An Implementation of the Histogram of Oriented Gradients (HOG) Algorithm on a Reconfigurable System on Chip

Υλοποίηση του Histogram of Oriented Gradients(HOG) αλγορίθμου σε μια επαναπρογραμματίσιμη πλατφόρμα ολοκληρωμένου κυκλώματος

Katsaros Nikolaos

Supervisor: Assoc. Prof. Bellas Nikolaos

2^{nd} committee member: Assoc. Prof. Gerasimos Potamianos

A Thesis submitted in fulfillment of the requirements
for the degree of Diploma Thesis
in the
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

June 2018

An Implementation of the Histogram of Oriented Gradients (HOG) Algorithm on a Reconfigurable System on Chip

Υλοποίηση του Histogram of Oriented Gradients(HOG) αλγορίθμου σε μια επαναπρογραμματίσιμη πλατφόρμα ολοκληρωμένου κυκλώματος

Katsaros Nikolaos

Supervisor: Assoc. Prof. Bellas Nikolaos

2nd committee member: Assoc. Prof. Gerasimos Potamianos

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την ....

....	....
Ν. Μπέλλας	Γ. Ποταμιάνος
Αναπληρωτής Καθηγητής	Αναπληρωτής Καθηγητής

Dedicated to

my family

Υλοποίηση του Histogram of Oriented Gradients(HOG) αλγορίθμου σε μια επαναπρογραμματίσιμη πλατφόρμα ολοκληρωμένου κυκλώματος

Περίληψη

Ο Histogram of Oriented Gradients είναι ένας από τους πιο δημοφιλείς αλγορίθμους όρασης υπολογιστών που χρησιμοποιούνται στην αναγνώριση αντικειμένων και στην συγκεκριμένη περίπτωση στην αναγνώριση πεζών. Παρουσιάζουμε μία υλοποίηση του αλγορίθμου σε μία ενσωματωμένη πλατφόρμα, συγκεκριμένα στην πλακέτα ανάπτυξης Zedboard η οποία περιέχει το ολοκληρωμένο κύκλωμα Xilinx Zynq®-7000 All Programmable το οποίο αποτελείται από ένα διπύρινο επεξεργαστικό σύστημα ARM-A9 καθώς και επαναπρογραμματίσιμη λογική (FPGA). Η πλατφόρμα αυτή μας έδωσε την δυνατότητα να εκτελέσουμε ταυτόχρονα διαφοερικά μέρη του αλγορίθμου τόσο στον επεξεργαστή, όσο και στην FPGA. Η υλοποίησή μας πετυχαίνει επιτάχυνση (speedup) ίση με 384x σε σύγκριση με την single-threaded υλοποίηση στον ARM επιτυγχάνοντας απόδοση πραγματικού χρόνου έχοντας την δυνατότητα να επεξεργαστεί 10 εικόνες VGA ανάλυσης (640 X 480 pixels) το δευτερόλεπτο με ελάχιστη μείωση στην ακρίβεια ανίχνευσης.

An Implementation of the Histogram of Oriented Gradients (HOG) Algorithm on a Reconfigurable System on Chip

Abstract

The Histogram of Oriented Gradients is one of the most popular computer vision algorithms used for object detection and in our case for pedestrian detection. We present a implementation of the algorithm on an embedded platform,namely the Zedboard development board which contains the Xilinx Zynq®-7000 All Programmable SoC that is comprised of both a Dual ARM-A9 processing system and an FPGA programmable logic. This platform gave us the ability to execute different parts of the algorithm simultaneously both on the CPU and the FPGA. Our FPGA implementation achieves a speedup of 384 (compared with a single-threaded software implementation in ARM) and achieves real-time pedestrian detection at 10 VGA (640 X 480 pixels) images per second with very small decrease in detection accuracy.

Acknowledgements

First and foremost, I would like to extend my deepest appreciation to my advisor, Prof. Nikolaos Mpellas, for his invaluable support and guidance, which has been instrumental in the development of this thesis. I would like to offer my special thanks to Prof. Gerasimos Potamianos for his advices on the Machine Learning part of our thesis.

I am grateful to my friend Nikolaos Patsiatzis for our excelent collaboration, while working on our dissertations and several projects through the last 3 years.

To my friends (and fellow students) in Volos, thank you for all your support specially during the last year of our studies. We shared experiences and moments that I will never forget.

To my family, thank you for encouraging me in all of my pursuits and inspiring me to follow my dreams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pedestrian detection is an integral part of any video surveillance system with numerous applications in numerous computer vision fields. For example, pedestrian detection is widely deployed in the automotive sector where it is involved in many ADAS(Advanced driver-assistance systems) developed by car manufacturers. ADAS comprises one or many subsystems that may provide information to the driver, thus increasing her awareness of the situation around her and actuating vehicle subsystems to prevent dangerous situations. Pedestrians represent a significant amount of fatalities in road accidents. The integration of pedestrian detection algorithms in ADAS, like the brake assistance system, is estimated to have reduced fatal and serious injuries among pedestrians by 10% according to [1] . Therefore, it comes as no surprise that pedestrian detection is one of the most important research topics in computer vision. Needless to say that real world problems such as accident avoidance systems require real-time processing capabilities.

Although various kind of methods for extracting feature data for pedestrian detection system are proposed, HOG (Histogram of Oriented Gradients) proposed N.Dalal and B.Triggs [2] has been proven to be one of the most effective ones. As a result, we decided to implement a HOG-like detector while keeping in mind it's basic characteristics.

Furthermore, due to the parallelizable and deterministic nature of the algorithm, using an embedded FPGA platform seemed like a rational solution. In addition, another great advantage of these platforms compared to GPUs or multicore CPUs is the significantly lower power consumption. Hence, it is more feasible to embed them on road vehicles.

1.1 Contributions

In this thesis, a thorough algorithmic and architectural exploration is performed. Several approximation techniques are proposed which result in no significant loss of the overall accuracy. Furthermore, since machine learning is a integral part of this algorithm, some research into its properties is performed and a modification of its parameters is introduced. Finally, contrary to the usual RTL implementations, a high level synthesis approach is examined proving that this method can also yield real-time results.

The initial implementation of the algorithm executed in the ARM processor of the Zedboard even with the -O3 optimization flag needed 30 seconds for one image, while our final embedded system is 384 times faster compromising only less that 1% of the overall accuracy.

1.2 Thesis structure

This thesis is divided into 7 main chapters, each one those includes smaller sections and possibly subsections.

Chapter 2 provides a background of the algorithm. All the steps are described at length, from the point that an image is captured until one that the algorithm classifies whether the image contains a pedestrian or not.

Chapter 3 provides information necessary for one to understand the development process and the hardware used in this project. At first, it describes the architecture and operation of FPGAs in general and then focuses on the technical characteristics of the ZedBoard, the board used for development. Two different hardware development approaches are compared regarding their design time and application performance.

Chapter 4 begins with a brief overview of the initial implementation and several modifications compared to the original one are illustrated. Moreover, each optimization step is analyzed and their impact on performance and accuracy are examined.

In Chapter 5 the hardware implementation is presented along with several optimization steps that aim to increase the throughput of the design.

Chapter 6 is an overview of the embedded system in general. It describes how the implemented hardware is used as a peripheral of the embedded operating system as well as the embedded application flow.

Chapter 7 mentions some related works and their contributions.

Finally, Chapter 8 concludes the thesis, stating our contributions and the future work.

Chapter 2

Histogram of Oriented Gradients

The Histogram of Oriented Gradients is a computer vision algorithm widely used for object detection. The input of the algorithm is an image or a video frame and the output is a prediction of whether and where this object exists in the image or frame.

2.1 HOG flow

The HOG detector, shown in Fig. 2.2, is a sliding window algorithm. This means that for any given image a detection window of 64 pixels horizontally and 128 vertically is moved following a specific pattern at all locations and scales and a descriptor is computed for this window. The window scans the input image with a stride of 32 pixels horizontally and 64 pixels vertically as shown in Fig. 2.1.

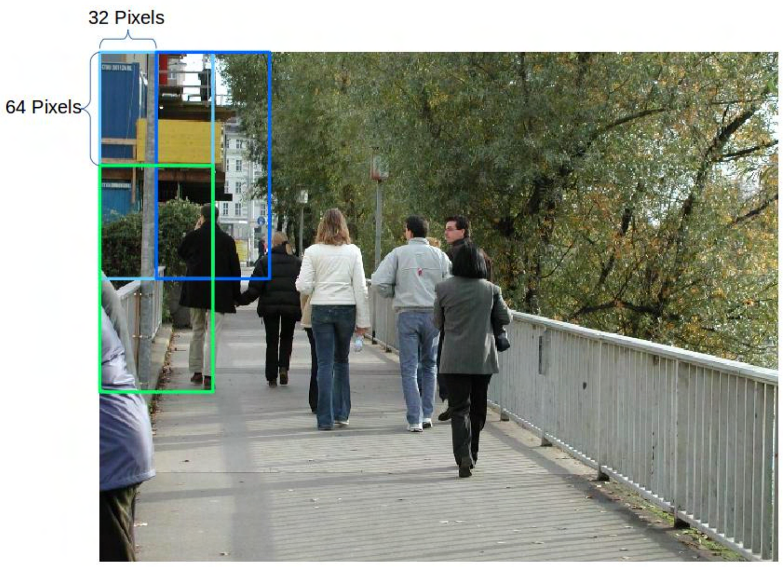


Figure 2.1: Sliding windows inside a frame

For each window a pre-trained classifier is used to assign a matching score to the descriptor. The classifier used is a linear SVM classifier and the descriptor is based on histograms of gradient orientations. Nearby detections of the same object are common with sliding-window frameworks and are typically merged using non-maxima suppression approaches in order to yield bounding boxes with confidence levels for the final detections.

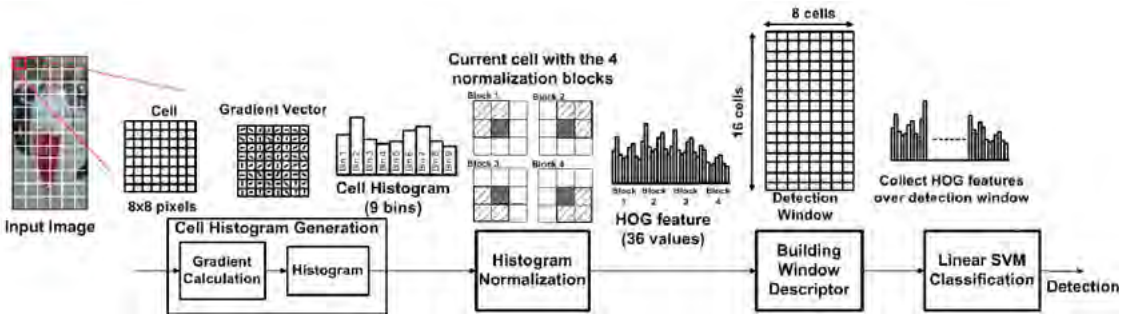


Figure 2.2: Overview of the HOG algorithm [3]

Firstly the image is padded and a gamma normalization is applied. Padding means that extra rows and columns of pixels are added to the image. This helps the algorithm deal with the case when a person is not fully contained inside the image. The gamma normalization has been proven to improve performance for pedestrian detection but it may decrease performance for other object classes. To compute the gamma correction the color for each channel is replaced by its square root.

2.2 Gradient vector of pixel magnitudes

An image gradient is a directional change in the intensity or color in an image. The gradient of the image is one of the fundamental building blocks in image processing. Mathematically, the gradient of a two-variable function (here the image intensity function) at each image point is a 2D vector with the components given by the derivatives in the horizontal and vertical directions. At each image point, the gradient vector points in the direction of largest possible intensity increase, and the length of the gradient vector corresponds to the rate of change in that direction.

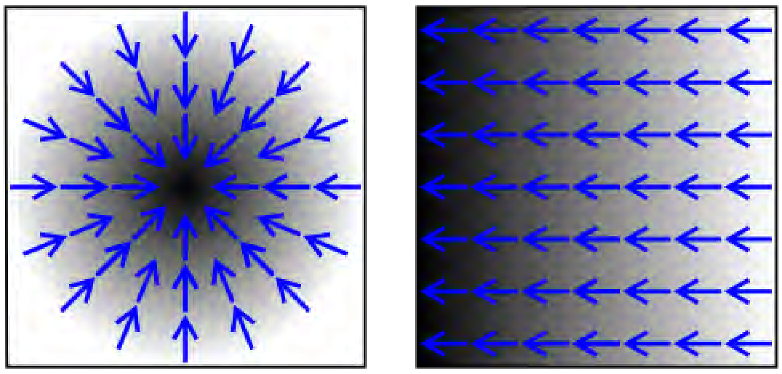


Figure 2.3: Gradient magnitudes and angle representation

In Fig. 2.3 the gradient magnitudes are represented in black and white, black representing higher values, and its corresponding angle is represented by blue arrows.

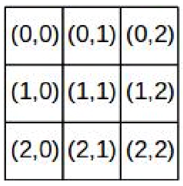
Gradient orientations and magnitude are obtained for each pixel from the pre-processed image. In RGB images the color with the maximum magnitude value(and its corresponding orientation) is chosen. The gradient of each pixel is computed by convoluting it with the 1-D centered kernel [-1 0 1] by rows and columns.

$$Gx = Mx * I, \quad Mx = [-1\ 0\ 1] \tag{2.2.1}$$

$$Gy = My * I, \quad My = [-1\ 0\ 1]^T \tag{2.2.2}$$

$$|G(x,y)| = \sqrt{Gx(x,y)^2 + Gy(x,y)^2} \tag{2.2.3}$$

Take for instance the array in Fig. 2.4. For $I = (1,1)$ the horizontal gradient would be $Gx = pixel(1,2) - pixel(1,0)$ and the vertical one $Gy = pixel(2,1) - pixel(0,1)$.



(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Figure 2.4: Pixel array representation

The gradient orientation angle can be calculated as:

$$tan((x,y)) = \frac{Gy(x,y)}{Gx(x,y)} \tag{2.2.4}$$

The Fig 2.5 shows the effect of applying the gradient function in a random part of an input image.

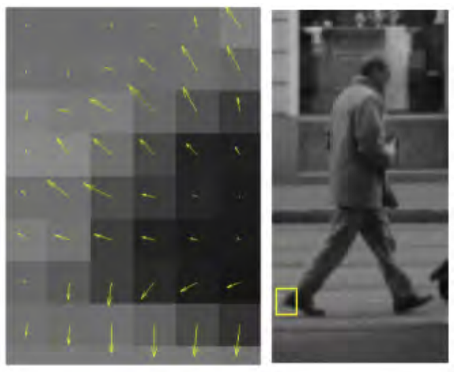


Figure 2.5: Pedestrian gradient example from the INRIA dataset [4].

2.3 Histogram extraction

A histogram is an accurate representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable. Histograms give a rough sense of the density of the underlying distribution of the data, and often for density estimation: estimating the probability density function of the underlying variable.

The detection window is divided into 8x16 rectangular local spatial regions called cells. Each cell consists of 8x8 pixels which are then discretized into 9 angular bins according to their gradient orientation. The bin of each pixel is computed as :

$$bin = (arctan\frac{Gy}{Gx}) \div 20^{\circ} \tag{2.3.5}$$

Each pixel contributes a weighted vote for its corresponding angular bin, the vote is a function of the gradient magnitude at the pixel. This way the information is compressed to a 9-dimensional space per cell. The angular histogram bins are evenly spaced over 0°- 180°.

For example, let $Gy = 30$ and $Gx = 70$. Therefore, the bin would be:

$$(arctan\frac{30}{70}) \div 20 = 23.2^{\circ} \div 20^{\circ} = 1$$

In addition, to reduce aliasing along spatial dimensions, votes are interpolated trilinearly between the neighbouring bin centres in both orientation and position.

