply accumulating gradient magnitudes over each cell using two successive actors : one summing values in in the x direction and the other in the y direction. These actors are named `xsum` and `ysum` in Fig. 5. On each bin channel k, they compute an histogram image h_k defined as

$$\forall (ii, jj), \quad h_k(ii, jj) = \sum_{j=(jj-1)\cdot d+1}^{jj\cdot d} \sum_{i=(ii-1)\cdot c+1}^{ii\cdot c} |q_k(i, j)| \tag{6}$$

where q_k is the k^{th} bin image produced by corresponding `conv` actor and (ii, jj) corresponds to the cell coordinates, given by $ii = \{1, 2, ..., \frac{n}{c}\}$ and $jj = \{1, 2, ..., \frac{m}{d}\}$ (where $n \times m$ is the input image size).

In other words, the histogram component corresponding to the k^{th} bin is given by

$$h_k = ysum(xsum(abs(q_k))) \tag{7}$$

where the three actors `abs`, `xsum` and `ysum` can be defined as follows.

The `abs` actor simply computes the absolute values of its input image :

$$abs(<< q_{1,1} \ldots q_{1,m} > \ldots < q_{n,1} \ldots q_{n,m} >>) \ = << r_{1,1} \ldots r_{1,m} > \ldots < r_{n,1} \ldots r_{n,m} >> \tag{8}$$

where

$$r_{i,j} = |q_{i,j}|$$

The `xsum` actor accumulates pixels in the x (row) direction. On each line of the input image, it reads c consecutive values and writes the sum of these values. Formally :

$$xsum(<< \underbrace{r_{1,1}, \ldots, r_{1,m}}_{m \text{ elements}} > \ldots < r_{n,1}, \ldots, r_{n,m} >>) = << \underbrace{x_{1,1}, \ldots, x_{1,s}}_{s = m/d \text{ elements}} > \ldots < x_{n,1}, \ldots, x_{n,s} >> \tag{9}$$

where

$$x_{i,jj} = \sum_{j=(jj-1)\cdot d+1}^{jj\cdot d} r_{ij}$$

The `ysum` operates as `xsum` but in the y direction. On each column of the input image, it reads d consecutive values and writes the sum of these values. Formally :

$$ysum(< \underbrace{< x_{1,1}, \ldots, x_{1,s} > \ldots < x_{n,1}, \ldots, x_{n,s} >}_{n \text{ lines}} >) = < \underbrace{< h_{1,1}, \ldots, h_{1,s} > \ldots < h_{p,1}, \ldots, h_{p,s} >}_{p = n/c \text{ lines}} > \tag{10}$$

where

$$h_{ii,j} = \sum_{i=(ii-1)\cdot c+1}^{ii\cdot c} x_{ij}$$

The composition of actors `xsum` and `ysum`, on a $n \times m$ input image produces a $n/c \times m/d$ output image. In our case, $c \times d$ is the cell size ($c = d = 8$).

B. Normalization

As illustrated in Fig. 3, the HOG descriptor of a given cell is obtained by gathering the histograms computed on a 2×2 neighbourhood. This means that each descriptor will have 32 components (eight bins $\times$ four neighboors).

Extraction of the 2×2 neighbourhood extraction is performed on each bin in parallel by the `block` actor. This actor takes one stream hk (representing the input image) and produces four streams z_{4k}, z_{4k+1}, z_{4k+2}, z_{4k+3}. If the input stream describes an image with p lines of s pixels, each output stream will describe one neighboring image (with $p-1$ lines of $s-1$ pixels). I.e. :

$$block(hk) = (z_{4k}, z_{4k+1}, z_{4k+2}, z_{4k+3}) \tag{11}$$

where

$$\begin{aligned}
z_{4k} &=<< hk_{1,1}, \ldots, hk_{1,s-1} > \ldots < hk_{p-1,1}, \ldots, hk_{p-1,s-1} >> \\
z_{4k+1} &=<< hk_{1,2}, \ldots, hk_{1,s} > \ldots < hk_{p-1,2}, \ldots, hk_{p-1,s} >> \\
z_{4k+2} &=<< hk_{2,1}, \ldots, hk_{2,s-1} > \ldots < hk_{p,1}, \ldots, hk_{p,s-1} >> \\
z_{4k+3} &=<< hk_{2,2}, \ldots, hk_{2,s} > \ldots < hk_{p,2}, \ldots, hk_{p,s} >>
\end{aligned}$$

In the previous definition, if we consider the first element of each stream z_i, the four tokens represent the neighbourhood of the current histogram location ($hk_{1,1}, hk_{1,2}, hk_{2,1}, hk_{2,2}$). Applied to each bin image, the `block` actor then generates 32 parallel streams, each stream carrying each one a descriptor component $(z_0, ..., z_{31})$. Each z_i stream is a HOG component stream of x, y tokens where respectively $x = n/d-1$ and $y = m/c-1$ (n,m in the image input size and c,d is the histogram cell size). A HOG component stream z_i is represented as:

$$z_i =<< z_{1,1}, \ldots, z_{1,y} > \ldots < z_{x,1}, \ldots, z_{x,y} >> \tag{12}$$

For each cell, the normalization of the descriptor $(z_0, ..., z_{31})$ is operated by two actors : the `norm_factor` actor computes the normalization factor $||z||$ for each cell (as defined in Eq. 13) and the `L1norm` actors performs the normalization itself (as defined in Eq. 15).

