FPGA implementation of a real-time human detector

Hugo Torres
hugo.p.torres@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisbon University, Portugal

Abstract—With the development of new technologies that are increasingly aware of their surroundings, such as autonomous vehicles, human detection assumes an unprecedented relevance. Computer vision algorithms often apply identical operations over large sets of data (SIMD). Therefore, they are unsuitable for common processors (CPUs) where parallelism is limited. More suitable architectures, such as dedicated processors or general purpose graphics processing units (GPGPUs), have other drawbacks. Dedicated processors, which frequently do not reuse resources, lead to solutions with high area requirements. GPGPUs, due to their complexity, have high power requirements.

The CoreWorks heterogeneous platform, which combines reconfigurable hardware accelerators with an embedded processor, delivers competitive performance while requiring only a fraction of the resources.

In this thesis, a human detection algorithm, based on Histograms of Oriented Gradients, was implemented in this platform. This process involved the adaptation of the algorithm to fixed-point arithmetic, allowing its efficient execution in the embedded platform. The resultant algorithm was accelerated through a SideWorks co-processor responsible for the main stages of the detection, namely the calculation of the gradients, votes and histograms in addition to the normalization and classification. It was then implemented and tested on a FPGA, resulting in runtimes over seven times smaller than the equivalent algorithm executed on the embedded processor.

Index Terms—histograms of oriented gradients, human detection, reconfigurable hardware accelerator, embedded platform, heterogeneous platform

I. INTRODUCTION

HUMAN DETECTION is one of the most challenging topics in computer vision. Some difficulties of this task include changes in lighting conditions, pose, scale, viewpoint, background and occlusion. Although highly difficult, human detection has applications in many fields, such as autonomous vehicles, Advanced Driver Assistance Systems (ADAS), advanced Graphical User Interfaces (GUI), image and video content indexing, surveillance, robotics and domotics.

A. Human detection

Most human detection algorithms share the same structure, as depicted in figure 1. After any necessary pre-processing, the input image is segmented into smaller regions, referred to as Regions Of Interest (ROIs). In each ROI, the features are then extracted and coupled to form a feature vector. Finally, each feature vector is fed into a classifier, producing a decision.

1) ROI selection: When its known a priori that not all image regions contain useful information for the detection process, ROI selection follows one of two possible approaches: key points or parts. In the first case, stable points with relevant information in their neighborhood, called key points, define the region for feature extraction [1]. The alternative approach relies on part detectors to identify a persons components, such as head, torso and limbs, allowing the selection of the correspondent ROI [2]. Both these approaches lead to detectors highly dependent on the robustness, repeatability and precision of the ROI selection process.

However, when detection is equiprobable in all image regions, ROI selection is frequently based on a sliding detection window. The window defines the ROI where the feature vectors are computed. Once finished, the window is moved over the image [3] and the extraction process is repeated. In the end, all processed windows form a dense grid over the image. In these detectors, the relevant regions for detection are determined only in the classification stage.

2) Features: Features retain relevant and discriminative information about the class of interest, such as details about luminosity, color, contour or texture, while discarding superfluous details, like irrelevant changes in lighting or viewpoint. This is accomplished through local operations, such as sums or histograms, over intensities or gradients of the images. So, features combine information from multiple pixels, allowing small feature vectors.

Features previously used in human detection include haar-like features [4], edgelets [5], Local Binary Patterns (LBP) [6], Scale Invariant Feature Transform (SIFT) [7], Histograms of Oriented Gradients (HOG) [3], shapelets [8], shape-contexts [9] and co-variance features [10]. Some complex features combine simpler features [11], [12] to improve detection rates. However, recent approaches choose to develop entirely new features, through neural networks [13], [14], instead of relying on manually crafted features.

3) Classification: The classifier, based on machine learning techniques, produces a decision for each feature vector. The classification is enabled by previously training the classifier...
B. Previous works

To achieve real-time detection, human detection algorithms often use dedicated, highly parallel and heterogeneous systems instead of more traditional solutions based on a single CPU. Most systems follow one of two approaches:

- **Dedicated processors** are designed to accelerate sections or the full detection algorithms. These processors rely on extensive pipelines with little to none resource reuse, resulting in high area requirement systems, rendering them unusable in some applications. Some processors can operate independently [16]–[18], while others require a CPU and several dedicated processors, integrating them in a traditional hardware/software co-project approach [19], [20].