Let h(x) denote the value of the histogram for the bin centred at x.Let w at the 3-D point x = [x, y, z] be the weight to be interpolated. Let x1 and x2 be the two corner vectors of the histogram cube containing x, where in each component x1 <= x < x2. Assume that the bandwidth of the histogram along the x, y and z axis is given by b = [bx, by, bz]. Trilinear interpolation distributes the weight w to the 8 surrounding bin centres as follows:

$$
\begin{aligned}
h(x1,y1,z1) &\leftarrow h(x1,y1,z1) + w\Big(1-\frac{x-x1}{bx}\Big)\Big(1-\frac{y-y1}{by}\Big)\Big(1-\frac{z-z1}{bz}\Big)\\
h(x1,y1,z2) &\leftarrow h(x1,y1,z2) + w\Big(1-\frac{x-x1}{bx}\Big)\Big(1-\frac{y-y1}{by}\Big)\Big(\frac{z-z1}{bz}\Big)\\
h(x1,y2,z1) &\leftarrow h(x1,y2,z1) + w\Big(1-\frac{x-x1}{bx}\Big)\Big(\frac{y-y1}{by}\Big)\Big(1-\frac{z-z1}{bz}\Big)\\
h(x2,y1,z1) &\leftarrow h(x2,y1,z1) + w\Big(\frac{x-x1}{bx}\Big)\Big(1-\frac{y-y1}{by}\Big)\Big(1-\frac{z-z1}{bz}\Big)\\
h(x1,y2,z2) &\leftarrow h(x1,y2,z2) + w\Big(1-\frac{x-x1}{bx}\Big)\Big(\frac{y-y1}{by}\Big)\Big(\frac{z-z1}{bz}\Big)\\
h(x2,y1,z2) &\leftarrow h(x2,y1,z2) + w\Big(\frac{x-x1}{bx}\Big)\Big(1-\frac{y-y1}{by}\Big)\Big(\frac{z-z1}{bz}\Big)\\
h(x2,y2,z1) &\leftarrow h(x2,y2,z1) + w\Big(\frac{x-x1}{bx}\Big)\Big(\frac{y-y1}{by}\Big)\Big(1-\frac{z-z1}{bz}\Big)\\
h(x2,y2,z2) &\leftarrow h(x2,y2,z2) + w\Big(\frac{x-x1}{bx}\Big)\Big(\frac{y-y1}{by}\Big)\Big(\frac{z-z1}{bz}\Big)
\end{aligned}
$$

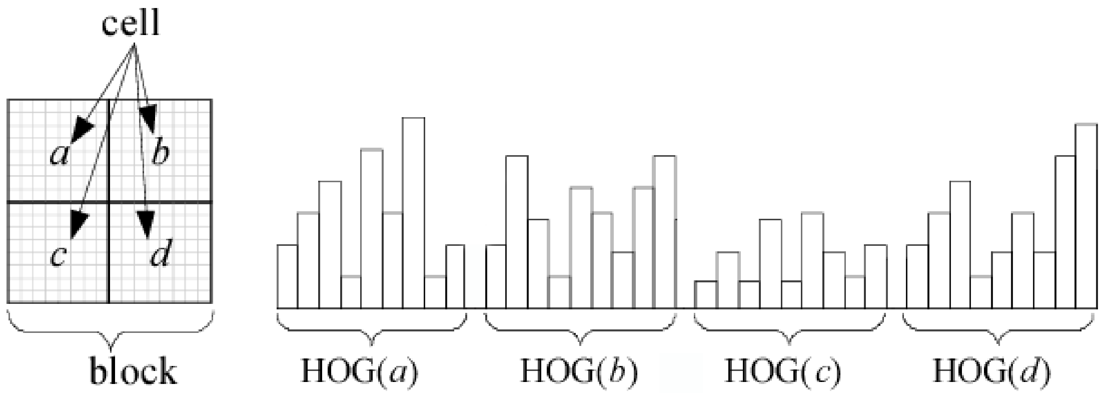


Figure 2.6: The concatenated HOG descriptor

The HOG descriptor is represented by a concatenation of all these blocks as it can be seen in Fig 2.6. In fact, blocks overlap with each other so that each cell response appears several times in the final feature vector. The default block stride is 8 pixels (1 cell), resulting in a fourfold coverage of each cell. To summarize, as shown in Fig. 2.7 each detection window is represented by 7x15 blocks. The number of horizontal blocks is calculated as

$$\frac{Window\,Width}{Block\,Stride} - 1$$

and the number of vertical blocks as

$$\frac{Window\ Height}{Block\ Stride} - 1$$

, a block consisting of 2x2 cells, a cell is represented by a 9-bin histogram, giving a total of (7x15)x(2x2)x9 = 3780 features.

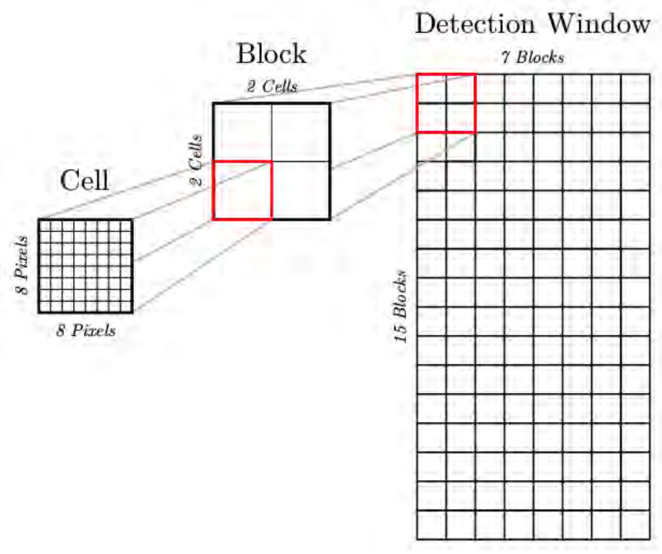


Figure 2.7: Division of the detection window into blocks and cells

2.4 Histogram Normalization

The next step of the HOG algorithm is to normalize the histogram descriptors before SVM classification. Normalization refers to adjusting values measured on different scales to a common scale, often prior to averaging. In more complicated cases, normalization may refer to more sophisticated adjustments where the intention is to bring the entire probability distributions of adjusted values into alignment. Gradient strengths may vary over a wide range due to shadows, local variations in illumination and foreground-background contrast. Therefore local contrast normalization is essential for good performance. For this purpose, groups of 2x2 adjacent cells are considered as spatial regions called blocks. Each block is represented by a concatenation of the corresponding four cell histograms, resulting in a 36-dimensional feature vector that is normalized to unit length, using the L2 norm. The following algorithm implements Histogram Normalization for one block.

Algorithm 1 L2-Hys

```
1: procedure NORMALIZATION($v$)
2:     $sum = \sum_{i=0}^{35} v[i]^2$
3:     for each 36-D feature vector do
4:         $v[i] = min((\frac{v[i]}{\sqrt{sum+\epsilon^2}}), 0.2)$
5:     end for
6:     $sum = \sum_{i=0}^{35} v[i]^2$
7:     for each 36-D feature vector do
8:         $v[i] = (\frac{v[i]}{\sqrt{sum+\epsilon^2}})$
9:     end for
10:    Return $sum$
11: end procedure
```

2.5 Classification

In machine learning, Support Vector Machines (SVMs) are supervised learning models used for classification and regression. For binary classification, an SVM training algorithm builds a model that classifies new examples making it a robust non-probabilistic classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate classes are divided by the largest possible margin. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they are assigned.

The generated HOG descriptor is used to categorize the detection window into one of the predefined classes, pedestrian or non-pedestrian. For this classification step, Dalal&Triggs employ a linear support vector machine (SVM) which is both accurate and efficient in terms of performance. The SVM predictor hypothesis equation is:

$$y(x) = w^T * x + b$$

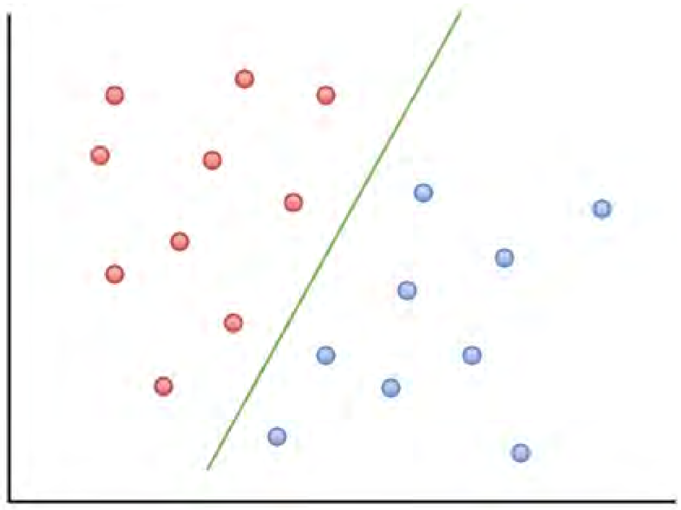


Figure 2.8: Linearly separable data points that belong to one of two classes, red and blue

The variable w represents the weight vector, x is the is the value of the input descriptor and is the bias. In case of HOG, we use the SVM^{light} toolkit [5] and the classifier is trained with the 3780-dimensional descriptor that is generated for 80% of the Daimler dataset [6] images resulting in 12528 positive (pedestrian) and 53952 negative (non-pedestrian) samples. The rest 20% of the dataset are used to test the accuracy of the system.

2.6 Non-maximal suppression

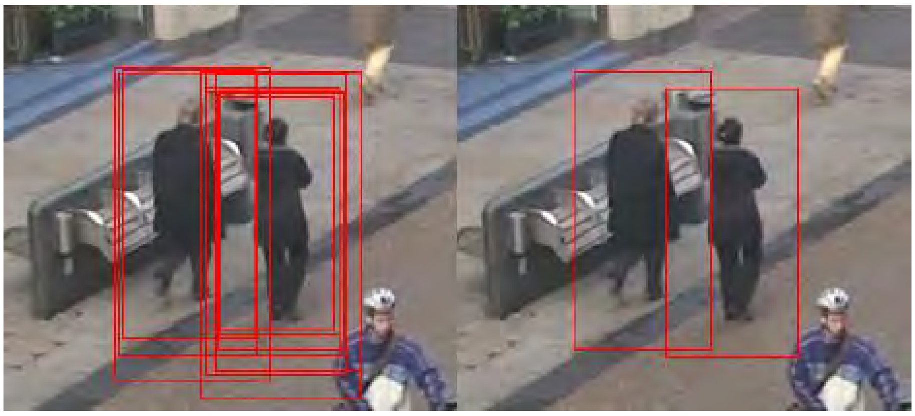


Figure 2.9: Non-maximal suppression reduces the effect of multiple detections of the same pedestrian

Invoking the SVM classifier across successive blocks and scales in the image may yield multiple detections for the same pedestrian (left of Fig. 2.9). Multiple overlapping detections need to be fused together. This is achieved using a mean shift algorithm in 3D position/scale space.

More specifically, all pedestrian detections are stored in a data structure along with their classification score as well as their coordinates. The Non-Maximal Suppression (NMS) algorithm scans this structure to find detections with more that 70% overlap. Detections with the lower score is removed from the structure. At the end of this process the output looks like the right of Fig. 2.9.

Chapter 3

Target Platform

3.1 Field Programmable Gate Arrays (FPGAs)

FPGAs are hardware-programmable semiconductor devices that consist of a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnects. As opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements. Although One-Time Programmable (OTP) FPGAs are available, the dominant type is SRAM-based which can be reprogrammed multiple times as the design evolves. Due to their programmability, FPGAs are ideal for a large variety of markets such as ASIC prototyping, Aerospace and Defense, Automotive, Communications, High Performance Computing, Industrial, Medical and Video and Image Processing.

3.2 Zedboard

The ZedBoard, shown in Fig. 3.1, is an evaluation and development board based on the Xilinx Zynq-7000 Extensible Processing Platform. As it is evident from Fig. 3.2 it combines a dual Cortex-A9 Processing System (PS) with Series-7 Programmable Logic (PL) cells, and therefore the Zynq-7000 EPP can be targeted for broad use in many applications. The Programmable Logic (PL) section is ideal for implementing high-speed logic, arithmetic and data flow subsystems, while the Processing System

(PS) supports software routines and/or operating systems, meaning that the overall functionality of any designed system can be appropriately partitioned between hardware and software. Links between the PL and PS are made using industry standard Advanced eXtensible Interface (AXI) Connections.

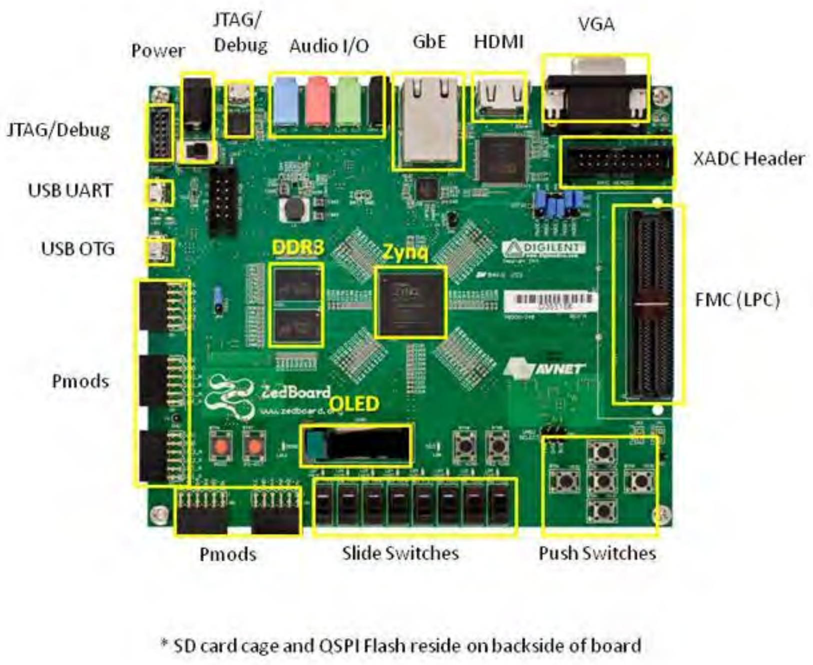


Figure 3.1: Overview of the Zedboard evaluation and development board

For the needs of this project, the following parts were used:

- **Zynq7020 FPGA fabric:**

7 Series PL Equivalent	Artix-7
Programmable Logic Cells	85K Logic Cells
Look-Up Tables (LUTs)	53,200
Flip-Flops	106400
Extensible Block RAM (# 36 Kb Blocks)	4.9Mb(140)
DSP Slices	220

- **DDR memory:**The ZedBoard includes two DDR3 memory components creating a 32-bit interface. The DDR3 is connected to the hard memory controller

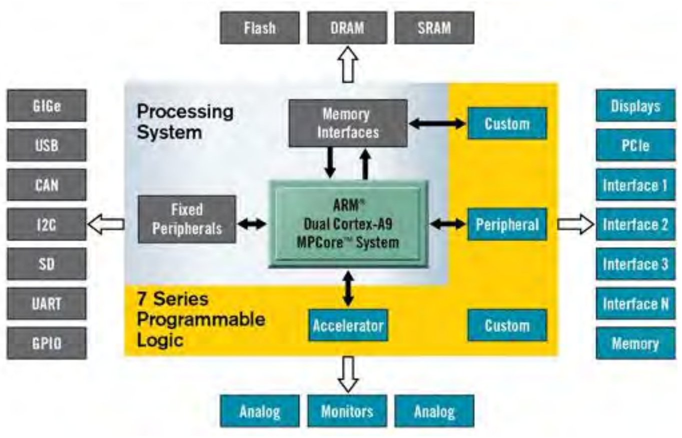


Figure 3.2: System architecture‘s block diagram for Zynq-7000 AP SoC

in the Processor Subsystem (PS).The multi-protocol DDR memory controller is configured for 32-bit wide accesses to a 512 MB address space. The PS incorporates both the DDR controller and the associated PHY, including its own set of dedicated I/Os.

- **SD Card Port:** The Zynq PS SD/SDIO peripheral controls communication with the ZedBoard SD Card The SD card can be used for non-volatile external memory storage as well as booting the Zynq EPP. Note: To use the SD Card, JP6 must be shorted.

- **USB OTG port:** ZedBoard implements one of the two available PS USB OTG interfaces. We use the UART port as a serial communication port between the zedboard and a laptop from which we execute commands to the linux OS on the zedboard. The second OTG port is used to connect the USB webcam to the zedboard.

- **VGA port:**The ZedBoard also allows 12-bit color video output through a VGA connector as shown in .Fig 3.3.

- **Ethernet port:** The ZedBoard implements a 10/100/1000 Ethernet port for network connection, giving us the opportunity to remotely control our system with ssh and sftp.

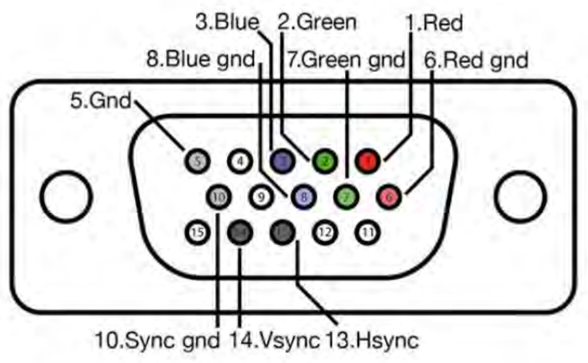


Figure 3.3: Zedboard's VGA port

3.3 FPGA hardware development

Two of the most common methodologies to design hardware circuits are the following:

- **Register-transfer level:** In digital circuit design, register-transfer level (RTL) is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. Register-transfer-level abstraction is used in hardware description languages (HDLs) like Verilog and VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design at the RTL level is typical practice in modern digital design

- **High-level synthesis:** High-level synthesis (HLS) is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. Synthesis begins with a high-level specification of the problem. As shown in Fig. 3.4, the code is analyzed, architecturally constrained, and scheduled to create a register-transfer level (RTL) hardware description language (HDL), which is then in turn commonly synthesized to the gate level by the use of a logic synthesis tool. The goal of HLS is to let hardware designers efficiently build and verify hardware, by giving them better control over optimization of their design architecture, and through the nature of allowing the designer to describe the design at a higher level of abstraction while the tool does the RTL implementation.