The `norm_factor` actor can be defined as follows:

$$norm\_factor(z_0, ..., z_{31}) = ||z|| \tag{13}$$

where

$$||z|| =<< z_{1,1}, ..., z_{1,y} > ... < z_{x,1}, ..., z_{x,y} >>$$

and

$$z_{i,j} = \sum_{i=0}^{31} zi_{i,j}$$

Before computing the normalization of each descriptor component, we first concatenate the 32 streams $(z_0, ..., z_{31})$ into a single stream zs (in figure 5). This serialisation reduces the number of `L1norm` actors from 32 to 1 limiting the hardware resources in implementation. The serialiser produces a $s \times t$ image stream where $s = x$ and $t = 32 \times y$ and can be defined as:

$$serialiser(\underbrace{z_0, ..., z_{31}}_{\text{32 streams}}) = \underbrace{zs}_{\text{1 stream}} \tag{14}$$

where

$$zs =<< zs_{1,1}, ..., zs_{1,t} > ... < zs_{s,1}, ..., zs_{s,t} >>$$

The full block descriptor normalization is performed by the `L1norm` actor which divides the serialized HOG descriptor stream zs by the norm factor (for each cell) . The `L1norm` actor can be defined by:

$$L1norm(<< zs_{1,1}, ..., zs_{1,t} > ... < zs_{s,1}, ..., zs_{s,t} >>, << z_{1,1}, ..., z_{1,y} > ... < z_{x,1}, ..., z_{x,y} >>) = x \tag{15}$$

where

$$x =<< x_{1,1}, ..., x_{1,t} > ... < x_{s,1}, ..., x_{s,t} >>$$

and

$$x_{i,j} = zs_{i,j}/z_{i,jj}, \quad jj = j/32$$

We now define the higher-order function $norm$ corresponding to the normalization of HOG. For the tuple $(h_0, ..., h_7)$ coming from the histogram function, the $norm$ function produce the HOG stream (x).

$$norm(h_0, ..., h_7) = L1\ norm\ (\ serialiser\ (z_0, ..., z_{31}),\ normfactor\ (z_0, ..., z_{31})) \tag{16}$$

where the tuple $(z_0, ..., z_{31})$ is obtained by applying the `block` actor to each histogram stream $(h_0, ..., h_7)$.

Now we have defined all actors or functions for the HOG extraction, we can simply express the relation between the input stream p with the output descriptor stream x. This function is a composition of the previous ones:

$$x = hog(p) = \ norm(hists(convs(p))) \tag{17}$$

where $hists$ and $convs$ are respectively the $hist$ function and $conv$ actor applied to all parallel bins.

C. Classification

To evaluate the SVM results over a detection window, we need to compute the dot product of all histogram blocks belonging to it using a pre-trained model. According to the linearity of the equation 4, we can compute partial dot products to obtain the final result. The corresponding computation is described using four actors, all shown in the lower part of Fig. 5 : `weight_distribution`, `dot_product`, `windowing` and `bias`. The SVM-based classification step can be defined by the following equation :

$$svm_{\mathbf{w},n,m,b}(x) = bias\ (windowing\ (dot\ (x, weight\ distribution\ (x, \mathbf{w}, n, m)), n, m)\, b) \tag{18}$$

where x is the input stream of HOG descriptors, $\mathbf{w}$ the weight vector obtained by the external training phase, n and m the sizes of the detection window size and b the bias.