- **General Purpose Graphics Processing Units (GPGPUs)** can also be used to accelerate sections or the full detection algorithms [21]–[24]. GPGPUs are highly complex programmable systems with high power requirements. Although this approach effectively reuses the available resources, it is inadequate for some applications.

Finally, some systems combine both approaches [25], [26] with varying degrees of success. However, these approaches have even higher requirements, both in terms of power consumption and area.

The CoreWorks heterogeneous platform offers a distinct approach, allowing comparable performance to other architectures with a fraction of the power and area requirements. In these systems, the designer can select the desired tradeoff between resource utilization and performance.

The paper is organized as follows. Section II describes the CoreWorks heterogeneous platform. Section III details the HOG human detection algorithm. Section V describes its software implementation. Section VI presents the detection system based on hardware/software co-design. Section VII describes the designed co-processor. Section VIII compares the fixed-point algorithm to its original implementation and evaluates the developed system. Finally, Section IX concludes the paper.

II. COREWORKS PROCESSING ENGINE

CoreWorks Processing Engine (CWPE), illustrated in figure 2, is a heterogeneous embedded platform based in hardware/software co-processing. Each CWPE core includes a CPU (FireWorks), one or more hardware co-processors (SideWorks), a bootloader, an input/output module (Data I/O) and a CCS (Configuration, Control and Status) register file.

FireWorks is a 32-bit RISC processor, which mostly executes the non-critical sections of the algorithm, controls the SideWorks co-processors and manages the communication with external modules. The processor has some peripherals, such as a RS-232 module and multiple timers, mostly used during development. The SideWorks co-processor executes the critical sections of the algorithm, allowing its acceleration.

The bootloader loads the embedded program into memory for execution. The Data I/O module manages the input and output interfaces, writing the received data into memory and producing the required output signals. In addition, this module also buffers the input data, allowing the stream processing. The CCS register file facilitates the communication between FireWorks and the external modules.

A. SideWorks Reconfigurable Accelerator

SideWorks is a technology for developing runtime reconfigurable hardware co-processors. Each SideWorks instance can perform multiple complex operations, called datapaths, between its internal memories. All possible operations are defined in the development stage. During operation, the SideWorks co-processor can reconfigure itself, allowing the execution of all its datapaths.

Since SideWorks accelerators are co-processors, they rely on commands from the FireWorks in order to perform the desired purpose. The most relevant commands are the execution (RUN) and load (LOAD). After their reception, SideWorks stores them internally, decoupling its operation from the FireWorks. Each LOAD causes SideWorks to autonomously load input data from external memory to its input memories. The input data received through Direct Memory Access (DMA) is written in one of four input memories connected in a ping-pong buffering scheme, which allows the simultaneous loading and processing of input data. Each RUN reconfigures the accelerator to the specified datapath, allowing the execution of the desired operation. Furthermore, each RUN also allows the FireWorks to send arguments to the respective datapath, allowing its utilization in multiple scenarios.

In order to construct each datapath, the SideWorks produces its functional units (FUs) in a pipeline structure for the desired operation. Pipeline operation allows for higher throughputs and operating frequencies. Although all connections between FUs are established in runtime, they rely on multiplexers with all necessary and sufficient connections for all datapaths, which are defined in the development phase.

The SideWorks produced results are written to memory through DMA, allowing further processing by the FireWorks processor.
SideWorks technology main advantages include: (1) high operating frequency, guaranteed by the pipeline structure of the datapaths; (2) reduced area, allowed by the reutilization of FUs between different datapaths; (3) post-implementation reconfiguration, enabled by the software-based configuration of the datapaths; (4) reduced development time, facilitated by the SideWorks simulator, which produces clock-exact results; (5) simplified design, allowed by the development tools, which convert the high-level specification of the datapaths to an hardware description language.

III. HISTOGRAMS OF ORIENTED GRADIENTS

The employed human detection algorithm [3], shown in figure 3, is based on the implementation from the OpenCV [27] (Open Source Computer Vision) library, which is pre-trained for human detection. Essentially, the detection relies on Histograms of Oriented Gradients (HOG) extracted in a sliding window approach, classifying each resultant feature vector with a Support Vector Machine (SVM).