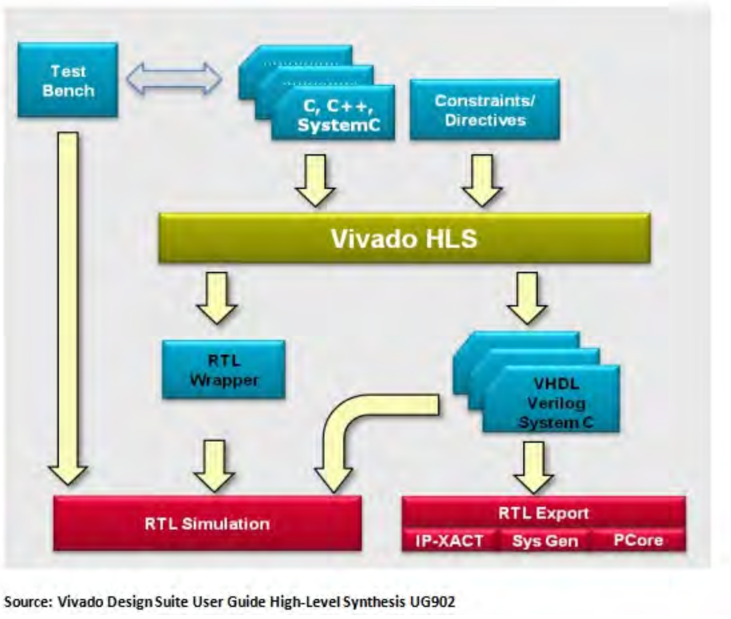


Figure 3.4: Vivado HLS Overview

Fig. 3.5 shows a comparison regarding the design time and the performance achieved in different platforms. It is clear that HLS tools allow the developers to achieve high good standards of performance with only a fraction of the time needed for the development of a hardware project.

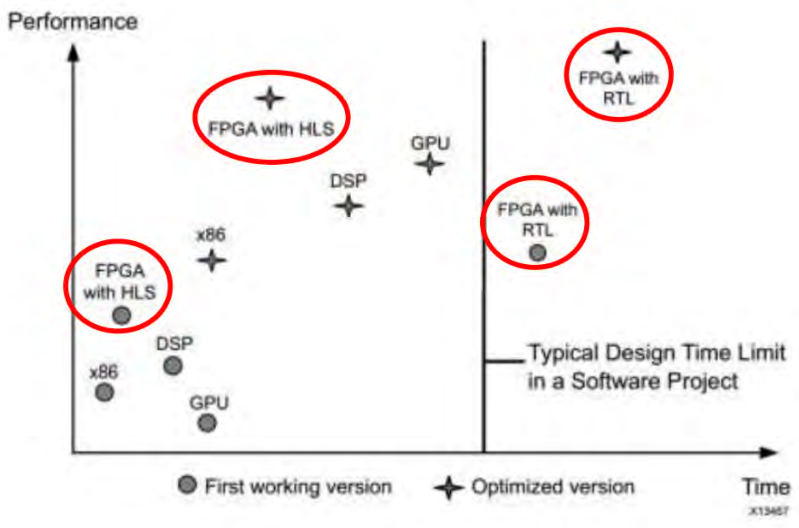


Figure 3.5: Design Time vs Application Performance for different platforms

Chapter 4

HOG Algorithmic Development and Optimization

Chapter 2 presented the original HOG algorithm [2]. However, this algorithm cannot easily achieve real-time performance in an embedded platform, even with hardware acceleration, owing to its high computational complexity. This chapter presents a number of algorithm-level optimizations which collectively aim at approximating the original HOG algorithm, while at the same time, seek to limit the adverse impact of these optimizations on detection accuracy.

Such optimizations are possible because for some applications (like HOG) not all computations and not all data are equally critical, requiring to be performed or maintained at 100% accuracy or correctness. We have the opportunity to trade-off quality of output for significant improvements in performance. For such applications, it may be possible to only approximate the final output (or part of it), rather than computing the exact result. The development outlined in this chapter took place in a laptop with an Intel® Core^{TM} i7-4710MQ CPU running at 2.5Ghz x 8, with 6MB LLC Cache and 8GB DDR3. The operating system is an Ubuntu 16.04 LTS and the compiler used is the gcc-5.4.0 with the -O3 optimization flag enabled. Fig. 4.1 shows the contribution to execution time of each major step of the HOG algorithm.

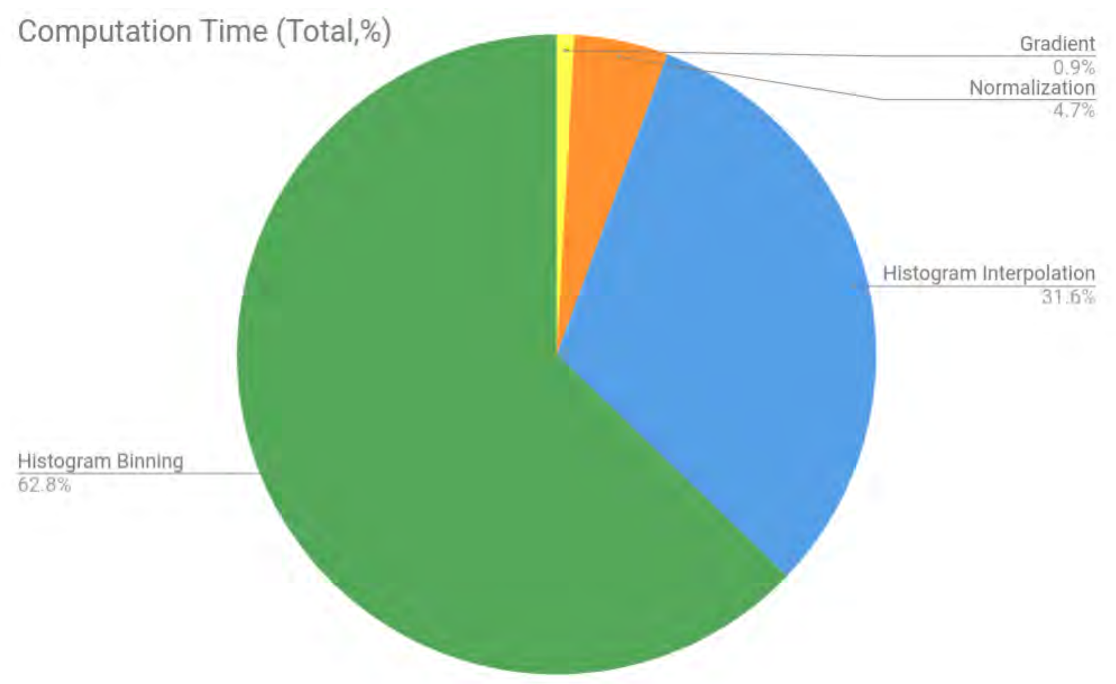


Figure 4.1: HOG profiling in an x86-64 CPU

Fig. 4.1 illustrates the results of a profiling that was performed on the algorithm. It became apparent that the Histogram consumes almost 95% of the total execution time. Consequently all our efforts focused on reducing the cost of this function.

By applying a number of algorithmic approximations, we obtained a collective speedup of 7x (in the x86-64 platform) compared with the original implementation. Some of these approximations were useful mostly for the hardware implementation of HOG. The following sections detail the optimization steps we followed and their impact on detection accuracy.

In table 4.1 it is outlined what are considered to be True Positive (TP), False Positive(FP), True Negative(TN) and False Negative(FN) detections. The accuracy of the system is computed as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.0.1}$$

PREDICTION	REALITY	
	Pedestrian	Non-pedestrian
Pedestrian	True Positive	False Positive
Non-pedestrian	False Negative	True Negative

Table 4.1: Results of Binary classification

As mentioned in Section 2.5, 20% of the Daimler dataset are used to test the accuracy of the system. That means, that there are 3132 pedestrian images and 13488 non-pedestrian ones. Note that after each optimization the output HOG descriptor is altered. Therefore, the SVM model had to be retrained in order to be in sync with the modifications of each step.

4.1 Histogram Binning

First of all, the initial algorithm used the gamma correction technique, a nonlinear operation ($Pixel = Pixel^{\gamma}$), used to encode and decode luminance of a pixel. However, we did not apply this method to the input image since it does not contribute significantly to the overall algorithm accuracy and more specifically according to Dalal & Triggs only by 1% 10^{-4} False Positives Per Window (FPPW).

An important source of complexity (esp. for hardware implementation) is the precise computation of complex functions (such as trigonometric functions or the square root). It should be noted that the modifications of this section did not lead to a performance improvement on the x86 platform. After the vertical(Gx) and horizontal(Gy) gradients are computed for each pixel, their magnitudes are calculated as the absolute value of their difference given by the equation $mag = |Gx - Gy|$ (instead of using the more expensive square root equation).

In addition, the computation of the *arctan* function to determine the bin in which the pixel magnitude will be assigned is replaced by a variation of the method proposed by S.Bauer [7]. We pre-compute (and store as constants) the values of $tan(angle)$ for $angle = 20°$, $angle = 40°$, $angle = 60°$, and $angle = 80°$. The quadrant of each pixel is computed and an angular quantization is performed into

9 evenly spaced orientation bins over 0°-180°as shown in Fig. 4.2. This way the computation of *arctan* is replaced by simple integer multiplications.

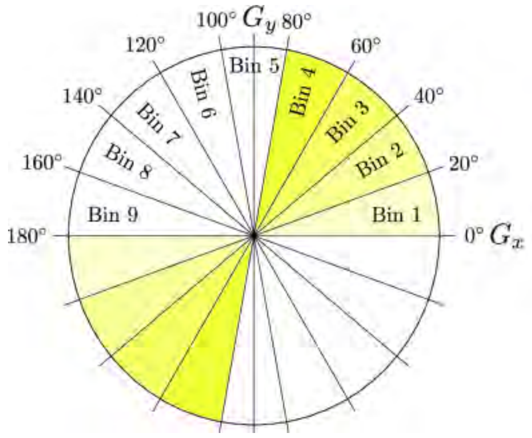


Figure 4.2: Angular quantization into 9 evenly spaced orientation bins over 0°-180°(unsigned gradient).

The angular quantization method is outlined in the following pseudo-code. As it can be seen, the first step is to determine the quadrant where the angle between *Gy* and *Gx* lies. After that, using the pre-computed values of $tan(angle)$, the bin in which the vote function will be added is defined.

Algorithm 2 Histogram Binning

```
procedure Angular_Quantization($Gx, Gy$)
    Input = $Gx, Gy$
        /*Quadrants I & III*/
    if $(Gx > 0\, and\, Gy > 0)$ or $(Gx < 0\, and\, Gy < 0)$ then
        if $(|Gy| < tan(20°) * |Gx|)$ then
            bin = 1
        else if $(|Gy| < tan(40°) * |Gx|)$ then
            bin = 2
        else if $(|Gy| < tan(60°) * |Gx|)$ then
            bin = 3
        else if $(|Gy| < tan(80°) * |Gx|)$ then
            bin = 4
        else
            bin = 5
        end if
        /*Quadrants II & IV*/
    else
        if $(|Gy| < tan(20°) * |Gx|)$ then
            bin = 5
        else if $(|Gy| < tan(40°) * |Gx|)$ then
            bin = 6
        else if $(|Gy| < tan(60°) * |Gx|)$ then
            bin = 7
        else if $(|Gy| < tan(80°) * |Gx|)$ then
            bin = 8
        else
            bin = 9
        end if
    end if
    Return $bin$
end procedure
```

4.2 RGB to Grayscale

First of all, RGB images require 3 times the resources that grayscale ones do. Although this doensn't immediately affect the software implementation, when it comes to an FPGA, resources are of the essence. Furthermore, it also comes with a computational cost, since in order to extract the gradient of each pixel, the maximum of the gradient values of each color should be computed.In addition, as shown by Dalal & Triggs moving from RGB to grayscale colors reduces the accuracy only by 1,5% at 10^{-4} FPPW. Due to those reasons we decided to transition from RGB images to grayscale ones.

4.3 Bin Interpolation

In chapter 2.3 it is shown that in order to reduce aliasing in the initial histogram implementation, gradient vector magnitudes are interpolated tri-linearly between the neighbouring bin centres. For instance, if the angle of a pixel was 22 degrees, had it not been for the interpolation, the whole of its magnitude would have been added to the second bin. Using interpolation, 25% of its magnitude will contribute to the 1st bin and the rest 75 to the second one. In contrast, if a pixel had an angle of 27 degrees, the 75% would be added to the second bin, while 25% of its magnitude would contribute to the third bin.

Our experimentation has shown in Fig. 4.1 that the interpolation is responsible for 31.6% of the total computation time. Therefore, we decided not to interpolate the vote function of each pixel to its neighbouring bins. This action impact on the performance and the accuracy of the system can be seen in Fig. 4.3.

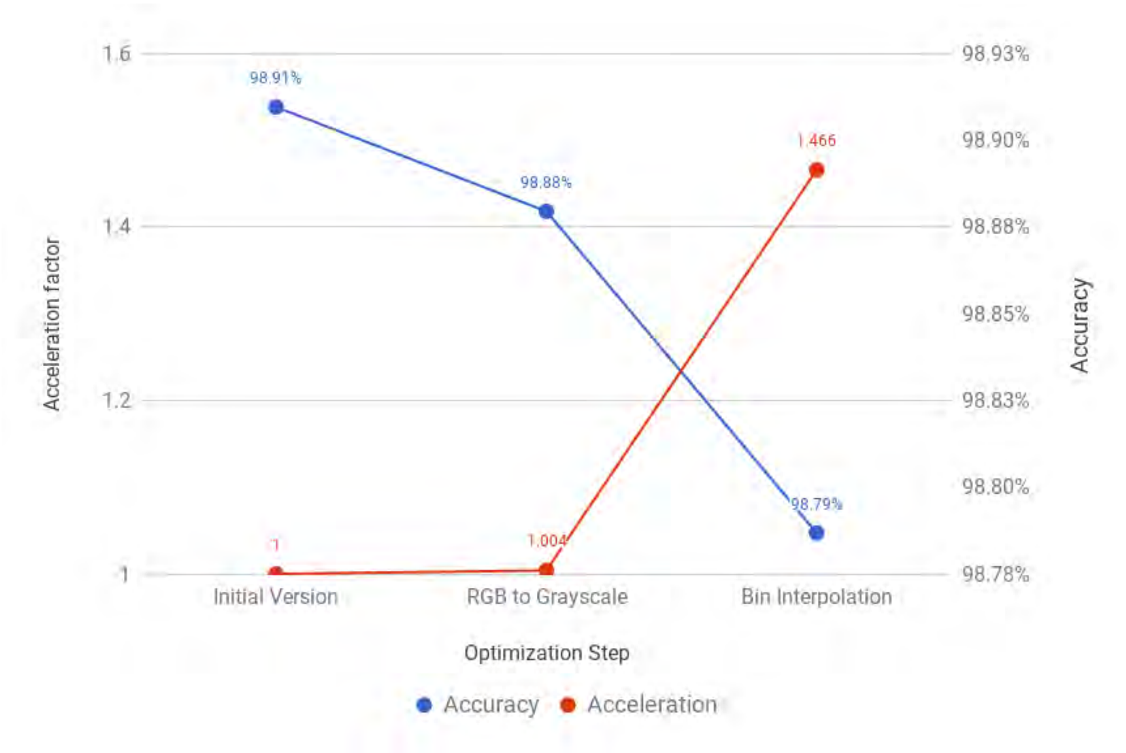


Figure 4.3: Accuracy vs Performance for the first two optimization steps

4.4 Block Stride

Even by removing the interpolation, the Histogram function still dominates the computation time. That's due to the fact that the number of iterations per pixel is much higher than the number of the input, mainly since the algorithm uses overlapping blocks in a detection window. The main focus of this step is to determine the effect of the block stride on both the accuracy and the performance of the algorithm.The results indicate that if we eliminate the overlap between the blocks,while the accuracy of the algorithm is not significantly impacted, we have a serious performance gain.

After this modification, each detection window consists of 4 horizontal blocks and 8 vertical ones, since the block stride increased to 16 pixels (2 x cell size). In addition, the histogram vector size decreased to 1152 (4x8x2x2x9). It is clear that not only was the computational cost reduced, but the memory requirements too, an important aspect when it comes to low-cost FPGA implementation.