The `weight_distribution` actor generated the partial weight vector w corresponding to the partial descriptor x. More precisely, given the structured stream x, it produces a synchronised stream of weights (from an external memory) as function of the pre-trained model and the detection window size (n, m). Considering the serialized input descriptor flow x (output of the hog extraction part) of a size s, t, the actor is then expressed as follows:

$$weight\ distribution(x, \mathbf{w}, n, m) =<< w_{1,1}, ..., w_{1,t} >, ..., < w_{s,1}, ..., w_{s,t} >> \tag{19}$$

Once the weight w has been generated, the `dot product` actor computes the projection of w over the current x as in equation 20.

$$dot(x, w) =<< d_{1,1}, ..., d_{1,t} >, ..., < d_{s,1}, ..., d_{s,t} >> \tag{20}$$

where

$$d_{i,j} = x_{i,j} \cdot w_{i,j}$$

The projection values are then processed by the `windowing` actor. It accumulates the results according to the window which the cell is belonging. The windowing actor is a composition of the two previous actors

`xsum` and `ysum` presented in the section III-A2 with a configurable detection window size n, m (sketched on figure 5).

$$windowing(<< d_{1,1}, ..., d_{1,t} >, ..., < d_{s,1}, ..., d_{s,t} >>, n, m) =<< f_{1,1}, \ldots, f_{1,l} > \ldots < f_{k,1}, \ldots, f_{k,l} >> \tag{21}$$

where

$$f_{ii,jj} = \sum_{j=(jj-1)\cdot m+1}^{jj\cdot m} \quad \sum_{i=(ii-1)\cdot n+1}^{ii\cdot n} d_{ij}$$

Finally, the actor `bias` adds the b parameter as expected by the SVM model. The detection result y is then expressed in equation 22.

$$bias(<< f_{1,1}, \ldots, f_{1,l} > \ldots < f_{k,1}, \ldots, f_{k,l} >>, b) =<< y_{1,1}, ..., y_{1,l} >, ..., < y_{k,1}, ..., y_{k,l} >> \tag{22}$$

where

$$y_{i,j} = \begin{cases} 0 & \text{if } f_{i,j} < b \\ 1 & \text{otherwise} \end{cases}$$

This completes the dataflow reformulation of the object detection algorithm. In the next section, this formulation will be turned into a concrete program to be implemented on a FPGA platform.

IV. Implementation

In this section, we describe a concrete implementation of the object detection application inspired from the dataflow formulation proposed in previous section. It is carried out using the CAPH dataflow programming language. A brief presentation of this language is first given in Sec. IV-A. Then the actual encoding of the application in CAPH is described in Sec.IV-B.

A. The CAPH language

CAPH [9]–[11] is a high-level, domain specific language (DSL) based upon the dataflow model of computation. Its main goal is to provide a fully-automated compilation path from high-level dataflow descriptions of algorithms to their implementations on FPGA-based platforms, with a specific focus on *stream processing* applications. For such applications, the CAPH toolchain provides an effective rapid prototyping environment for FPGA programming, producing ready-to-synthetize RT-level VHDL code.

The CAPH language embodies the dataflow model by providing two formalisms : one for describing the *behavior* of actors operating on structured streams of values and another for describing the *structure* of dataflow graphs built from such actors.

Following the principles introduced in Sec. III, the behavior of actors in CAPH is expressed as a set of *transition rules*. Each rule describes what happens (in terms of computation and/or I/O) whenever a new token (or set of tokens) is available on the input(s). It consists of a set of *patterns*, involving inputs and/or local variables and a set of *expressions*, describing modifications of outputs and/or local variables.

The actor sub-language of CAPH has already been described in previous papers [9]–[11] ([10], in particular, describes a preliminary CAPH implementation of the HOG algorithm) and will not discussed further in this paper.

Description of dataflow graphs (DFGs) in CAPH is carried out using a textual *network description language* (NDL). This NDL is a small, higher-order, purely functional language in which DFGs are described by defining and applying *wiring functions*. A wiring function is a function accepting and returning *wires* (graph edges). This concept is illustrated in Fig. 6, where the DFG on the left is described by the CAPH program on the right. Here, two wiring functions are defined : `neigh13` and `neigh33`. The former takes a wire and produces a bundle of three wires representing the 1×3 neighborhood of the input stream, by applying twice the built-in actor `dp` (one-pixel delay). The latter takes a wire and produces a bundle of nine wires

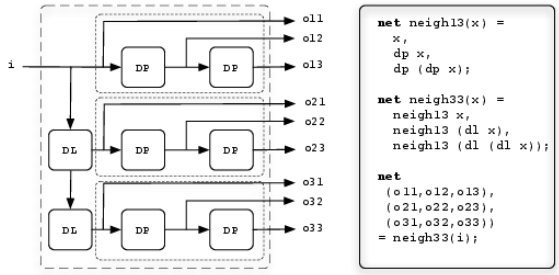


Fig. 6: Graph description with wiring functions in CAPH

representing the 3×3 neighborhood of the input stream, by applying the previously defined `neigh13` function and the built-in `dl` actor (one-line delay)[5].

The main originality of the CAPH NDL is to allow the definition of graph *patterns* as polymorphic, higher-order wiring functions, which greatly eases the description of large and complex graphs. This aspect has proven to be of significant importance for our object detection application, as will be evidenced in the next section.