![Fig. 3. HOG single-scale human detection algorithm.](image)

Normally, the first stages of the algorithm are applied over the full image, while all the subsequent stages consider each detection window independently. In the considered sliding window approach, all neighboring windows are distanced of 8 pixels, in each direction (horizontal and vertical).

Each detection window is subdivided into disjoint regions, called cells (8 × 8 pixels), over which the cell histograms are computed. Overlapping groups of 4 neighboring cells form blocks, whose descriptors are produced by joining all the histograms of cells inside them. After normalization, block descriptors are referred to as HOG features. The gathering of all the 105 HOG features within each detection window forms a feature vector. Based on this vector, the SVM produces a score, describing its confidence in the presence (or not) of a person in the corresponding detection window.

A. Padding and gamma correction

The padding creates a frame of one pixel surrounding the image. Each frame pixel receives the value of its closest neighbor still belonging to the original image. The gamma correction is applied independently to each image channel, according to expression (1). $I_C$ represents the normalized channel from the image $I$ with only one channel (B, G or R). The indexes $(x, y)$ define the pixel position within the image, considering the origin in the upper left corner.

$$I_{C_{yx}} = \sqrt{I_{yx}}$$  \hspace{1cm} (1)

B. Gradients

The gradient vector in each pixel is computed through two 2-D convolutions with distinct derivative masks, according to expressions (2) and (3). Color images require six convolutions, since each channel (B, G and R) is considered individually.

$$G_X = I_C \ast [-1 \hspace{1cm} 0 \hspace{1cm} 1]$$  \hspace{1cm} (2)

$$G_Y = I_C \ast [0 \hspace{1cm} -1 \hspace{1cm} 1]$$  \hspace{1cm} (3)

The gradient vectors are then converted to polar coordinates (magnitude and angle) through expressions (4) and (5).

$$M_{yx} = \sqrt{G_{X_{yx}}^2 + G_{Y_{yx}}^2}$$  \hspace{1cm} (4)

$$\Theta_{yx} = \arctan(G_{Y_{yx}}, G_{X_{yx}})$$  \hspace{1cm} (5)

The function $\arctan(y, x)$ computes the principal angle, in the interval $[-180^\circ, 180^\circ]$, of the vector with cartesian coordinates $(x, y)$. In color images, both the magnitude and angle for each pixel result from the channel with greater magnitude on that pixel.

Finally, all angles are corrected in accordance with expression (6).

$$\Phi_{yx} = \Theta_{yx} \times 9/180 - 0.5$$  \hspace{1cm} (6)

C. Votes

Due to the trilinear interpolation, each pixel produces two distinct votes for the histograms, each one formed by a bin and weight pair. The bin defines the class of the histogram, while the weight defines the intensity of that particular vote. Both required bins, in the interval $[0, 8]$, are computed through expressions (7) and (8).

$$bin_1(x, y) = \begin{cases} \lfloor \Phi_{yx} \rfloor + 9 & \text{if } \Phi_{yx} < 0 \\ \Phi_{yx} & \text{if } \Phi_{yx} \geq 0 \end{cases}$$  \hspace{1cm} (7)

$$bin_0(x, y) = \begin{cases} 0 & \text{if } bin_1(x, y) = 8 \\ bin_1(x, y) + 1 & \text{c.c.} \end{cases}$$  \hspace{1cm} (8)

Pixels with negative angles are mapped to the histogram class with homologous direction, in the interval $[0^\circ, 180^\circ]$. Taking into consideration the expression (9), both required weights of each pixel are computed through expressions (10) and (11).

$$\text{frac}(z) = z - [z]$$  \hspace{1cm} (9)

$$vote_1(x, y) = M_{yx} \times (1 - \text{frac}(\Phi_{yx}))$$  \hspace{1cm} (10)

$$vote_0(x, y) = M_{yx} \times \text{frac}(\Phi_{yx})$$  \hspace{1cm} (11)
D. Histograms

Cell histograms are assembled by locally accumulating pixel votes over the respective accumulation zones. As a result of the trilinear interpolation, each accumulation zone extends beyond cell limits, incorporating neighboring pixels within the same block. Thus, each relative cell position within a block (upper-left, upper-right, lower-left or lower-right) defines a distinct accumulation zone, as shown in figure 4.

![Fig. 4. Accumulation zones within a block, considering the four cells.](image)

Furthermore, considering a pixel position within the accumulation zone, both its votes are multiplied by a distinct constant factor. These factors, illustrated in figure 5, depend both on the pixel position within the accumulation zone and on the cell position inside the block. The multiplied factors are a consequence of the gaussian mask applied over every block and of the resulting weights from the interpolation.