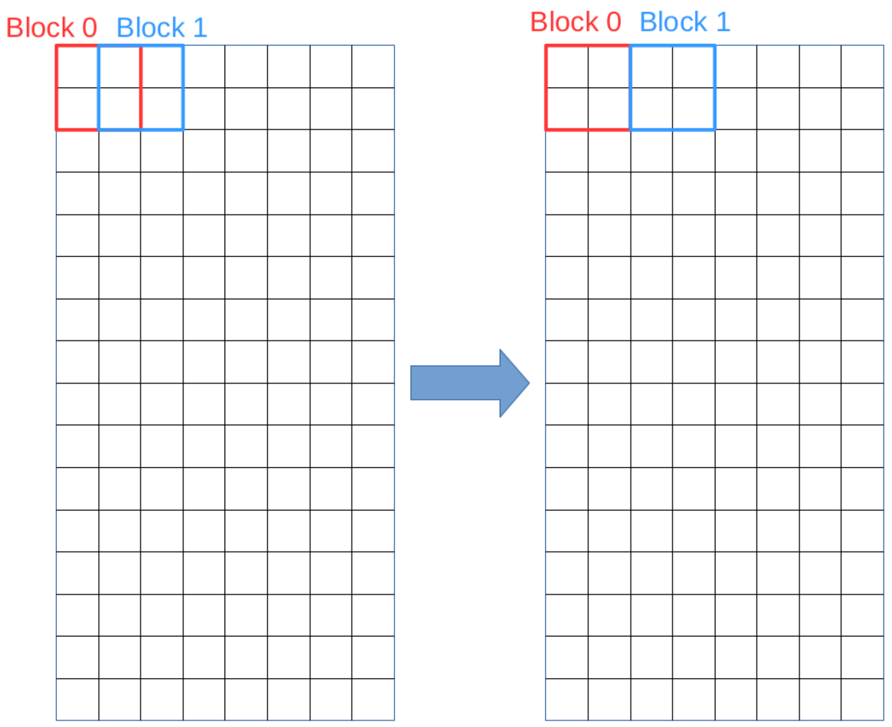


Figure 4.4: Block stride change visualization in a detection window

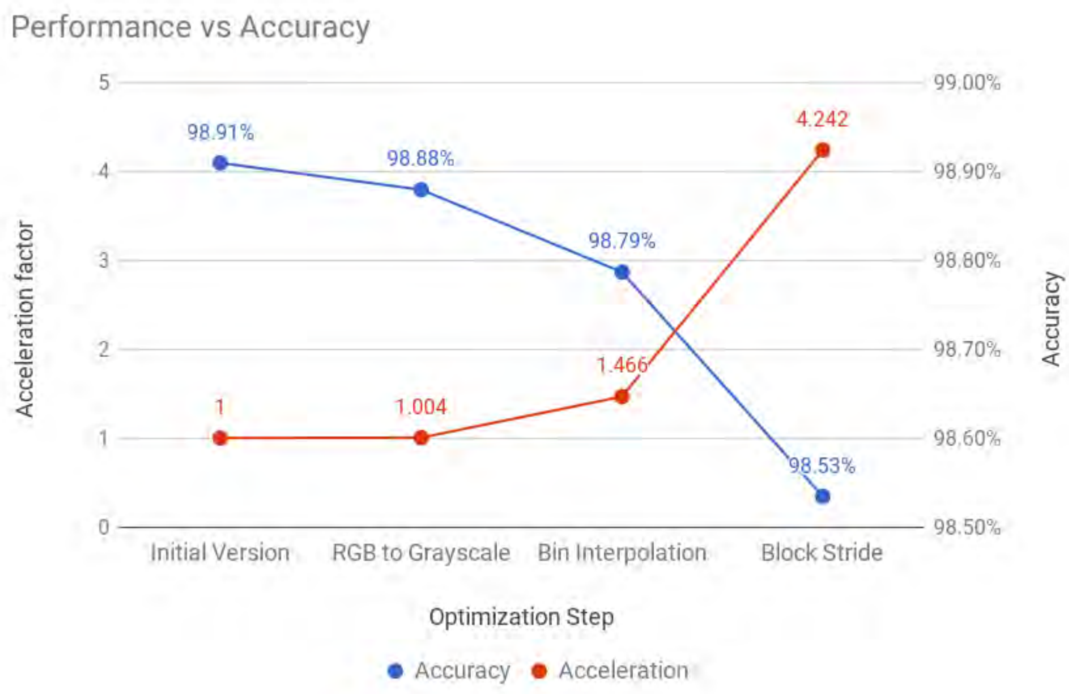


Figure 4.5: Accuracy vs Performance for the first three optimization steps

4.5 Cell column skip

So far, the classification scores between the pedestrian and non-pedestrian samples differ significantly. This is mainly due to the fact that the human shape leads to high magnitude values in specific regions of the detection window. As a result, the HOG descriptor of positive and negative samples follow different patterns. Based on this fact, we wanted to prove that using only alternate pixels, it would not lead to significant miss-classified results but it would rather just decrease the classification score margin between a pedestrian and non-pedestrian sample.

For our implementation only the odd columns of each cell contribute to the HOG descriptor as shown in Fig. 4.6.

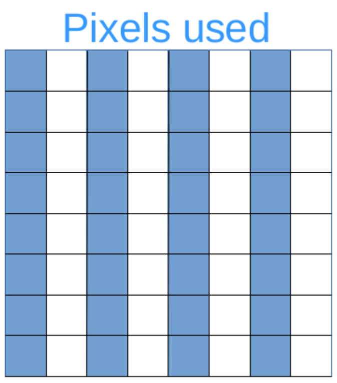


Figure 4.6: Overview of a cell's pixels used

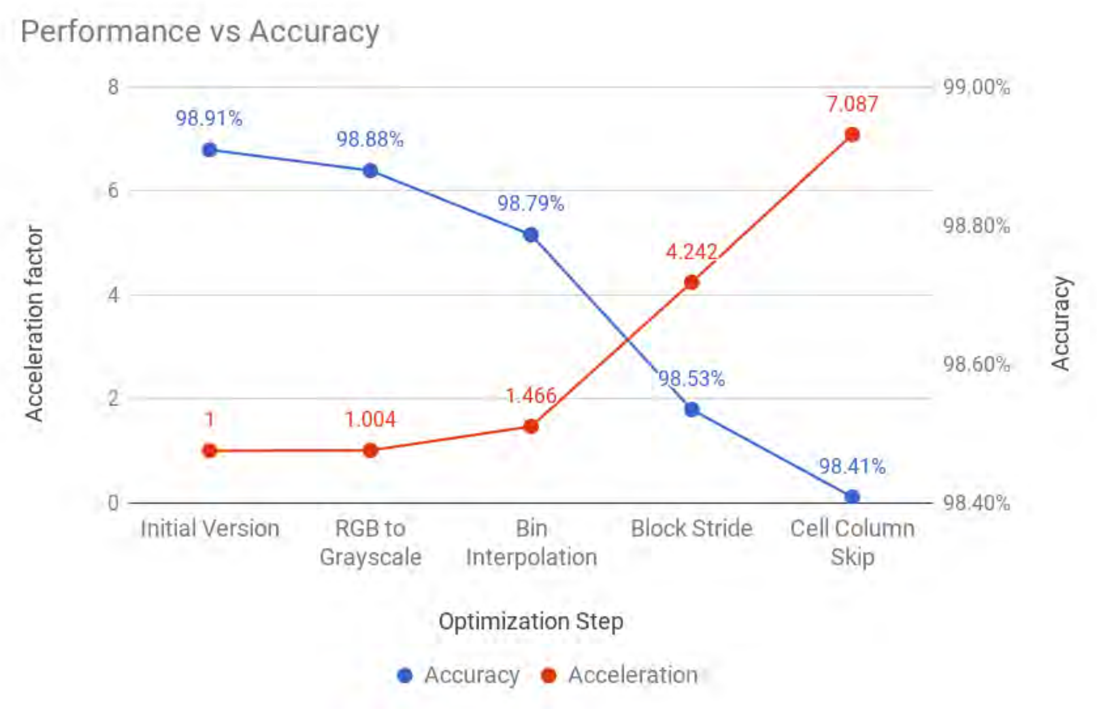


Figure 4.7: Accuracy vs Performance for the first four optimization steps

The effect of this approximation on both the accuracy and the performance of our implementation can be seen in Fig. 4.7

4.6 SVM parameterization

As mentioned in Sec. 2.5, it is clear that the label distributions are 12528 positive (pedestrian) and 53952 negative (non-pedestrian). Moreover, it is also clear that the class balance will be very skewed, with the negative samples outnumbering the positives leading to results as shown in Fig. 4.8 where the true positive accuracy is lower than the true negative accuracy. In addition, for most applications of pedestrian detection, we prefer a larger number of correctly identified pedestrians (true positives), and we, thus, adjust the classifier in favor of detecting pedestrians more easily.

A well-known approach for improving classifier performance in the face of such skewed class distributions is to incorporate the notion of cost-sensitive learning. The SVM^{light} toolkit comes with built-in techniques estimating cost-sensitive models directly. Morik et al. [8] introduced a notion of unsymmetric cost factors in SVM learning. This means associating different cost penalties with false positives and false negatives. After experimenting with accuracy analysis, we set the parameter $cost\_ratio = 2.3$ for the positive samples. Values below 2.3 resulted in a high number of false negatives, while higher ones increased the number of false positives. It should be noted that this value covers our needs for this dataset. A different dataset, may require a lightly different value of *cost_ratio* to optimize true positive detection. Fig. 4.9 shows a large improvement of true positives at the expense of a limited reduction of true negatives.

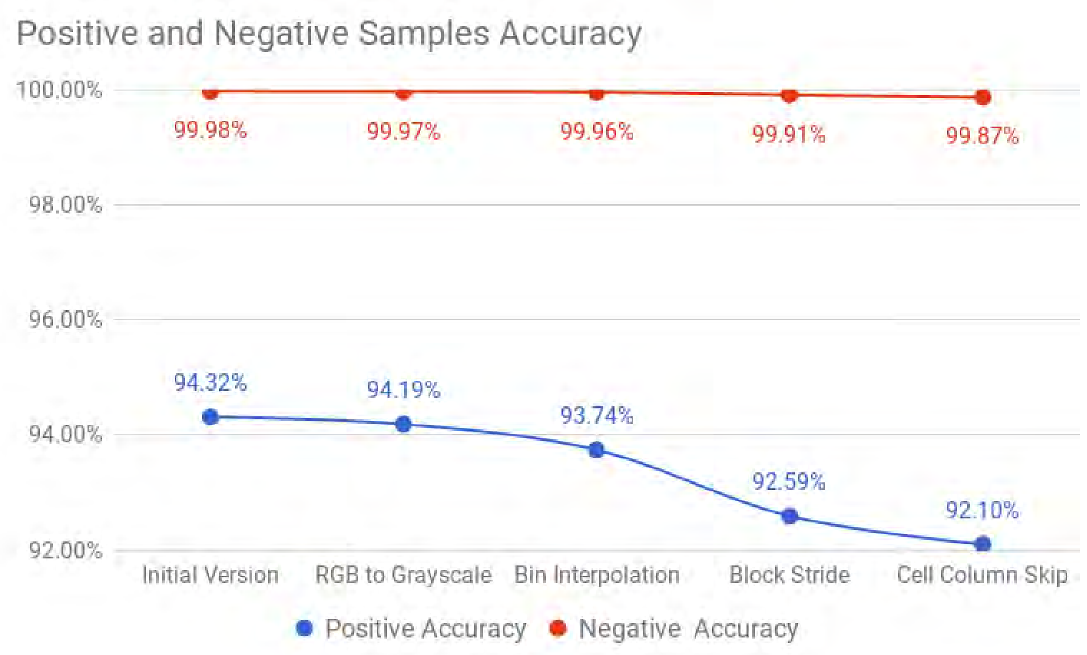


Figure 4.8: Positive and Negative accuracy for each optimization step

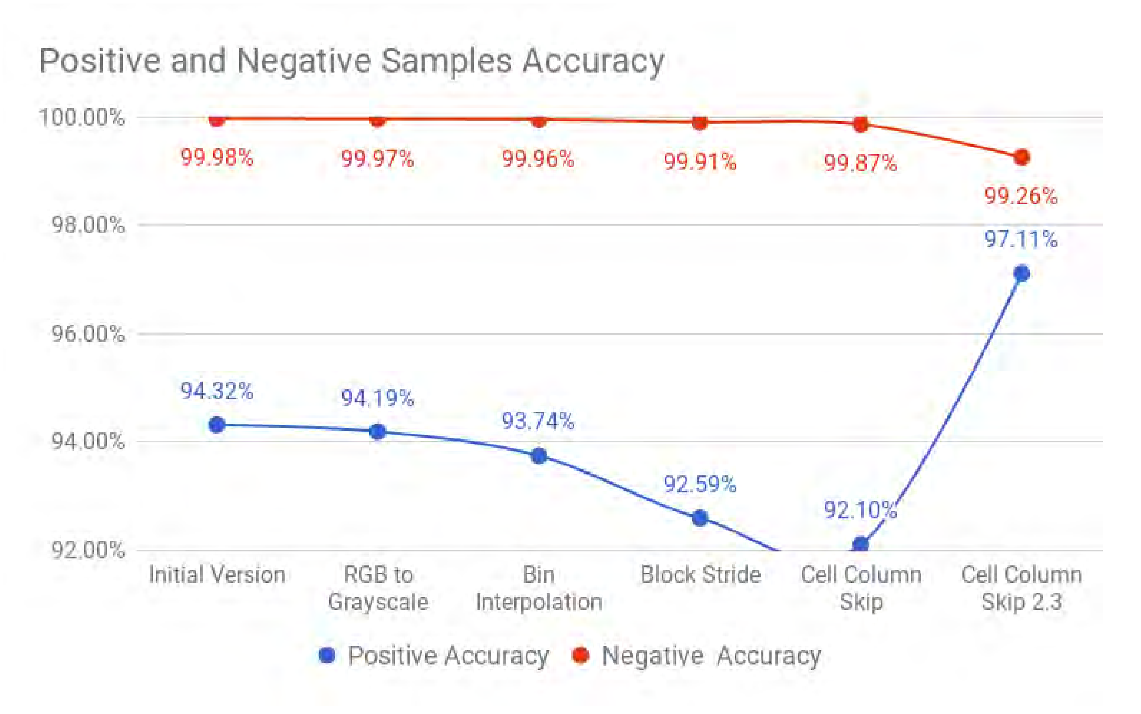


Figure 4.9: Positive and Negative accuracy for each optimization step after the *cost_ratio* parameter modification

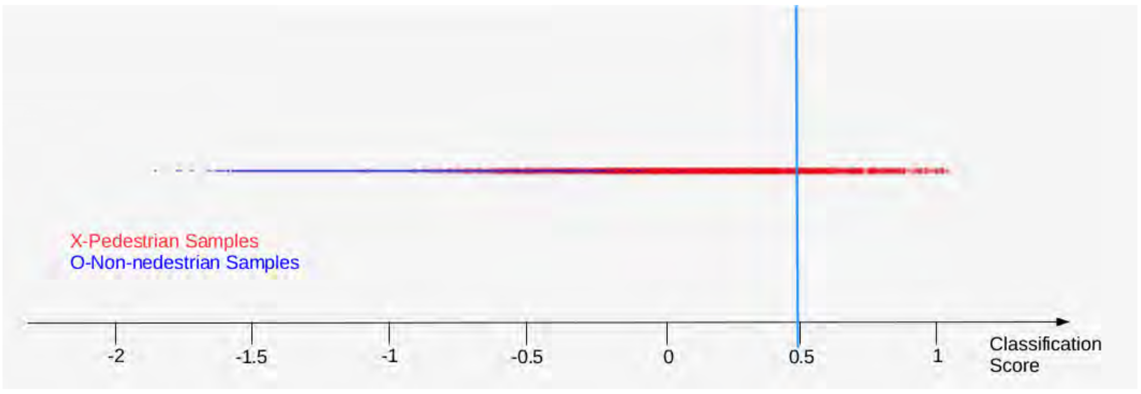


Figure 4.10: Initial SVM's detection performance

A visual representation of the impact of this change on the accuracy of the algorithm, is shown in Figs. 4.10 and 4.11. The graphs depict the plane for the two different configurations. All the samples to the left of the vertical line are classified as non-pedestrians. While initially a large number of pedestrians is misclassified, the improved version greatly improves classification accuracy.

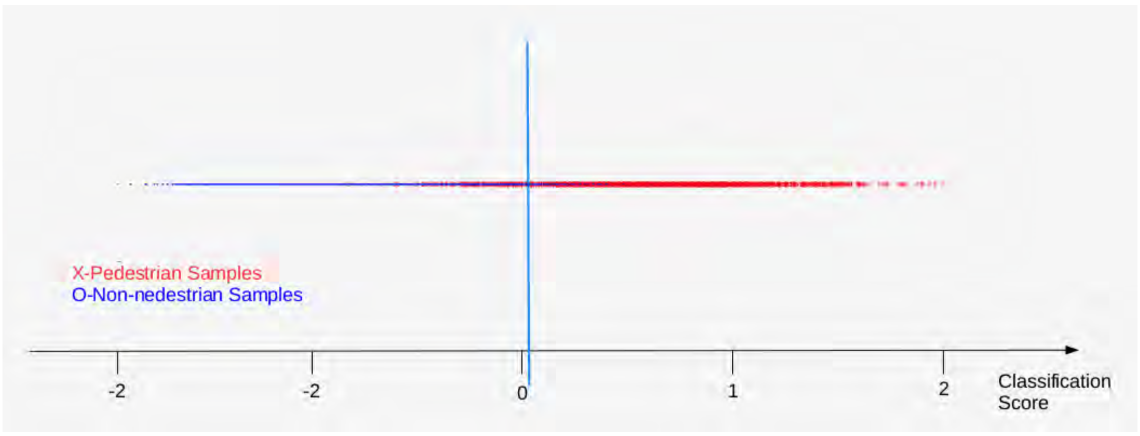


Figure 4.11: SVM's detection performance after the modification

4.7 Histogram Normalization

This algorithmic optimization targets mainly efficient hardware implementations by using a faster and more resource efficient histogram normalization method. Instead of using the L2-Hys norm which is highly expensive cycle and resources wise, a method is proposed that quantizes the values of each block into 8 categories based on their average value. However, this approximation technique has 10 of the 16620 test samples accuracy loss in comparison to the last approximation in Sec. 4.6.

$$normalized\_value = \begin{cases} 0.4, & \text{for } value > 2 \cdot block\_average \\ 0.35, & \text{for } value > 7 \cdot block\_average/4 \\ 0.3, & \text{for } value > 6 \cdot block\_average/4 \\ 0.25, & \text{for } value > 5 \cdot block\_average/4 \\ 0.2, & \text{for } value > block\_average \\ 0.15, & \text{for } value > 3 \cdot block\_average/4 \\ 0.1, & \text{for } value > 2 \cdot block\_average/4 \\ 0.05, & \text{for } value > 1 \cdot block\_average/4 \\ 0, & else \end{cases}$$

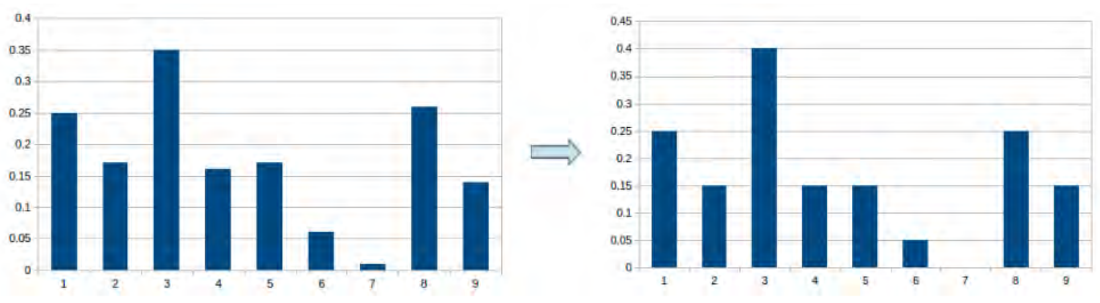


Figure 4.12: Small scale example of the quantized normalization method

Chapter 5

Hardware Implementation

The original software implementation of the HOG algorithm is written in single-threaded C and targets execution on general purpose CPU platforms such as x86 or ARM. The objective of this chapter is to gradually improve the performance of the algorithm in a hardware accelerated platform (Zedboard) by applying a set of optimizations. Some of these optimizations trade-off accuracy and performance as was explained in chapter 4, whereas some optimizations exploit parallelism (at all levels of the algorithm) to improve throughput.