B. The object detection application in CAPH

In Sec. III, the CAPH dataflow formulation of the object detection application is given. Since a prototype version of this implementation has already been described in [10], we will focus here on the *higher-order* capabilities of the language. These features have not been described hitherto and we will try to demonstrate how they greatly ease the description of the application.

The toplevel description of the application is shown in listing 2. The corresponding dataflow graph (generated by the compiler) is shown in Fig. 7. Input and output are introduced by the `stream` keyword. They respectively correspond here to the sequence of 8-bit images produced by the camera (each image beeing encoded as a structured stream of pixels) and to the sequence of binary images showing detection results (each image being also encoded as a structured stream of pixels). Parameters – the complete set of weights w for the classification step, the dimensions n and w of the detection window and the bias factor b – are communicated by means of asynchronous ports[6].

Distinction between data and control tokens is achieved by using a so-called *variant* type (`dc`) for input and output. The type `dc` is a polymorphic algebraic data type which can defined as

```
type t dc = SoS | EoS | Data of t
```

where `SoS` (Start of Structure), `EoS` (End of Structure) and `Data` are *value constructors* encoding respectively the "<" and ">" control tokens and data tokens. The t parameter denotes a *type variable*[7].

```
1  net convs rep x =
2    let ff i =  conv_33 (coeff[i]) in
3    mapi ff (rep x);
4
5  net hists xs ys hx =
6    let hist xs ys i = ysum ys (xsum xs (abs i)) in
7    map (hist xs ys hx);
8
9  net hog xs ys i =  norm (hists xs ys (convs rep8 i));
10
```

[5]The (sub)graphs resulting from the application of the `neigh13` and `neigh33` wiring functions are delineated using boxes drawn with short and long dash, respectively, on the figure.

[6]Asynchronous ports provide a means to modify application parameters without having to reload the whole application on the target FPGA.

[7]The type system of CAPH is similar to thoses equiping modern functional programming languages such as Haskell or ML. It supports parametric polymorphism and higher-order functions, in particular.

```
11 net norm (h0,...,h7) =
12   let (z0,...,z31) = map (block_extraction)  (h0,...,h7) in
13   let z = norm_factor(z0,...,z31) in
14   let zi =  serializer(z0,...,z31) in
15   l1_norm(zi, z);
16
17 net svm xi w n m b =
18   decision (b , (windowing n m (dot_prod (desc, weight_distr (xi,w,n,m)))));
19
20 net y = svm w n m b (hog 8 8 i);
21
22 stream i: unsigned<8> dc from "/dev/cam0";
23 stream y: unsigned<1> dc to "/dev/disp1";
24
25 port w : unsigned<8> from  "/dev/port0" init 0;
26 port n: unsigned<8> from "/dev/port1" init 0;
27 port m: unsigned<8> from "/dev/port2" init 0;
28 port b: unsigned<8> from "/dev/port3" init 0;
```

Listing 2: Toplevel description of the HOGSVM application in CAPH

The algorithm itself is encoded using several *wiring functions*[8] : `convs`, `hists`, `hog`, `norm` and `svm`. Each of these functions corresponds to a step of the algorithm described in the previous sections (in particular, the `hog` and `svm` functions directly correspond to the feature extraction and classification steps). For legibility, the code of the actors (`conv_33`, `xsum`, `ysum`, `abs`, `block_extraction`, `norm_factor`, `serializer` and `l1_norm`) has not been reproduced here.

The `convs` function (lines 1-3) creates the eight parallel streams representing the images q_k by instanciating the convolution actor `conv_33` with eight distinct kernels (stored a 2D array constant `coeff`, not shown here). For this, it uses the `mapi` builtin higher-order wiring function, which can be defined as

$$\text{mapi } f\ (x_1, \ldots, x_n) = (f\ 0\ x_1, \ldots, f\ (n-1)\ x_n)$$

In our case, the tuple on which the `mapi` is applied is generated by the `rep` actor. This simply generates n multiple copies of its input stream ($n = 8$ here).

The `hists` function (lines 5-7) generates the images representing the eight histogram bins. The `abs` actor computes the absolute value of the input stream coming from the convolution stage. The `xsum` and `ysum` actors are accumulators where the number of iterations is specified by the xs and ys parameters respectively. Replication of the processing on the eight parallel streams corresponding the different bins is expressed using the `map` builtin higher-order wiring function, whose functional definition is :

$$\text{map } f\ (x_1, \ldots, x_n) = (f\ x_1, \ldots, f\ x_n)$$

The `map` function is here applied to the locally defined function `hist`, which expresses the computation of the histogram on a single bin. Its definition (line 6) is the direct translation of Eq. 7.

The definition of the `hog` function is a direct translation of Eq. 17 : it is a composition of the functions *convs*, *hists* and *norm*. The first two arguments (`xs` and `ys`) give the histogram cell size (set to 8 here when the `hog` function is called, line 20).