![Fig. 5. Multiplication factors applied over the accumulation zones. Lighter pixels represent values closer to one, while darker pixels depict values near zero. The factors are distinct according to the relative cell position within the block: (a) Upper-left; (b) Upper-right; (c) Lower-left and (d) Lower-right.](image)

Every cell histogram is the weighted accumulation of all votes within a cell accumulation zone. Each vote is weighted according to its weight and its position within the accumulation zone, contributing to one class of the histogram, defined by the bin.

All block descriptors are created by combining their cell histograms in accordance with expression (12).

\[ \mathbf{b} = (h_{UL}, h_{LL}, h_{UR}, h_{LR}) \]  

E. Normalization

Each block descriptor is independently normalized, forming a HOG feature. The used L2-Hys normalization requires two sequential stages.

The first stage, shown in expressions (13) and (14), sums all elements of the block descriptor, computing the first scale factor. Afterwards, the scale factor is multiplied by all block elements and each element is clipped to maximum value.

\[ SF_1 = \frac{1}{\sqrt{\sum_{i=0}^{35} b_i^2 + 3.6}} \]  

\[ b_i = \min(b_i \times SF_1, 0.2) \]  

The second stage, depicted in expressions (15) and (16), performs a similar operation over the block descriptor, but without applying any clipping.

\[ SF_2 = \frac{1}{\sqrt{\sum_{i=0}^{35} b_i^2 + 0.001}} \]  

\[ b_i = b_i \times SF_2 \]  

F. Classification

Each feature vector results from coupling all of 105 HOG features within the relevant detection window, according to expression (17).

\[ \mathbf{d} = (b_0, b_1, ..., b_{103}, b_{104}) \]  

Since indexes are consistent with a vertical scanning of the detection window, each block position is implicitly coded in the feature vector.

Each feature vector is classified through the expression (18). This SVM integrates the training data (vector t and constant b) and generates a score for each window.

\[ \text{Score} = b + \sum_{i=1}^{3780} d_i \times t_i \]  

A null or negative score represents a negative detection, while a positive score corresponds to a positive detection, with confidence proportional to the score modulus.

G. Multi-scale detection

Multi-scale detection is accomplished by applying the single scale algorithm over the distinct levels of an image pyramid, like the one in figure 6.

![Fig. 6. Image pyramid with three scales.](image)
IV. INPUT/OUTPUT DATA

Since the detection algorithm processes each frame independently, they are sent to the core separately. Each frame is composed of all its channels integrally and sequentially stored. Grayscale frames, with only one channel, are sampled sequentially using a depth of 8 bits per pixel. Color frames require three subsequent channels (B, G and R), each one organized like the single channel of a grayscale image.

The core produces a detection vector for each processed frame. Each detection, specifically its detection window, is identified through the coordinates of its upper-left and lower-right vertices. The core output buffer can store up to 2048 detections per frame.

V. SOFTWARE IMPLEMENTATION

The software implementation of the detector includes some optimizations for the FireWorks processor. These include a refined single-scale algorithm, fixed-point operation, simple image resizing with nearest neighbor interpolation and the exclusion of post-processing.

A. Single-scale detection

The single-scale algorithm performs the first stages of the detection over the full image, obtaining the votes of all pixels before starting the windows scanning. This approach allows the padding to be based only on pointers, avoiding the explicit construction of the padded image.

Subsequently, the image windows are scanned, line by line. In each detection window, the correspondent feature vector is constructed and classified, producing the window score. All extracted HOG features, in this stage, are stored into a hash table, allowing their reuse between distinct windows. This approach guarantees that after the first line of detection windows, each analyzed window only requires the extraction of some HOG features, reusing all other required features from the hash table.

In each block, all cell histograms are computed simultaneously, avoiding redundant operations in the overlapping pixels between different accumulation zones.

B. Multi-scale detection

The multi-scale detection starts by applying the single-scale algorithm over the original image. Then, the image is scaled according to the scale factor, and the single-scale algorithm is repeated, considering the smaller image. The process is repeated, each time considering smaller images, while the resizing produces images of higher resolution than the detection window (64 × 128 pixels). The resizing algorithm performs nearest neighbor interpolation, using only fixed-point operations in the format Q16.16.