All of the above mentioned parts of the algorithm except for the image downscale and the non-maximal suppression are implemented on the Zedboard's FPGA. More specifically, each accelerator execution computes the HOG descriptor and the classification result for a detection window. Thus, the accelerator is invoked multiple times according to the number of detection windows in an image. In this chapter several hardware implementation and optimization techniques will be presented. For every optimization step, a diagram will be presented which shows the FPGA's utilization regarding several resources as well as a performance metric. The metric used in this case is the number of detection windows that the hardware is able to compute in a second, which is in fact a widely used metric for this type of applications. The synthesis report of the Vivado HLS tool provides the developers with information about the latency in cycles for the implemented hardware. Therefore, the windows per second are computed as:

$$\frac{10^9\ ns}{latency * period\ in\ ns}$$

5.1 Initial Hardware implementation

The initial version of the hardware design is based on the previous optimization steps. However, the programming style in FPGAs should be more stream like. In contrast, CPU programs are generally written as discrete functions for each task. Thus, while on the software there were two separate functions for the gradient and histogram, on the hardware those functions were merged.

The two figures below show the change that was made. Initially, for each window the horizontal and vertical gradients were computed and then stored into two separate arrays. After that point the histogram kernel accessed those arrays in order to compute the HOG descriptor.

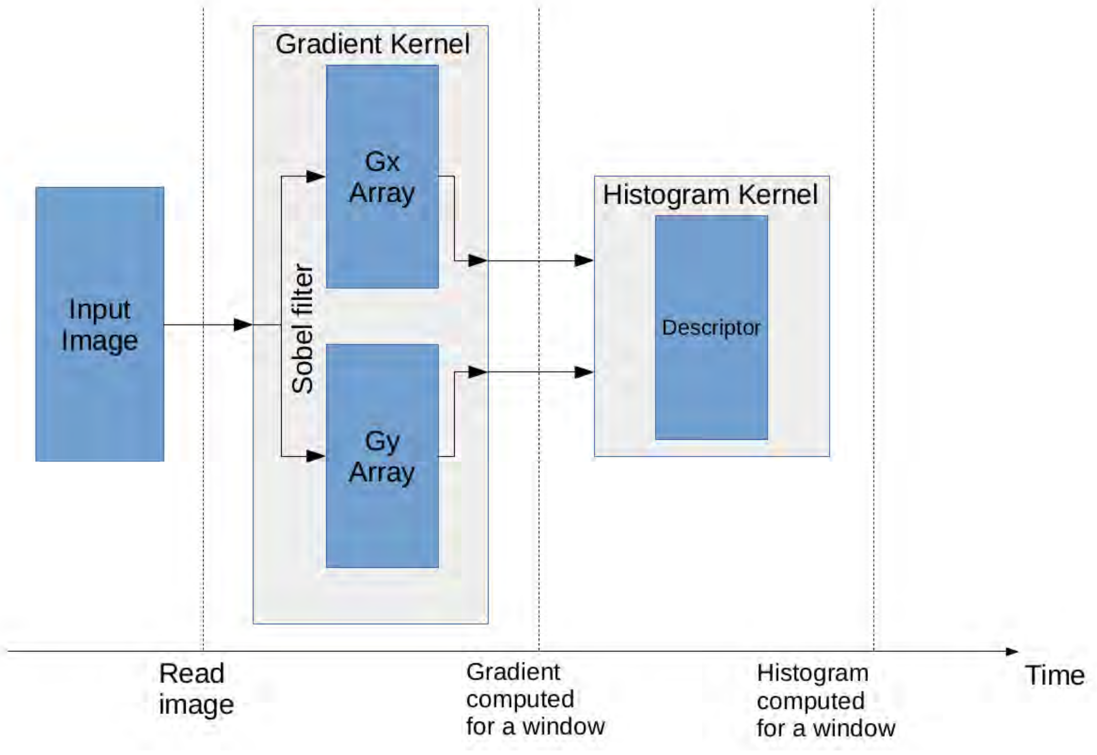


Figure 5.1: Initial Hardware implementation

As it can be seen from Fig. 5.2, the gradients for each are now not stored in arrays but instead the are used in the calculation for the descriptor value for this pixel. As a result, the gradient computation time is completely deducted from the overall computation time leading to a considerably faster implementation.

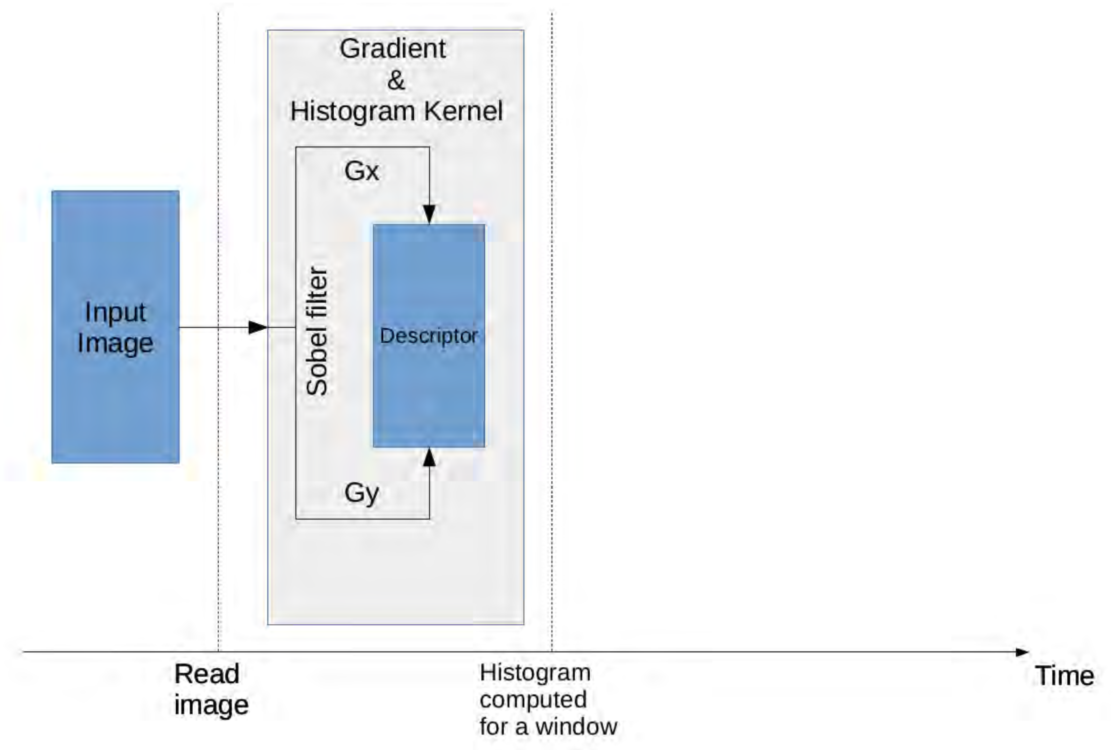


Figure 5.2: Hardware stream-like implementation

In addition, the SVM classification is executed in the FPGA by extracting the linear weights from the trained model into a lookup table. As a result, the prediction is calculated by means of a dot product between the lookup table and the HOG descriptor of each window. This calculation takes great advantage of the pipeline techniques and and the parallelism in the FPGA. For instance, this calculation needs 102.510 us per window to execute on the Zedboard's ARM, while it takes only 0.367 us on the FPGA with a clock of 190 MHz frequency.

Moreover, after careful examination, we used appropriate storage types (BRAM, LUTRam etc.) for all the data types of the implementation. More specifically, three categories of storage devices are used:

- **Dual Port Block-RAMs:** This type is preferred for arrays who are not accessed more than two times in a clock cycle.

- **Partitioned LUT-RAMs:** These are arrays who enable the implemented hardware to access them more that two times in a clock cycle. The use of this type of arrays is extensively described in Section 5.3.

- **Read-Only Block-RAMs:** As it is indicated by their name, this type of arrays consist of constant elements. For instance, the SVM weights, since they are not altered during the execution of the accelerator, are stored in this type of array. In addition, the results of calculations between variables of either limited range or known value are precalculated and stored as look-up tables in this type of arrays.

Furthermore, it is clear that certain HLS pragmas are necessary so as the synthesized hardware to be able to achieve a decent performance. On of the most important HLS pragmas is the **#pragma HLS pipeline** which allows the operations in a loop to be implemented in a concurrent manner as shown in Fig 5.3. In addition, another pragma that was used in certain parts of the code (mostly in array initialization loops) is the **#pragma HLS unroll**, which instances multiple copies of the loop body, so that Vivado HLS can exploit more parallelism among these operations as shown in Fig 5.4 and in Fig 5.5.

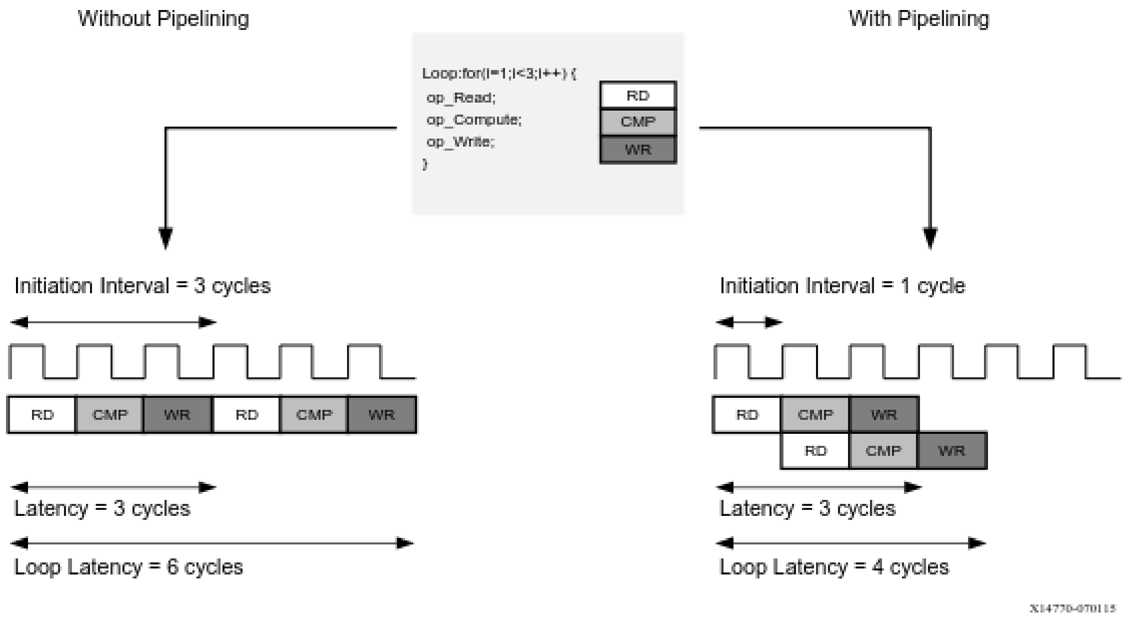
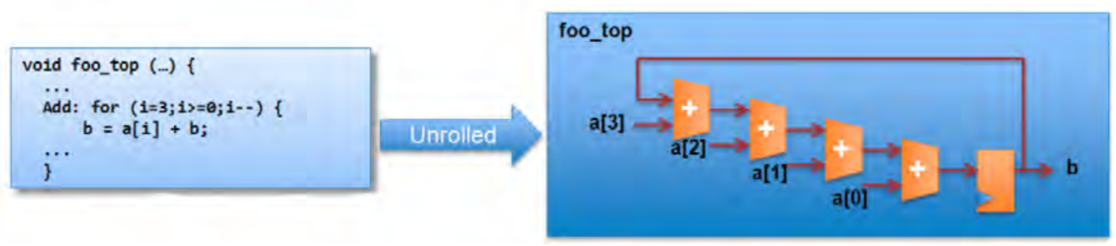


Figure 5.3: Impact of the pipeline pragma on a loop's latency



```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
       b = a[i] + b;
  ...
}
```

Figure 5.4: Unrolled loop's architecture

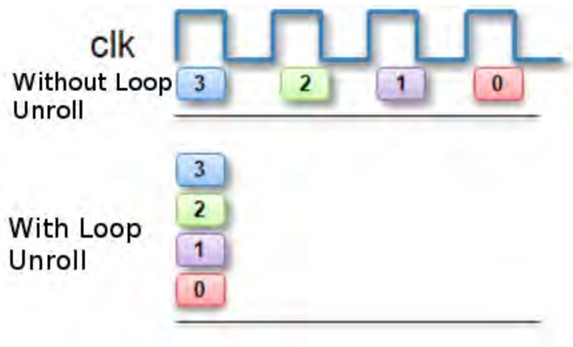


Figure 5.5: Impact of the unroll pragma on a loop's latency

5.2 Fixed point arithmetic

One of the most used optimization techniques is the fixed point arithmetic. In this case, prior to the computation the values are shifted by 10 bits to the left and when the computation is over they are shifted back to the right. There was only a slight loss of accuracy and more specifically, from the test-set of 16620 images, only 50 more in comparison to the float point computation were misclassified since this technique distorts the value after the fourth decimal point. However, the benefits in both latency and area are significant.

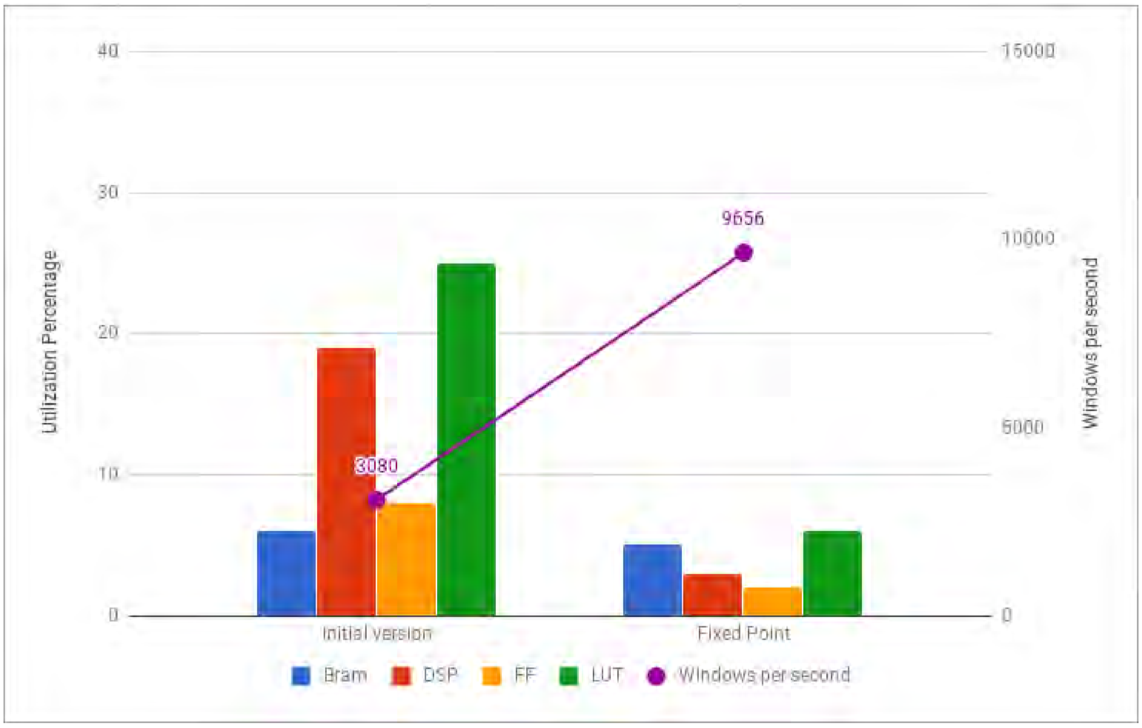


Figure 5.6: Resources vs Latency for the first optimization step

5.3 Arbitrary precision and throughput

Arbitrary precision data types were used to specify the bit length of each variable since they have several advantages. Most importantly, these types allow variables to be defined as any arbitrary bit-width (6-bit, 12-bit, 143-bit etc.), while standard C data types allow variables to be modeled on 8-bit boundaries (8-bit, 16-bit, 32-bit, etc.).This allows the C code to accurately model, and be synthesized to, the exact bit-widths required in hardware. For example, this ensures that if a multiplication operation only requires 18-bits, the designer is not forced to use a standard 32-bit C data-type, which would force the multiplier to be implemented with more than one DSP48 macro in the FPGA.