The `norm` function (lines 11-15) is also a direct translation of Eq. 16, combining actors `block_extraction`, `norm_factor`, `serializing` and `l1_norm`. The `map` higher-order function takes here as input the tuple $(h0, ..., h7)$ which corresponds to the outputs of the histogram step as H_0 to H_7 in figure 5.

The `svm` function (lines 17-18) implements the classification step. It operates on the serial descriptor produced by `hog` function. It is, again, a direct translation of Eq. 18.

Finally, the function describing the global application (line 20) is obtained by composing the `hog` and `svm` functions.

[8] In CAPH, as in most of functional programming languages, function application is denoted without parenthesis, so that fxy means (here) $f(x, y)$.

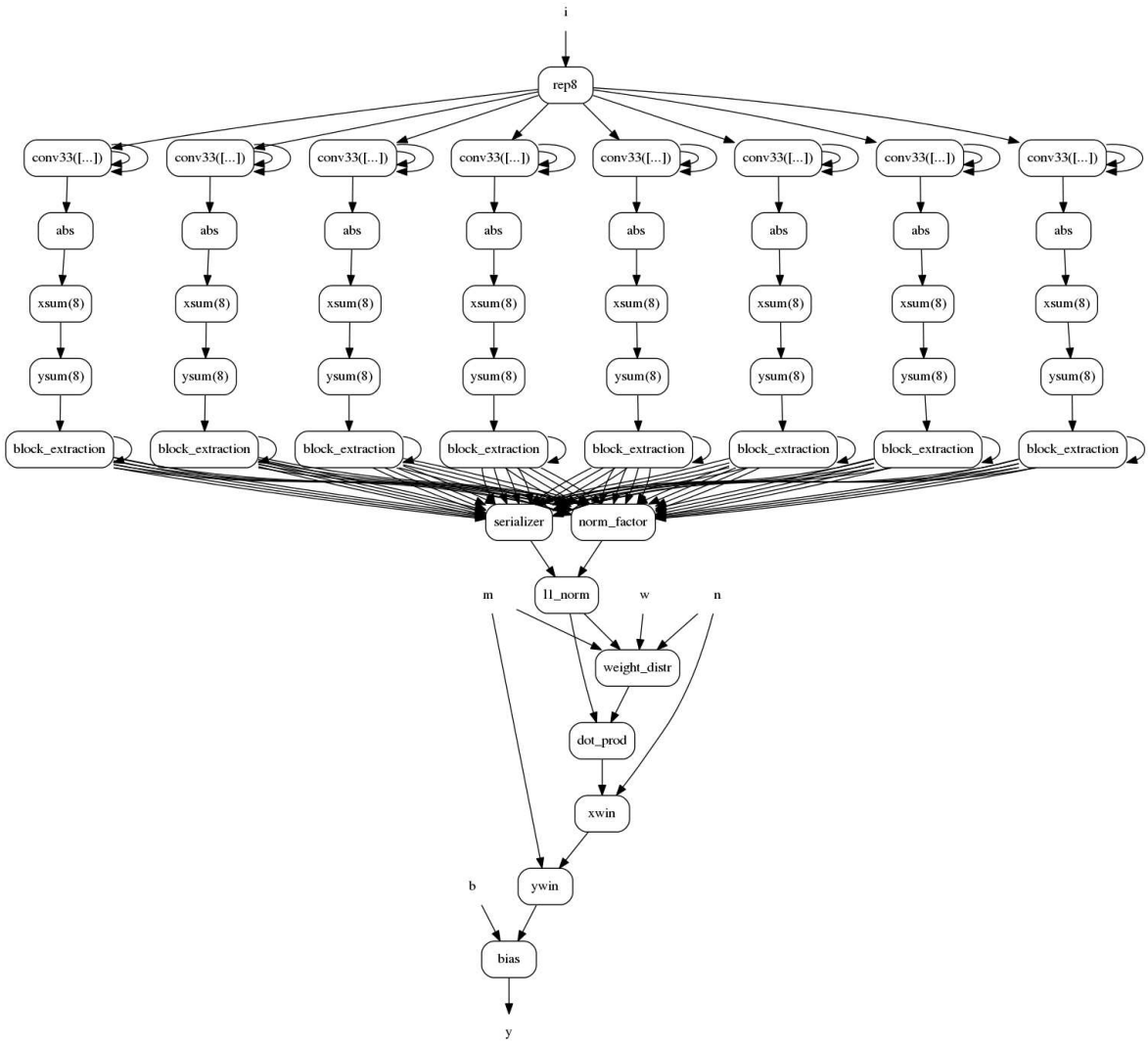


Fig. 7: Dataflow graph for the HOGSVM application from the code in listing 2

C. Platform

In this section, we present our FPGA-based smart camera platform, the DREAMCAM [21]. For the object detection application discussed here, the architecture of the DREAMCAM can be described as a combination of three modules as depicted in Fig. 8. The first module – which is actually application-independant – manages the external CMOS camera interfaces, generating the structured pixel streams for the following modules. The second module is the one which is actually generated by the CAPH compiler from the code described in the previous section. It implements the dataflow graph depicted in Fig. 7. The third module interprets results generated by the application for visualization (to produce images like illustrated in Fig. 9b). It is implemented using an embedded softcore microprocessor which actually acts as a system controller and manager. Moreover, since the pre-trained object model (w,n,m,b ports in listing 2) are inputs of the processing, the softcore has the ability to update these values, changing the object targeted in the camera stream.

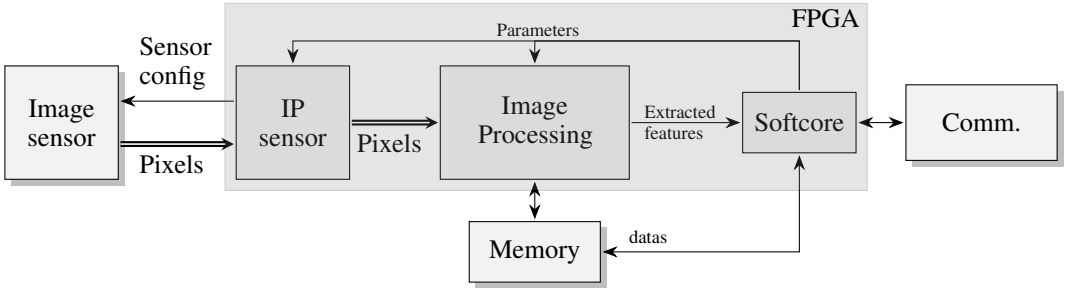