C. Fixed-Point arithmetic

Fixed-point arithmetic reduces the processing time on the FireWorks processor and makes feasible the development of a SideWorks accelerator. By default, the algorithm uses the format Q12.20, using the format Q16.16 in the square root operations and the format Q24.40 in some detection stages, such as the gradients calculation, the normalization and the classification. All 64-bit results are converted to a 32-bit format before being stored into memory for further processing.

The lack of support for 64-bit operations on the FireWorks processor requires their implementation through software.

Sums and subtractions of 64-bit integers rely on two distinct assembly instructions, each one considering only halfwords of the operands. The desired 64-bit operation is achieved by propagating the carry/borrow bit between operations.

Multiplications also require one extra assembly instruction, which returns the most significant 32 bits of the result. These are combined with the remaining 32 bits of the result, obtained through the common multiplication instruction, for conversion to the desired fixed-point format.

The division and square root operations are adapted from the fixed-point arithmetic library libfixmath [28]. Both operations rely on iterative algorithms which essentially perform sums and shifts. However, the division algorithm also requires 32-bit integer divisions, implemented using the available hardware divisor, and a Count Leading Zeros (CLZ) instruction, also present in the FireWorks.

Lastly, the arctangent algorithm is adapted from the OpenCV library [27]. The double precision floating-point operations are converted to fixed-point, using exclusively the format Q12.20. The algorithm requires a fixed sequence of divisions, sums, multiplications and comparisons.

VI. HARDWARE/SOFTWARE IMPLEMENTATION

The stages of the detection algorithm are reorganized for hardware/software co-processing, extending the window-based processing to more stages. This approach allows the SideWorks to perform more detection stages, contributing to a higher overall acceleration. So, in addition to all stages previously performed over detection windows, the gamma correction, gradients and votes are also executed in the SideWorks. Only the padding and resizing stages are performed over the full image by the FireWorks processor.

A. Single-scale detection

The single-scale detection is performed through the SideWorks accelerator, which processes detection windows. After the reception of an image and all necessary initializations, the FireWorks processor pads the image and sends all required LOADs and RUNs to SideWorks. SideWorks processes all detection windows, writing their scores sequentially in the score vector in memory. When SideWorks concludes its processing, FireWorks reads the score vector, searching for detections and writing them in the core output buffer.

In this approach, the reutilization of HOG features is performed in the SideWorks accelerator. Therefore, two distinct types of windows are considered: full and incremental. The first window in each line of detection windows is always a full window, while all remaining windows are incremental windows. Each full window computes all of its HOG features to produce its feature vector. However, one incremental window only computes the last row of HOG features within the
window, combining them with the previous window descriptor to produce its own feature vector. So, each full window requires an area of 66 × 130 pixels, while incremental windows only need 18 × 130 pixels, taking into account that the gradients computation requires padding around each window.

**B. Multi-scale detection**

In multi-scale detection, FireWorks waiting periods are used for the computation of the next image pyramid level, allowing for the concurrent operation of FireWorks and SideWorks. To accomplish this, FireWorks requires two input buffers into SideWorks. Figure 7 illustrates the multi-scale operation.

![Fig. 7. Multi-scale detection temporal diagram with two buffers.](image)

After receiving each image, FireWorks applies the required padding and stores the padded image in one buffer, instructing SideWorks to process all detection windows in that buffer. Afterward, instead of waiting for SideWorks to finish its processing, FireWorks computes the next level in the image pyramid (resizing and padding), storing it in the available buffer. When SideWorks finally concludes its processing, FireWorks reads the produced score vector and writes the found detections to the core output buffer. The input buffers are then switched and SideWorks starts processing the previously computed pyramid level. After sending all the required RUNs and LOADs, FireWorks computes the next pyramid level and stores it in the now available input buffer. This process is repeated while there are still image pyramid levels left to process. The final detections vector of each frame is only sent after processing all required pyramid levels.

**VII. SideWorks**

The designed SideWorks processes each detection window independently. After receiving each window (full or incremental), SideWorks executes a sequence of datapaths, which ultimately produce the window score, writing it into memory through DMA.

**A. Window loading**

The windows are loaded using SideWorks LOAD commands, which allow the loading of sequential lines from memory to SideWorks input memories, through DMA. However, this operation requires SideWorks to know the size of each image line. So, before any other command, FireWorks sets the image resolution, enabling SideWorks to load windows. This value is updated each time the image processed by the SideWorks changes resolution.