This change didn't have any benefits performance wise, but reduced the resources needed as far as DPSs and LUT are concerned which is important in case multiple accelerators are to instantiated in a single block design.

In addition, some optimizations were performed with the aim of increasing the local memory bandwidth. The Vivado HLS tool provides some pragmas specifically for this cause. For instance, the **#pragma HLS array_partition** which partitions the arrays as shown in Fig 5.7 into smaller ones, effectively increasing the number of load/store ports so as to increase the local memory throughput.

Vivado HLS provides three types of array partitioning:

- **block:**The original array is split into equally sized blocks of consecutive elements of the original array.
- **cyclic:**The original array is split into equally sized blocks interleaving the elements of the original array.
- **complete:**The default operation is to split the array into its individual elements. This corresponds to implementing an array as a collection of registers rather than as a memory.

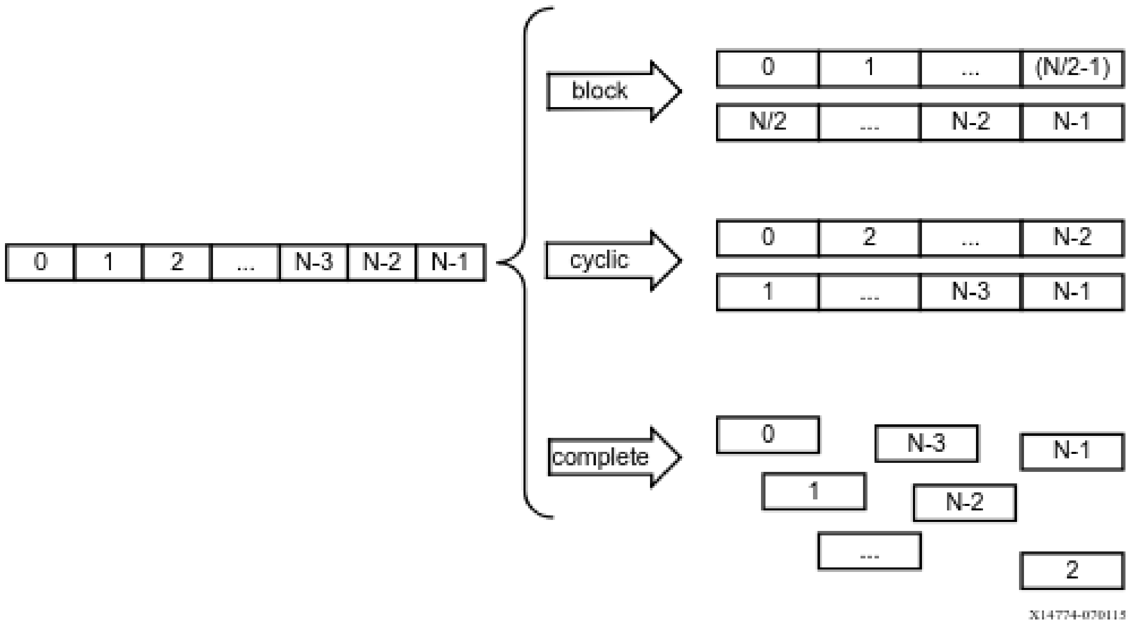


Figure 5.7: Overview of the HLS partition pragma

Due to the specific memory patterns of this algorithm, the complete option is used. More specifically, this pragma is used for the input image memory so as to be able to be accessed 4 times simultaneously. For example, in order to calculate the gradient of the first pixel one would need to access the pixels painted in yellow as shown in Fig 5.8:

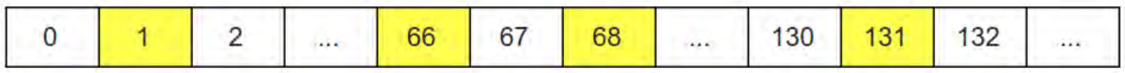


Figure 5.8: Initial input image array

An overview of the partitioned array and its access pettern can be seen in Fig 5.9.

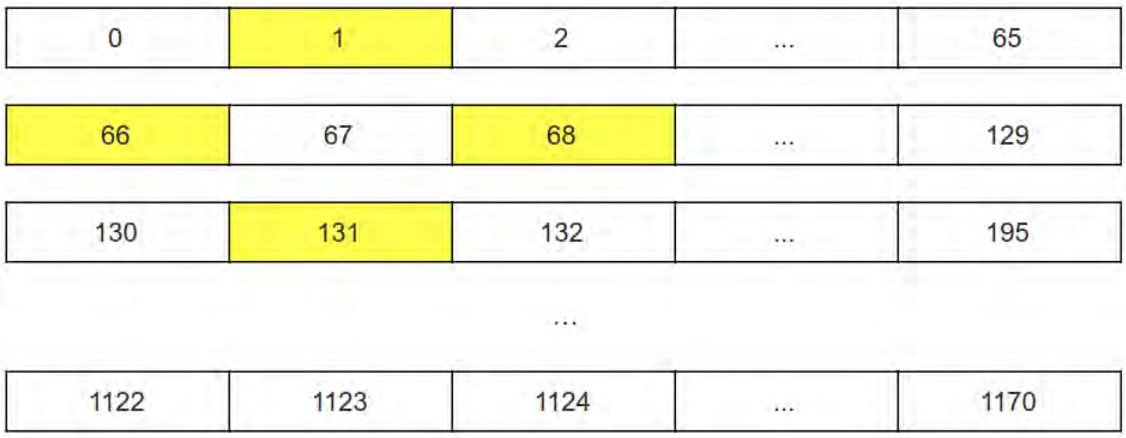


Figure 5.9: Partitioned image array

As a result, the iteration interval for gradient of each pixel is reduced at 1. Nonetheless, the lookup tables utilization is considerably increased due to the more logic needed for this action. As it can be seen from the graph, the block ram used is reduced due to the fact that the input array is now stored in 18 distributed LUT Rams.

The other HLS pragma used to increase the memory throughput is the **#pragma HLS dependence** which is used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals). Under certain circumstances, such as variable dependent array indexing, or when an external requirement needs to be enforced (for example, two inputs are never the same index), the dependence analysis might be too conservative. This pragma allows you to explicitly specify the dependence and resolve a false dependence.

In this project, the dependence pragma is used on the variable in which the histogram bin is accumulated, in order to notify the synthesizer that actually no dependence exists, taking full advantage of the pipeline's capabilities. As a consequence, the iteration interval of this action is reduced from 2 to 1 cycles. Consequently, the windows per second metric increases while there is a small difference in the resources utilization.

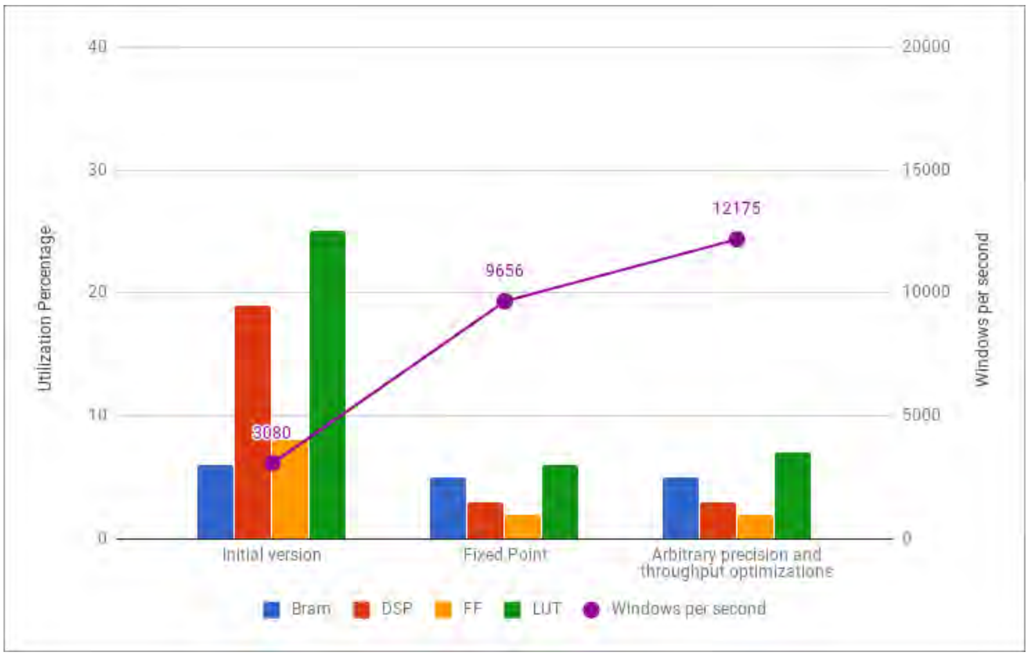


Figure 5.10: Resources vs Latency for the second optimization step

5.4 Multiple accelerators

The previous optimization steps aimed to increase the memory throughput of the accelerator and if possible reduce the hardware resources required. From this point, since there are enough resources available, more than one accelerators are instantiated. Particularly, due to the fact a detection window consists of 8 vertical and 4 horizontal blocks, the number of the instantiated accelerators is 8 as seen in Fig. 5.11. Therefore, each accelerator computes 1/8 of each detection window which means 4 horizontal blocks of it. After that, each accelerator computes the 1/8 of the window prediction and finally their classification values are added to have the overall result.

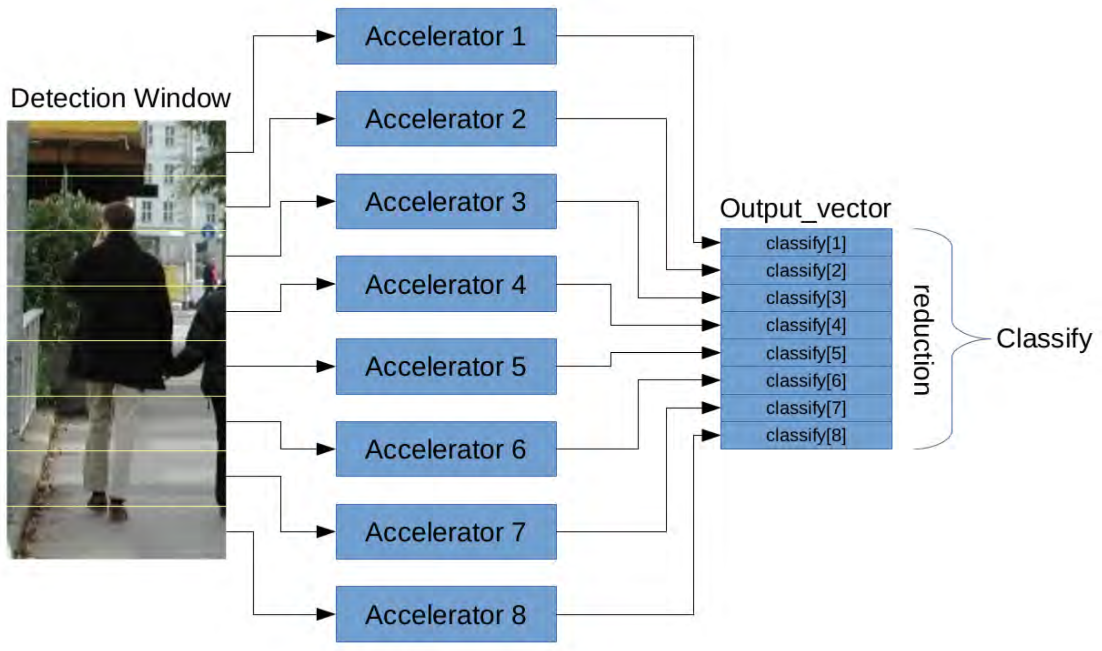


Figure 5.11: Abstract overview of the implemented hardware input & output

In Fig. 5.12 an overview of a single accelerator's flow is presented.

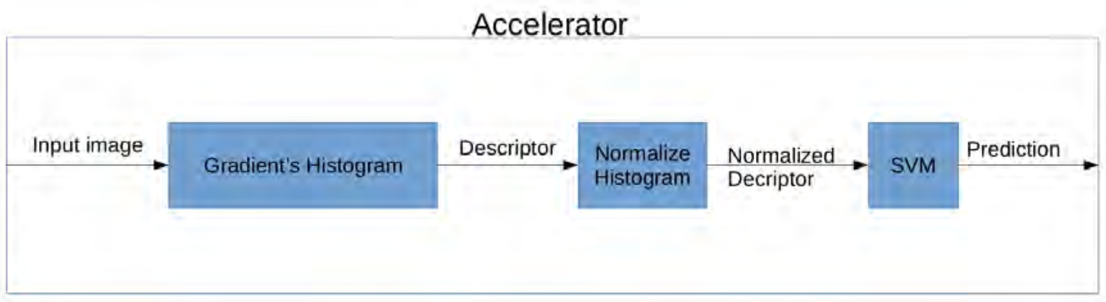


Figure 5.12: Abstract overview of the accelerator's flow

It is obvious that by using multiple accelerators, the utilization overall linearly increases while the latency decreases by the same factor, as it is evident in Fig 5.13.

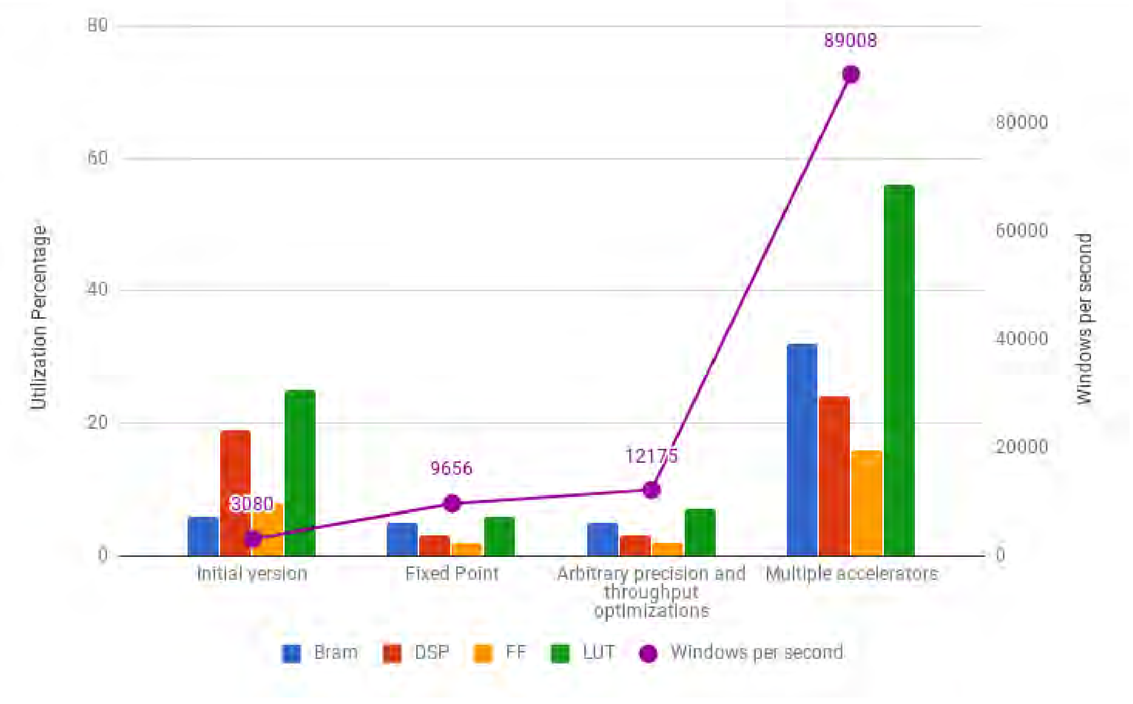


Figure 5.13: Resources vs Latency for the third optimization step

5.5 Parallel data processing

The next step would be to instantiate more accelerators in our implementation. A way to improve the throughput would be for each accelerator to compute two rather than four horizontal blocks. However, as it is clear from Fig. 5.13, the available LUTs wouldn't suffice. Therefore, we focused on increasing the accelerator's throughput by computing data in parallel.

First of all, the input image is stored into two separate arrays. As a result, two instances of Histogram and Normalize functions can execute in parallel. This means that each one of them computes the descriptor of two of the four horizontal blocks of the detection window. Finally the execution time of the SVM function is also reduced by a factor of 2, since it can compute the classification results of the two normalized HOG descriptors in parallel. As shown in Fig. 5.14, although the Read_image still consumes the same amount of time, the computation part of the accelerator(Histogram,Normalize,SVM) requires now half the cycles.