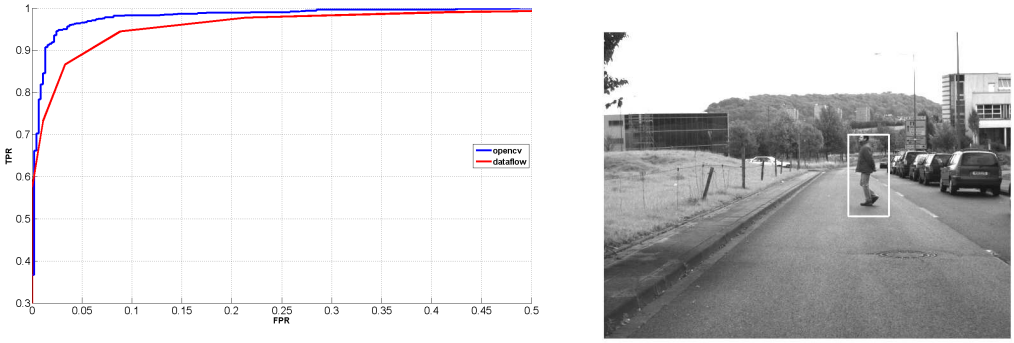


Fig. 8: Architecture of the object detection application as implemented on the DREAMCAM platform



(a) (b)

Fig. 9: (a) ROC metric on INRIA dataset. (b) Example of pedestrian detection on the Daimler dataset

V. RESULTS

In this section, we discuss the results obtained by deploying the VHDL code produced by the CAPH compiler on our FPGA-based smart camera. We first present the actual classification results, obtained on real image sequences, in order to assess the correctness of the implementation. Then the efficiency of the implementation is evaluated, by comparing some performance indicators with those obtained with other, state-of-the-art implementations.

1 Classification results

The proposed detection system has been firstly benchmarked to detect pedestrians on image. The classifier model has been obtained by a training phase with the INRIA dataset with the SVMlight tool [22]. In figure 9a the impact of the HOG dataflow reformulation over the classifier detection is shown. To evaluate the classifier performance, the test results are expressed with the Receiver Operating Characteristics (ROC) metric, where the True Positive Ratio (TPR) is plotted as function of the False Positive Ratio (FPS). For comparison, in figure 9a the OpenCV HOG implementation, based on Dalal algorithm [12], has been evaluated with respect to ours.

Even though our detection system is using fixed-point arithmetic – compared to the floating point OpenCV implementation – it is showing only a slight performance decrease (up to 3% at 3.3% FPR). Over the INRIA pedestrian dataset [12], our dataflow reformulation achieves 90% TPR and 3.3% FPR at 50 fps while the hardware reference implementation [18] provides 2% higher TPR (92% TPR at 3.3% FPR). Moreover, it delivers comparable results with those obtained with a full-fledged floating point implementation, while offering a path for FPGA friendly hardware implementations.