Since color image channels are stored independently, each window requires three distinct LOADs, one for each channel. Logically, grayscale images only need one LOAD per window. Full windows transfer 130 lines of 66 pixels each, while incremental windows only require 130 lines of 18 pixels each.

1) **Input memories**: The developed SideWorks input memories allow three keypoints: (1) Asymmetrical reading and writing - since the writing is based on 32-bit words and reading relies on 8-bit words; (2) Additional input data removal - since the horizontal dimensions of windows (66 and 18 pixels) are not multiple of 32-bit words, the loading operation must ignore the last 16-bits of each loaded line; (3) Simultaneous read of independent loaded channels - although each channel is independently loaded, all channels are read simultaneously.

For the desired operation, each input memory requires two dual port memories for each image channel, which equates to six memories overall.

Each LOAD selects the relevant channel according to its writing base address. Once the desired channel is selected, the writing process uses two internal counters to address the memories of that channel. In each LOAD, both counters start at zero, being incremented every time a new word for writing is received through DMA. Each memory stores a halfword of the 32-bit received word. When a line transition occurs, one of the counters is decremented by one and the halfwords sent to each memory are switched. Therefore, the least significant halfword, previously written in one of the memories, is replaced by the first most significant halfword of the new line, replacing the last 16 bits of the previous line. In the next line transition, the alternative counter is decremented and the halfwords sent to each memory are switched again. This procedure is repeated throughout the loaded window, guaranteeing the sequential writing of its pixels and ignoring the last 16 bits of every received line.

During the reading operation, both memories of each channel are read simultaneously using only the 12 most significant bits of the reading address. Their outputs are then combined to form a 32-bit word, using the remaining 2 bits of the address to select the desired pixel (8 bits). Furthermore, all channels of one pixel are simultaneous read by concurrently addressing all three channels (six memories).

**B. Window processing**

Figure 8 shows the sequence of datapaths used for processing both full and incremental windows.

Both types of windows require essentially the same datapaths. But, incremental windows require an additional datapath, allowing the utilization of the previous window feature vector. Since incremental windows require fewer operations, most datapaths account for both operation modes through their arguments.

Datapath grad_1 receives the window width and the number of pixels within the window as arguments, allowing it to process both types of detection windows. Datapaths grad_2, grad_3 and votes receive the number of pixels inside the window as an argument, permitting them to process windows of varying sizes. Datapath cellhist receives the width of the
considered window as an argument, enabling it to properly address its input memories. Additionally, since incremental windows compute fewer HOG features, they require fewer RUNs of this datapath. The partial normalization, required for incremental window processing, is assured by the arguments sent to all normalization datapaths, including the address of the first block to normalize and the number of blocks to process. Only the classification datapath, which receives the score vector offset as an argument, performs strictly the same operation for both window types.

Overall, each full window requires \(222\) distinct RUNs, while each incremental window only needs 43 RUNs.

### C. Memories

Figure 9 shows the memory utilization in the datapaths, illustrating SideWorks data-flow.

![Memory Utilization Map](image)

Fig. 9. Memory utilization map. Full dots represent writing ports and empty ones correspond to reading ports.

Overall, SideWorks contains seven internal memories (\(M0\) – \(M6\)), six ROMs (\(RS0\) – \(RS2\), \(RW\) and \(RT0\) – \(RT1\)) and four input memories in the ping-pong scheme (MEM). Two internal memories (\(M5\) and \(M6\)) are used exclusively for storing the feature vector, allowing its reutilization between windows. The first three ROMs (\(RS0\) – \(RS2\)) store the square root values used during gamma correction. ROM \(RW\) stores the multiplication factors required for the histograms computation. Finally, the last two ROMs (\(RT0\) – \(RT1\)) store the training data used during classification.

### D. Datapaths

This section briefly describes each datapath, stating all performed operations, the available options, latencies, throughputs and runtimes.

1) **Datapath grad 1:** This datapath performs the gamma correction, computes both the required convolutions (horizontal and vertical) and the sum of their squares for each pixel in the window. All channels of each pixel are processed simultaneously. The gamma correction is applied through the used ROMs (\(RS0\) – \(RS2\)). For each pixel, the channel with the highest sum of convolution squares (gradient magnitude) defines the final results for that pixel. The datapath has an argument, for grayscale images, which instructs it to ignore two channels, selecting all results from one predetermined channel. After the initial latency of 20 clock cycles, this datapath processes one pixel (three writes) each two clock cycles. Overall, it requires 16404 clock cycles for each full window and 4116 for every incremental window.