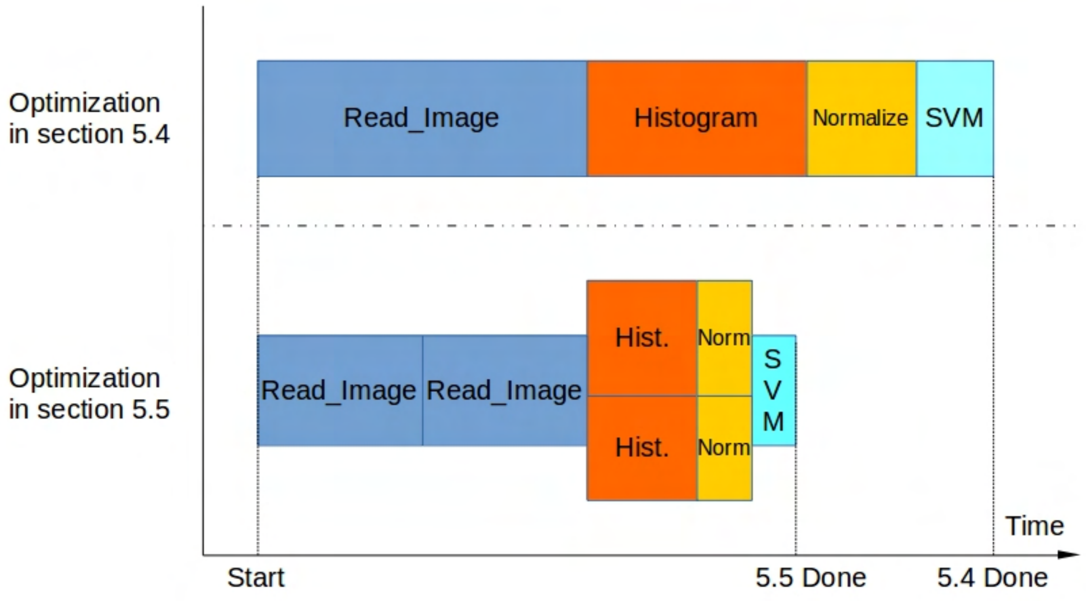


Figure 5.14: Parallel data processing

Fig. 5.15 outlines the performance and the resource's utilization after applying the Parallel data processing inside the accelerator.

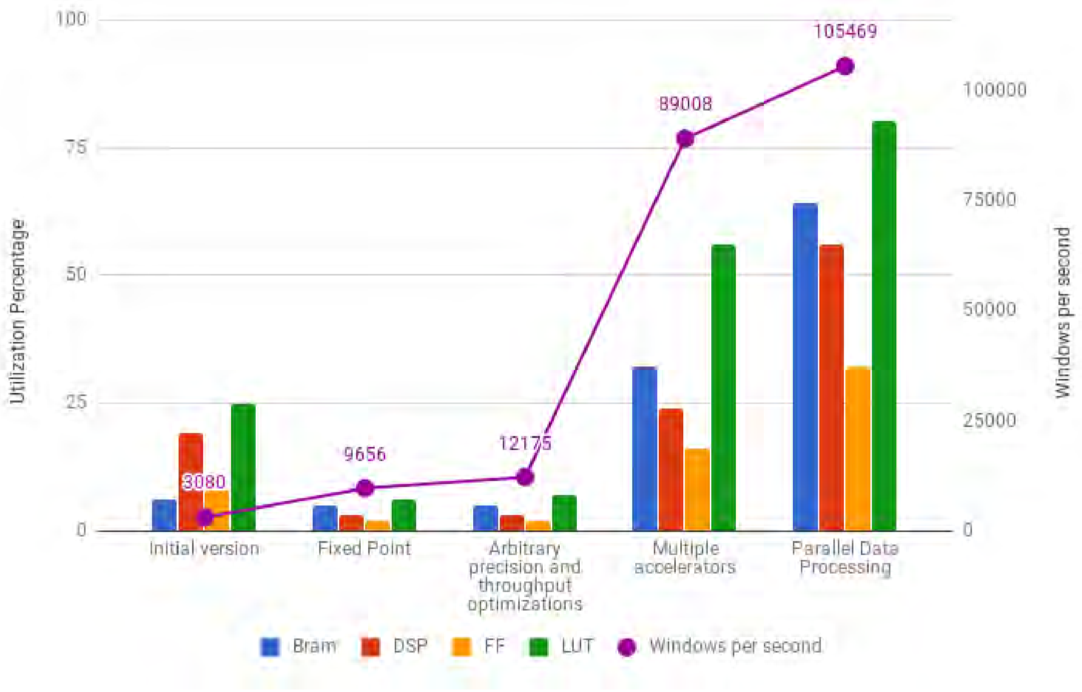


Figure 5.15: Resources vs Latency for the final optimization step

5.6 Block design generation

After the C++ code is synthesized, the generated RTL is exported and 8 accelerators are instantiated into the Vivado block design. Here is how an instance of the HOG accelerator looks like in Fig. 5.16:

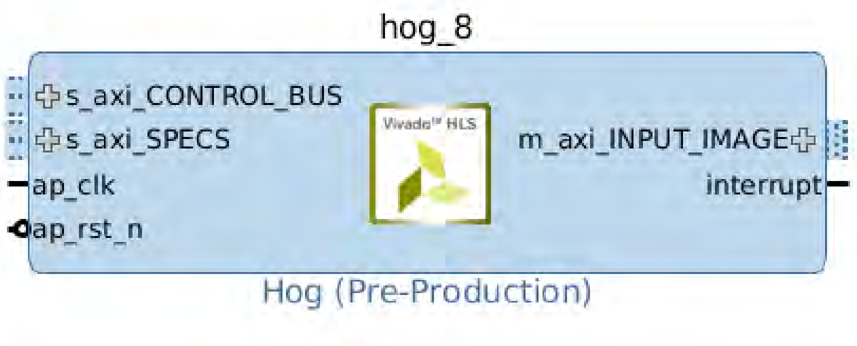


Figure 5.16: Overview of the accelerator's IP ports

The port *s_axi_CONTROL_BUS* is an AXI-4 Lite wrapper of the ap protocol control signals of the accelerator so it can be controlled and monitored form the embedded application. Obviously clock and reset signals are needed for the accelerator to function. On the upper left corner is a AXI-4 Lite port *s_axi_SPECS* from which the accelerator reads various information such as the position of the detection window in the input image and also serves as the accelerator's output where the classification result is stored. The next port *m_axi_INPUT_IMAGE* is the one through which the accelerator accesses the memory where the input image is stored. The full AXI protocol interface is used, since it can transfer data in bursts, thus providing a higher throughput than the Lite. Finally, the interrupt port is not used.

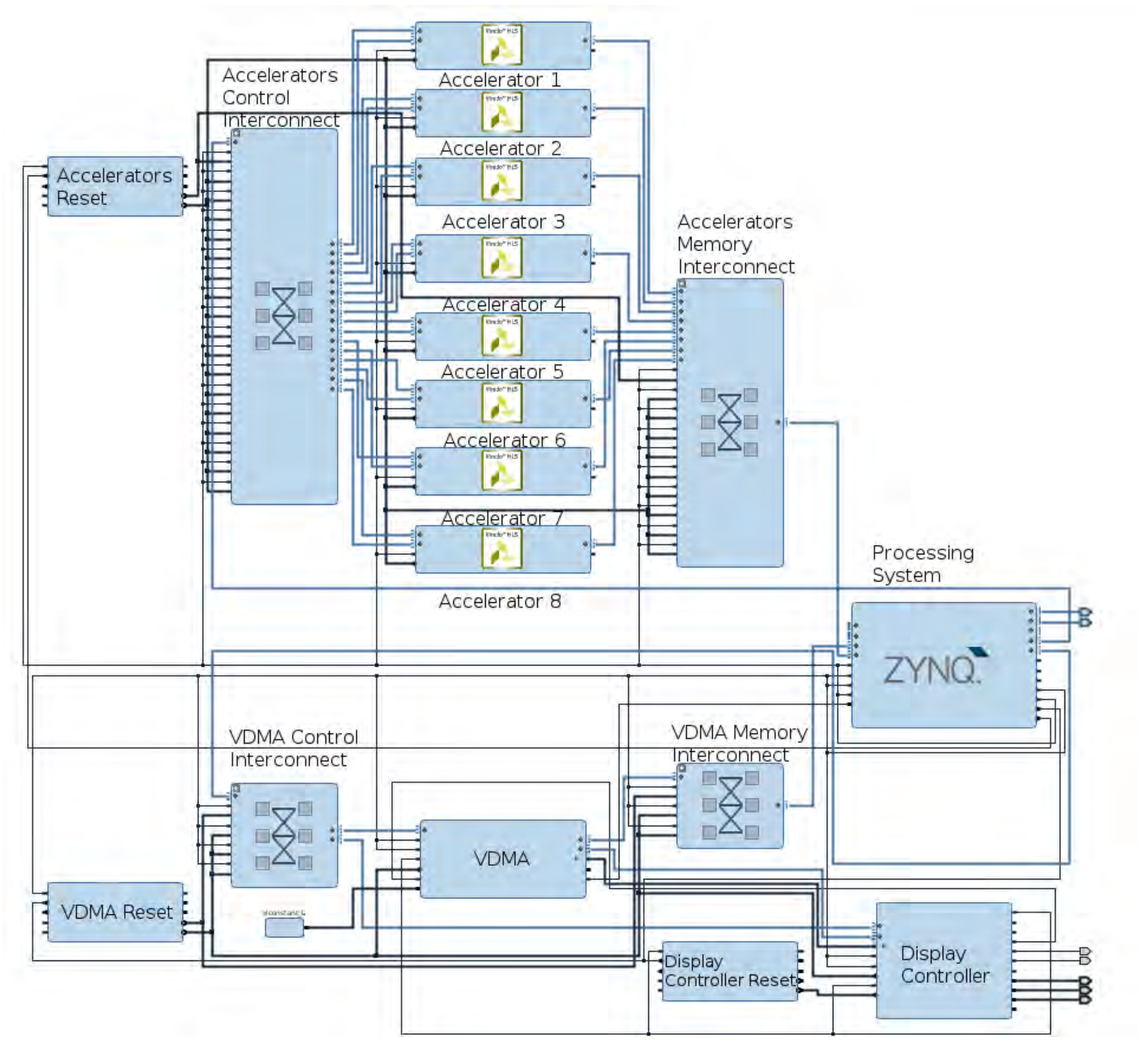


Figure 5.17: Overview of the block design

One AXI interconnect for the accelerators data is instantiated, since it can hold up to 16 slave interfaces, which is exactly the number that was needed for this implementation and it is connected to a High Performance(HP) port of the processing system. Another memory interconnect is used for the control of each accelerator. The output of the system is projected from a VGA monitor connected to the Zedboard. Hence, a VGA controller was needed. In this design a Display controller by Digilent is used, since it supports the VGA protocol. Basically, the display controller accesses the output image data through a Video DMA and renders the image to a VGA monitor. The system supports up to 4 separate clocks. In this case, a 180 MHz clock is used for the accelerators, while a 100 MHz one is used for the VGA driver. After the design is validated, it is implemented into the PL fabric and the bitstream file is exported.

Chapter 6

System Software Integration

6.1 Embedded Linux

An embedded operating system is an operating system for embedded computer systems. This type of operating system is typically designed to be resource-efficient and reliable. Resource efficiency comes at the cost of losing some functionality or granularity that larger computer operating systems provide, including functions which may not be used by the specialized applications they run.

In this case Petalinux OS was preferred since Xilinx provides a tool chain to generate Linux kernel images, root file systems and kernel modules for ZYNQ like embedded systems with programmable hardware(for different hardware designs in the FPGA section). Using PetaLinux tool chain, we can easily build kernel and modules for ZYNQ PS without using separate cross compilation tools. PetaLinux tool can generate U-Boot files, First Stage Boot Loader(FSBL) and BOOT.BIN for a specific hardware design. Same things can be done using Xilinx SDK. For this project a Petalinux extracted linux kernel with an Ubuntu 16.04 rootfs is used.

Alongside the bitstream file, the Vivado tool exports also a Hardware Description File (hdf) which is a Xilinx proprietary file format and contains information about the block design. More specifically, memory map information about the implemented hardware is provided, as well as internal connectivity information (incl interrupts, clocks, . . .) and external ports too. This is the initial information that is required in order to build a Petalinux image. Additionally, the Petalinux kernel is modified so as to be able to boot from the SD card and to support a webcam. The latter is done by enabling usb and uvc modules of the kernel.

As the last step, the operating system needs to be able to access the implemented hardware. This is done by providing the Linux device tree with information about, the addresses and the compatibility of the programmable logic(PL). For the needs of this project the accelerators needed to be configured and controlled from the user space. Linux provides a standard called UIO (User I/O) framework for developing user-space-based device drivers. The UIO framework defines a small kernel-space component that performs two key tasks:

- Indicate device memory regions to user space.
- Register for device interrupts and provide interrupt indication to user space.

The kernel-space UIO component then exposes the device via a set of sysfs entries like /dev/uioXX. The user-space component searches for these entries, reads the device address ranges and maps them to user space memory. The user-space component can perform all device-management tasks including I/O from the device.

6.2 System Overview

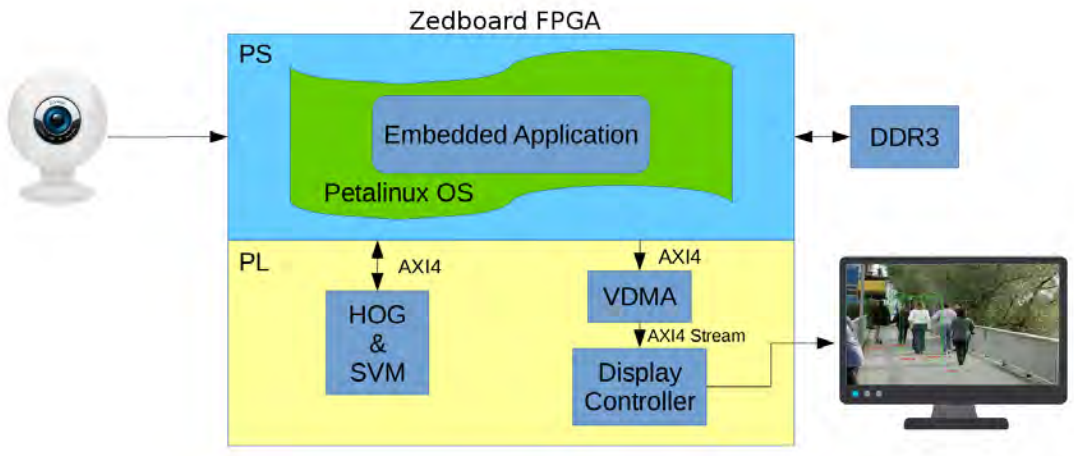


Figure 6.1: System Overview

The image above is an abstract representation of the implemented system on the Zedboard FPGA. As it can be seen, the input is captured from the webcam connected to the FPGA and the output is projected to a VGA monitor. An embedded application is being executed on the Petalinux operating system running on the processing system of the board. Finally, all the accelerators and the display controller which are implemented on the programmable logic are being controlled from the embedded application.

6.3 Embedded Application

All the image processing is performed using the OpenCV framework, which is a library of programming functions mainly aimed at real-time computer vision applications. More specifically, it is used for the frame grab, the conversion from RGB colorspace to grayscale as well as for the image downscale.

In order for the hardware to be able to access the data created from the software, their physical address is needed. However, in user-space only virtual addresses can be handled. Therefore, a mapping from a physical to a virtual memory was necessary. This is performed via the mmap function.

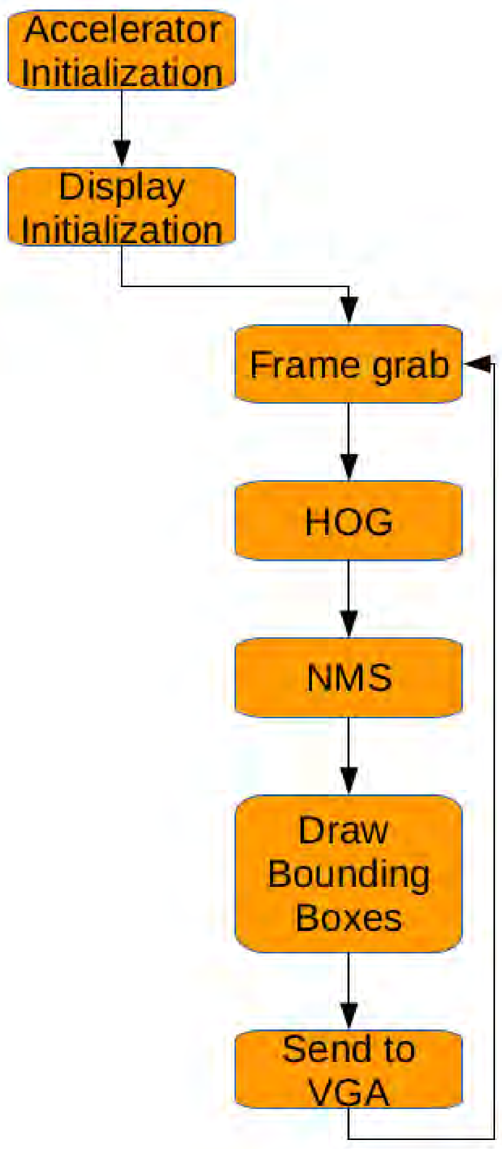


Figure 6.2: Application flow

Accelerator Initialization: The implemented hardware accelerators can be initialized and controlled by the automatic generated drivers from the Vivado HLS.

Display Initialization: Here, parameters of the VGA driver are initialized, as well as the physical address of the output image.

Frame grab: An frame is captured from the webcam connected to the Zedboard and stored into the internal memory. In addition, the image is converted to grayscale and a padding of 8 pixels per side is added.