In figure 9b an example of pedestrian detection is shown. The original picture has been taken form the Daimler dataset [23] and processed with our dataflow system with the same training model as above.

2 Performance results

Deployment is performed using Quartus II 13.1 synthesis toolchain for synthetizing the VHDL code generated by the CAPH compiler. Two indicators are used to assess performances : resource usage and number of processed frames per second (FPS).

Table I gives resource usage on the Altera Cyclone III EP3C120 embedded on the DREAMCAM platform for a 1280×1024 image resolution, expressed as in terms of used logic elements (LEs), memory bits and embedded multipliers, for each step of the algorithm. When a given step involves several distinct actors (e.g., gradient) results are given for each involved actor.

The whole dataflow implementation requires 16392 LEs (around 17% of the target FPGA) and 70 kbits of embedded memory cells. Only one DSP module is used, for computing the dot product in the SVM prediction step[9].

A significant part of the logic elements are used to implement the FIFO channels used to exchange tokens between actors. The width and depth of these FIFOs is computed by the CAPH compiler . These FIFOs are inherent to our dataflow model of execution.

TABLE I: Hardware resource usage level for 1280×1024 image resolution

Operation	Actors	Logic elements (LE)	Memory (kbits)	DSP
Gradient	rep8	20	0	0
	8 × conv_33	1036	25.7	0
Histogram	8 × abs	238	0	0
	8 × xsum	744	0	0
	8 × ysum	981	20.8	0
Normalization	8 × block_extraction	2283	23.7	0
	serializer	829	0	0
	norm_factor	161	0	0
	L1norm	320	0	0
SVM prediction	weight_distr	51	0	0
	dot_prod	29	0	1
	xwin	240	0	0
	ywin	130	1	0
	decision	10	0	0
Channels Fifo		9320	0	0
Total		16392 LE (17%)	70 kbit (2%)	1/576 DSP

Despite the overhead induced by the FIFO channels, the hardware occupancy performance are comparable with state of the art systems.

In table II our implementation is compared, both in terms of resource usage and maximum FPS, with those reported by three other authors on several FPGA plaforms. Since LE count is highly device dependent, comparison of resource usage is carried out in terms of LUTs and registers. For this, the LE reported in table I are first split in two : those implementing LUTs and those implementing registers.

Our implementation, obtained without writing a single line of HDL code, offers performances which are comparable in terms of resource usage and FPS. Mizuno's [24] and Hahnle's [18] implementation use a hand-crafted HDL optimized pipeline to boost up performances. In particular, compared to Mizuno [24] we significantly reduce the LUTs and registers footprint without sacrifying throughput (fps × resolution). Compared to Hahnle [18], our implementation requires more logic resources but need less internal memory (70 kbits out of 936 kbits for the HOG extraction). Compared to Kadota [25], our implementation performs better in terms of FPS, but uses more LUTs and registers, due to the increased image resolution.

[9]For gradient computation, no DSP is used since all the required multiplications involve only constant factors.

TABLE II: State of the art comparison

reference	platform	resolution	fps	LUT	Register	Memory (kbits)	DSP	Mhz
Kadota [25]	Altera Stratix II	640 x 480	30	3794	6699	n/a	12	127
Mizuno [24]	Altera Cyclone IV	1920 x 1080	30	34 400	23247	348	68	76
Hahnle [18]	Xilinx Virtex-5	1920 x 1080	64	5188	5176	1 188	49	270
our	Altera Cyclone III	1280 x 1024	50	5062	10900	70	1	60

VI. Conclusion and Future Work

We have described the application of a real-time object detection application on a FPGA-based smart camera architecture.For this application, we have shown that an efficient implementation can be obtained with a dataflow based language whose abstraction level is significantly higher than that of traditional HDL languages such as VHDL or Verilog. Together with previous experimentations [10], [26], this confirms that, for applications having to operate on the fly on video data streams, the dataflow model of computation, used jointly as a programming model and an execution model can offer a very effective way to conciliate abstraction and efficiency when programming FPGAs. This in turn opens significant opportunities to exploit this kind of devices in architectures such as smart cameras, since, in the mid and long term, it is not realistic to require that programmers of these architectures rely on low-level hardware description languages.

We intend to continue working in several directions in order to support this claim. First, for the discussed application, by assessing several alternative formulations and generalisations. For example, using overlapping detection windows to improve detection reliability in complex scenes or merging multi-scale SVM outputs to go towards a neuro-inspired classification scheme. Second, in reformulating and implementing other applications. Third, in improving the CAPH compiler, by optimizing the code generated by VHDL backend in order to minimize resource usage or reduce critical paths.