2) **Datapath grad 2:** This datapath performs the square root operation, computing the gradient magnitude for every pixel within the window. After the initial latency of 218 clock cycles, a new square root is computed every clock cycle. Overall, the datapath requires 8410 clock cycles for full windows and 2266 for incremental ones.

3) **Datapath grad 3:** This datapath implements the arctangent operation, computing the gradient angle of each pixel inside the window. Furthermore, it also applies the correction found in expression (6). This datapath has a latency of 259 clock cycles, subsequently producing a new result every clock cycle. In total, it requires 8451 clock cycles for each full window and 2307 per incremental window.

4) **Datapath votes:** This datapath computes the votes of each pixel within the window, through their gradients (angle and magnitude). The votes of two pixels are computed simultaneously to fully utilize the datapath memories. Since the bins of both votes of each pixel only require 16 bits, they are written in the same memory. After the initial latency of 12 clock cycles, the datapath processes two pixels each clock cycle, generating four votes. Overall, this datapath requires 4108 clock cycles for full windows and 1036 clock cycles for incremental windows.

5) **Datapath cellhist:** This datapath computes two histograms from cells within the same block. It starts by clearing all used memory positions. Then, for each histogram, the datapath considers both votes of each pixel in the accumulation area, multiplying them by their respective multiplication factors, stored in the ROM \(RW\), and accumulating them in dedicated accumulators (one for each histogram class). After
updating a class, its updated results are written to memory. The datapath splits the feature vector between two internal memories, writing each simultaneously computed histogram in a distinct memory. This way, one memory stores half of each block descriptor (two cell histograms). This datapath has 13 clock cycles of latency, subsequently updating two classes per cell histogram per clock cycle in all its remaining 144 clock cycles.

6) Datapath norm_1: This datapath initiates the normalization process, computing the sum of all element squares of each block. Considering the feature vector distribution between two internal memories, each one containing two cell histograms of every block, this datapath reads a complete block in 9 clock cycles. The accumulation is performed in the format \( Q_{24.40} \), subsequently converting the result to \( Q_{16.16} \) before writing it to memory. After the initial latency of 21 clock cycles, this datapath produces a new result every nine clock cycles. Overall, it requires 958 clock cycles for each full window and 148 clock cycles for each incremental one.

7) Datapath norm_2: This datapath computes all required denominators for each normalization stage. Therefore, the datapath computes the square root of each block sums, subsequently adding them to a constant (3.6 or 0.001) received through an argument. After the initial latency of 219 clock cycles, this datapath produces one new denominator per clock cycle. This datapath requires 324 clock cycles for each full window and 234 for each incremental one.

8) Datapath norm_3: This datapath computes the reciprocal of each previously calculated denominator, obtaining all scale factors required in the normalization. This datapath has a latency of 222 clock cycles, subsequently computing a new result every cycle. Overall, it requires 327 clock cycles for each full window and 237 for each incremental window.

9) Datapath norm_4: This datapath finishes the first stage of normalization and begins the second stage. Therefore, each previously computed scale factor is multiplied by the correspondent block elements, subsequently applying the clipping to each element. After that, the datapath applies the same operation as datapath norm_1, summing all element squares for each block. The first operation starts producing results after 11 clock cycles, while the second has 26 clock cycles of latency. In every subsequent 9 clock cycles, the datapath finishes a block normalization and computes a new sum. Overall, this datapath requires 963 clock cycles to process each full window and 153 for every incremental one.

10) Datapath norm_5: This datapath concludes the second stage of the normalization process, multiplying each block element by the previously computed scale factor. This datapath has 10 clock cycles of latency, subsequently processing a new block each 9 clock cycles. Overall, it requires 955 clock cycles for full windows and 145 for incremental ones.

11) Datapath svm: This datapath evaluates the feature vector in memory, computing the respective window score. The datapath multiplies each feature vector element by its corresponding value in the training data vector, stored in ROMs \( RT0 \) and \( RT1 \). The products are then summed through 64-bit accumulations, producing the window score. Subsequently, the datapath requests access to the external memory, waiting for it to become available before writing the computed score through DMA. Hence, this datapath produces one result with variable runtime, which is always greater or equal than 960 clock cycles.

12) Datapath reuse: This datapath shifts the feature vector stored in memory, allowing its partial reutilization between neighboring windows on the same line. According to the used window stride (8 pixels), the first 15 HOG features of the feature vector are removed, shifting the remaining 90 features to its beginning. Since the feature vector is split between two internal memories, they are both shifted. This datapath has 4 clock cycles of latency, subsequently performing two writes per clock cycle, one in each internal memory. Overall, it requires 1624 clock cycles.

E. Square-root and division

The fixed-point algorithms for square root and division require two essential modules (controller and multiplier), repeating them throughout one pipeline to perform the desired operation. In each pipeline level, the controller analyses the previous stage result, comparing it to an initial argument. Through this comparison, the controller determines one bit of the result, updating its current estimate. Afterwards, the controller computes new guess values required in the next stage. Figure 10 shows the square root and division controller.

![Fig. 10. Square root and division controller.](Image)

Considering the 32-bit operands and the fact that the algorithm only allows positive integers, each operation requires 31 sequential stages. Accordingly, each square root has a latency of 213 clock cycles while the division requires 217 clock cycles. Figure 11 shows the pipelines for each one of the required operations.

VIII. RESULTS

The developed core was implemented and tested on a FPGA, specifically the device 5AGTDF7K3F40I3 from Altera, having produced the expected results. The used clock frequency was 100MHz. The scaling factor for multi-scale detection was 1.2 [17].

A. Fixed-point algorithm

This section evaluates the detection performance of the implemented fixed-point (FP) algorithm against the original double-precision floating-point implementation from the OpenCV library (OCV).
The Detection Error Tradeoff (DET) curve measures the proportion of true detections against the proportion of false positives, essentially conveying the same information as Receiver Operating Characteristic (ROC) curves while allowing a better distinction between smaller probabilities [29].

Figure 12 plots the miss rate vs. False Positives Per Window (FPPW) and figure 13 plots the miss rate vs. False Positives Per Image (FPPI). Lower miss rates, at lower FPPWs or FPPIs, represent better detection performance. The detection threshold was adjusted to obtain each point on the curves, starting from the highest threshold to the lowest. Both plots were obtained using the test examples from the INRIA Person dataset [3].

B. Temporal results

The detection system was evaluated considering different resolutions, image types (BGR or Grayscale) and types of detection (single-scale or multi-scale). All per frame results were measured since the frame reception until the writing of its last detection in the core output buffer.

Figure 14 illustrates the Single-Scale (SS) final system speedups against the equivalent algorithm executed on FireWorks.

Considering the grayscale results, we conclude that for SS operation, the complete system is eight times faster than the competing alternative. The observed speedups decrease with higher resolutions, since the software implementation is progressively more efficient, thanks to its full reutilization of HOG features. The color image results (BGR) present lower speedups, probably due to the increased time required to pad all image channels. However, their speedup is never lower than seven times.

Figure 15 represents the Multi-Scale (MS) final system speedups against the software implementation. Overall, the MS speedups are higher than the SS ones, allowing us to conclude that the double input buffers do allow the concurrent processing of SideWorks and FireWorks. This effect is especially relevant at lower resolutions where the processing time is more evenly distributed. With increasing resolutions, SideWorks requires a progressively higher proportion of the overall detection time to process all windows.
The proposed detection system is unable to perform real-time detection. Even considering the lower resolution images (QVGA), with 320 × 240 pixels, the detection requires 0.20s over grayscale images and 0.26s for color images. These processing times are definitely incompatible with real-time processing. In fact, for most resolutions, the time spent by processing the detection at 100 frames per second, in Computer Vision and Pattern Recognition, 2012 IEEE Conference on. IEEE, 2012, pp. 2903–2910.

Although not able to perform the detection in real-time, the proposed system shows a speedup of over seven times when compared with the equivalent algorithm executed on the embedded processor. Further optimizations, in the algorithm and co-processor, are required to achieve higher accelerations.

IX. CONCLUSIONS

Although not able to perform the detection in real-time, the proposed system shows a speedup of over seven times when compared with the equivalent algorithm executed on the embedded processor. Further optimizations, in the algorithm and co-processor, are required to achieve higher accelerations.

REFERENCES