Acknowledgment

This work has been sponsored by the French government research programme ”Investissements d’avenir” through the IMobS3 Laboratory of Excellence (ANR-10-LABX-16-01), by the European Union through the program Regional competitiveness and employment 2007-2013 (ERDF Auvergne region), and by the Auvergne region.

References

[1] M. Graphics and H. Tool, “Catapult c,” 2010.

[2] J. Frigo, M. Gokhale, and D. Lavenier, “Evaluation of the streams-c c-to-fpga compiler: an applications perspective,” in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. ACM, 2001, pp. 134–140.

[3] A. Antola, M. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi, “A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper,” in *Programmable Logic, 2007. SPL’07. 2007 3rd Southern Conference on*. IEEE, 2007, pp. 221–224.

[4] Y. Yankova, K. Bertels, S. Vassiliadis, R. Meeuws, and A. Virginia, “Automated hdl generation: Comparative evaluation,” in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, May 2007, pp. 2750–2753.

[5] J. Eker and J. Janneck, “Cal language report,” University of California at Berkeley California, Berkeley, CA 94720, USA, Technical Memorandum, 2003.

[6] W. A. Najjar, E. A. Lee, and G. R. Gao, “Advances in the dataflow computational model,” *Parallel Computing*, vol. 25, no. 1314, pp. 1907 – 1929, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819199000708

[7] J. Sérot, G. Quénot, and B. Zavidovique, “Functional programming on a dataflow architecture: Applications in real-time image processing,” *Machine Vision and Applications*, vol. 7, no. 1, pp. 44–56, 1993. [Online]. Available: http://dx.doi.org/10.1007/BF01212416

[8] J. Sérot, G. M. Quénot, and B. Zavidovique, “A visual dataflow programming environment for a real-time parallel vision machine,” *Journal of Visual Languages and Computing*, vol. 6, pp. 327–347, 1995.

[9] J. Sérot, F. Berry, and S. Ahmed, “Caph: A language for implementing stream-processing applications on fpgas,” in *Embedded Systems Design with FPGAs*, P. Athanas, D. Pnevmatikatos, and N. Sklavos, Eds. Springer New York, 2013, pp. 201–224. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-1362-2_9

[10] J. Sérot, F. Berry, and C. Bourrasset, "High-level dataflow programming for real-time image processing on smart cameras," *Journal of Real-Time Image Processing*, pp. 1–13, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11554-014-0462-6

[11] "The Caph Programming Language home page." [Online]. Available: http://caph.univ-bpclermont.fr

[12] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *International Conference on Computer Vision & Pattern Recognition*, C. Schmid, S. Soatto, and C. Tomasi, Eds., vol. 2, June 2005, pp. 886–893. [Online]. Available: http://lear.inrialpes.fr/pubs/2005/DT05

[13] V. Vapnik, *Estimation of Dependences Based on Empirical Data: Springer Series in Statistics (Springer Series in Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[14] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF00994018

[15] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, 1980.

[16] S. Lee, K. Min, and T. Suh, "Accelerating histograms of oriented gradients descriptor extraction for pedestrian recognition," *Computers and Electrical Engineering*, pp. 1043 – 1048, 2013.

[17] S. Bauer, S. Kohler, K. Doll, and U. Brunsmann, "Fpga-gpu architecture for kernel svm pedestrian detection," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, June 2010, pp. 61–68.

[18] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "Fpga-based real-time pedestrian detection on high-resolution images," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2013 IEEE Conference on*, June 2013, pp. 629–635.

[19] Q. Zhu, M.-C. Yeh, K.-T. Cheng, and S. Avidan, "Fast human detection using a cascade of histograms of oriented gradients," in *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, ser. CVPR '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 1491–1498. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2006.119

[20] F. Porikli, "Integral histogram: A fast way to extract histograms in cartesian spaces," in *in Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2005, pp. 829–836.

[21] M. Birem and F. Berry, "Dreamcam : A modular fpga-based smart camera architecture," in *Journal of System Architecture - Elsevier - To appear - 2014*, 2014.

[22] T. Joachims, "Svmlight: Support vector machine," vol. 19, no. 4, 1999.

[23] M. Enzweiler and D. M. Gavrila, "Monocular pedestrian detection: Survey and experiments," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 31, no. 12, pp. 2179–2195, 2009.

[24] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "Architectural study of hog feature extraction processor for real-time object detection," in *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, Oct 2012, pp. 197–202.

[25] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura, "Hardware architecture for hog feature extraction," in *Proceedings of the 2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, ser. IIH-MSP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1330–1333. [Online]. Available: http://dx.doi.org/10.1109/IIH-MSP.2009.216

[26] J. Sérot, F. Berry, and S. Ahmed, "Implementing stream-processing applications on fpgas: A dsl-based approach," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, 2011, pp. 130–137.