HOG: Two functions are executed here. The first is the resize function which downscales the input image and the HOG controller which executes the accelerators and computes the classification result. These two functions run in parallel using a double buffering technique with the OpenMP framework taking full advantage of the two available cores on the ARM. That means that while the first CPU controls and monitors the hardware accelerators, the other one prepares the next downscaled image and vice versa.

NMS: The non maximal suppression is computed here, which merges the bounding boxes with an overlap of over 70%.

Draw bounding boxes: The detection windows alongside their scores and detection scale are printed on the output image.

Send to VGA: This is the last step of the application where the output image is rendered on the VGA monitor.

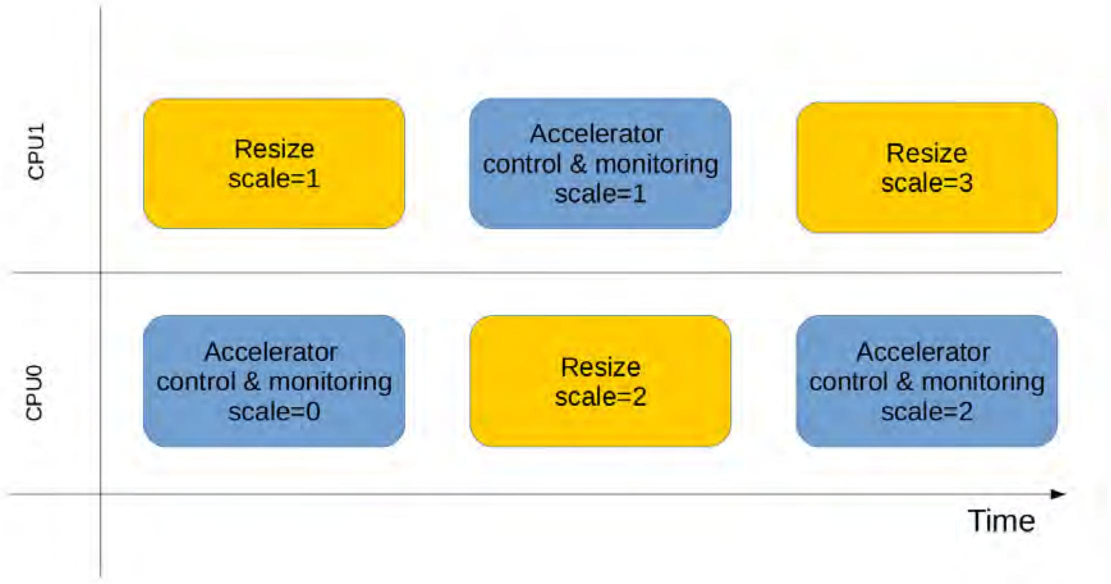

CPU1
Resize scale=1
Accelerator control & monitoring scale=1
Resize scale=3
CPU0
Accelerator control & monitoring scale=0
Resize scale=2
Accelerator control & monitoring scale=2
Time

Figure 6.3: Time representation of the double buffering

6.4 Performance Comparison

A profiling is performed in order to provide a performance analysis on each of the software and hardware implementations on our embedded platform. For each implementation we measured the application's computational time for the HOG descriptor extraction as well as the SVM evaluation for one frame for 12 different scales. It should be stated that all the implementations are compiled with the gcc compiler with the -O3 optimization flags enabled. The results of the profiling are shown in Table 6.1. The initial software implementation is the version of the HOG algorithm without any software optimizations. The approximation and software optimization techniques as described in Section 4.7 are implemented in the optimized software version of the software. In addition, the OpenMP framework is used to take advantage of the parallelism that is able to be performed on the two available ARM cores. Finally, the hardware implementation involves all the algorithmic and hardware optimizations as described in Chapter 5.

Initial Implementation	Optimized Software	Hardware Implementation
30s	1.450s	0.080s

Table 6.1: Performance comparison

It is clear that the approximation techniques performed on the software yield a more that 20x better performance than the initial version. However, with hardware optimization techniques on the FPGA, the system was able to perform in real-time achieving 384x acceleration from the initial implementation.

Chapter 7

Related Work

FPGA Implementation of a HOG-based Pedestrian Recognition System.
Sebastian Bauer, Ulrich Brunsmann, Stefan Schlotterbeck-Macht

Their proposed system is based on an FPGA-CPU-GPU network. The system consists of two FPGA boards. The former (Xilinx Spartan 3 XC3S 2000) provides data transfer interfaces to the camera and to the host (PCIe), the latter(Spartan 3 XC3S 4000) and on-board memory is available for for the computation of the HOG descriptor. The descriptor normalization step is then performed on the CPU, being the central entity of our system. The SVM-based sliding-window evaluation is currently running on a publicly available GPGPU solution [cuSVM].

The algorithm is implemented as proposed by Dalal&Triggs. The only exceptions are that the vote function of each gradient is not interpolated over neighbouring bins, as well as the bin index calculation is not extracted using the arctan function but by performing. Angular quantization into 9 evenly spaced orientation bins over 0°-180°(unsigned gradient). This last contribution is of great significance and it has been used in a lot of papers on this topic in the last ten years.

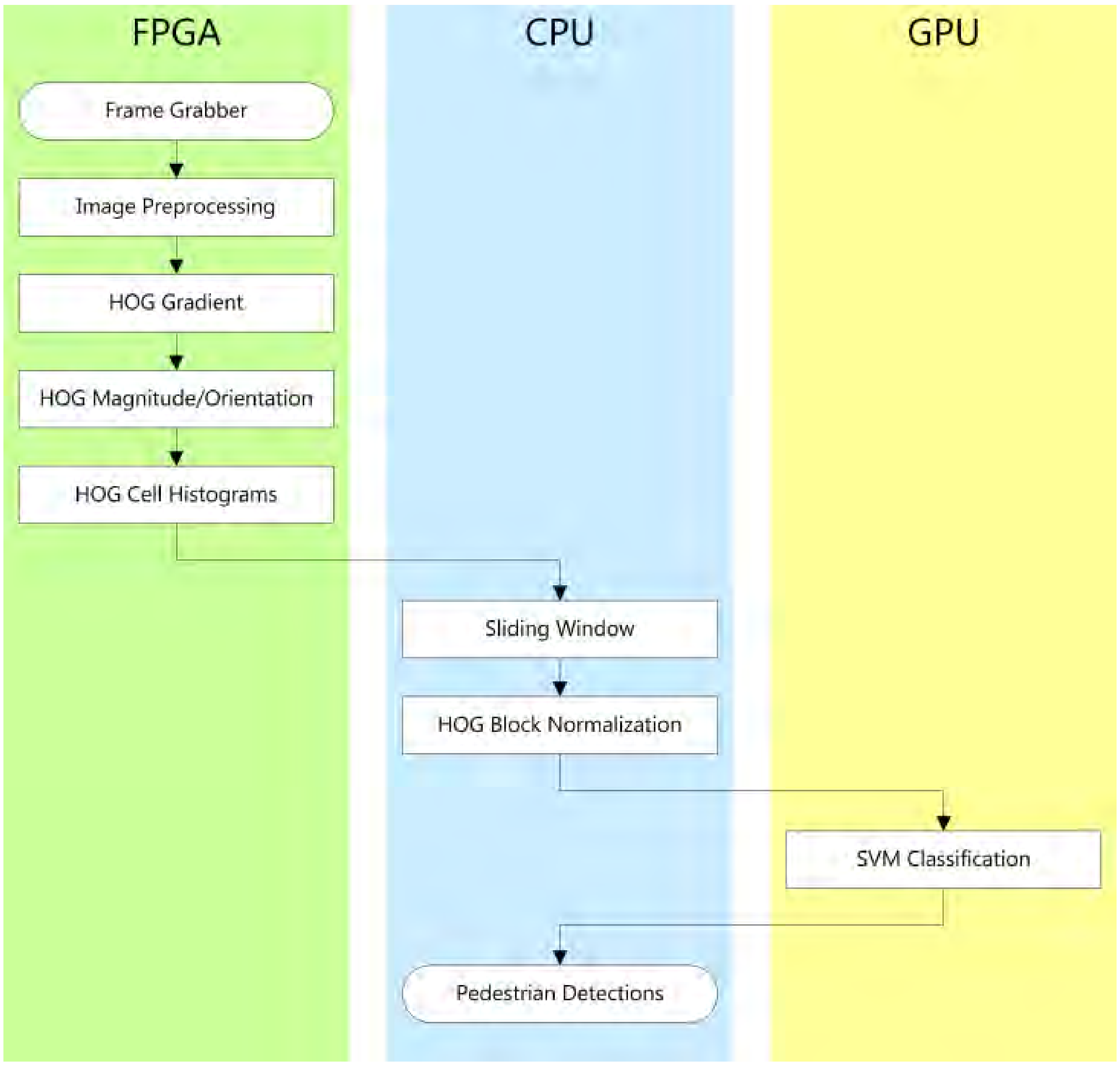


Figure 7.1: Overview of the implemented FPGA-CPU-GPU framework

FPGA-based Real-Time Pedestrian Detection on High-Resolution Images. ***Michael Hahnle, Frerk Saxen, Matthias Hisung, Ulrich Brunsmann, Konrad Doll*** [9]

Their test system consists of a Full-HD camera triggered to deliver 50 fps with 8 bit grayscale. The image data are transferred via GigE to an FPGA Board with a Xilinx Virtex 5 FPGA (XC5VFX200T). A GigE Vision Core receives the data and converts it to a pixel stream, which is analyzed by their pedestrian detection module. The results together with the input image are transferred by a PCIe Core 2 using DMA to a PC for further processing.

They proposed a more resource efficient implementation that presented them with the opportunity to instantiate several parallel accelerators for different scales of the input image. One of the was the introduction of the L1-sqrt-norm as the way to normalize the pixels of a block, which is very effective resource-wise compared to the L2-Hys norm which was used until then.

Binarization Based Implementation for Real-Time Human Detection. ***Shuai Xie, Yibin Li, Zhiping Jia, Lei Ju*** [10]

The proposed human detection was developed in Verilog-HDL and synthesized for a low-end Xilinx Spartan-3e XC3S500E device with Xilinx ISE 13.1 tools. To train the classifier, HOG features were generated by a modified OPENCV. Using Libsvm library of MATLAB, linear SVM classifier is trained offline, and detection accuracy is evaluated.

In this implementation, they adopted a modified binarization process in place of normalization process. With this process, classification process can be implemented by addition operation.

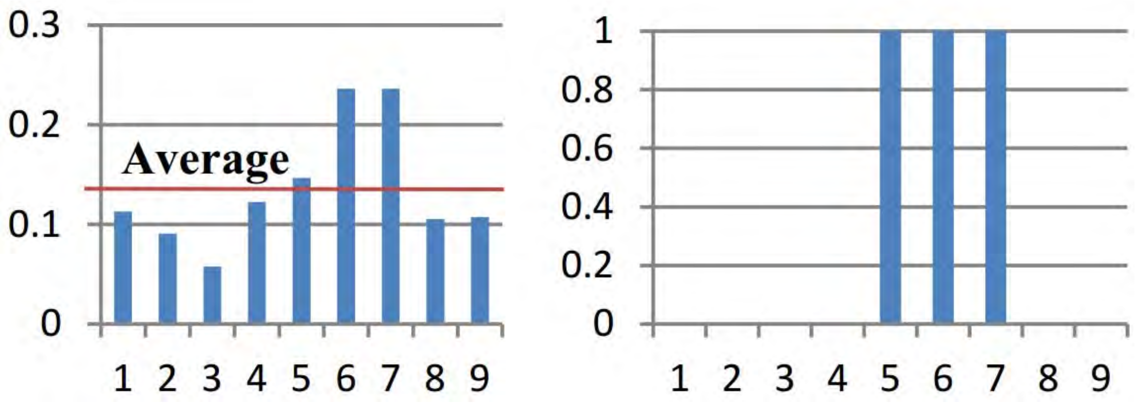


Figure 7.2: An example of binarization

FPGA Implementation of a Real-Time Pedestrian Detection Processor Aided by E-HOG IP. ***Ai-Ying Guo, Mei-Hua Xu, Feng Ran and Ang Li*** [11]

Their circuit board is divided into two boards: Cyclone III core and backboard. Through two independent boards, the backboard and core board can be extended to other functions. If this platform were to be promoted, just one of them may need to be amended. The core board contains one FPGA, two SDRAM and some other devices. The core device is Cyclone III. The backboard main devices are MT9 M111, one SRAM, power and interfaces for debugging and downloading program. E-HOG is embedded into the system on the backboard.

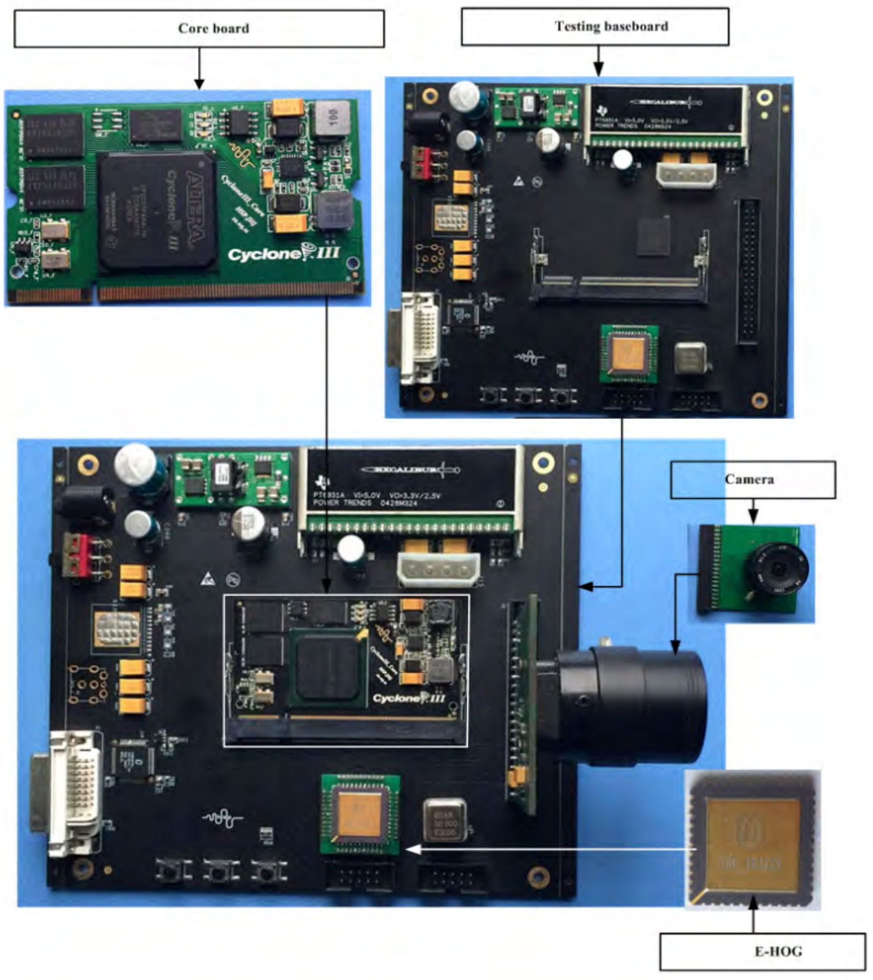


Figure 7.3: FPGA board with E-HOG IP

This paper has discussed a three-stage pedestrian detection on board that can be used as the real-time pedestrian detection processor. The three-stages contains: Sobel-step is applied as the first stage operator and to extract windows of interest; Uniform-LBP with SVM in the second stage can describe edge detection with less features; In the third stage, E-HOG and Linear SVM will distinguish the pedestrian zone from non-pedestrian. In order to decrease the cycle counts, cell-based is proposed to implement the uniform-LBP.

Chapter 8

Conclusion

In this thesis a real-time HOG algorithm implementation on a low-cost FPGA device is presented. Several optimizations both on algorithmic and architectural level are examined. Many approximation techniques are proposed without significant accuracy loss proving the algorithm's redundancy. Furthermore, a small modification of the SVM's parameters is introduced in order to yield better accuracy results. This implementation showcases the fact that high level synthesis despite lacking the versatility that an RTL design can offer, can still be used in time critical applications with satisfying results.

However, there is still room for future improvements to increase the performance of the system. First of all, since the amount of the parallelism is only bound by the available resources, a migration to a higher end FPGA would immediately result to a significant performance gain. In addition, a transition from high level synthesis to RTL design can be examined, since a more throughput-friendly design can be implemented in this case. Furthermore, it is a fact that is a stark contrast between the number of the overall detection windows in a single frame compared to the windows that actually contain a pedestrian. A way to mitigate this problem is to employ techniques with the aim of determining Regions Of Interest(ROI) and in this case, windows of interest. Finally, as far as accuracy is concerned, it could be improved by applying motion estimation techniques. For instance, in case of pedestrian detection, the neighbouring windows of the next frame could be classified in a more positively biased manner.

Bibliography

[1] European Road and Safety Observatory. *Advanced driver assistance systems.* URL: `https://ec.europa.eu/transport/road_safety/sites/roadsafety/files/ersosynthesis2016-summary-adas5_en.pdf`.

[2] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*. 2005, pp. 886–893. DOI: `10.1109/CVPR.2005.177`. URL: `https://doi.org/10.1109/CVPR.2005.177`.

[3] Amr Suleiman and Vivienne Sze. “An Energy-Efficient Hardware Implementation of HOG-Based Object Detection at 1080HD 60 fps with Multi-Scale Support”. In: *Signal Processing Systems* 84.3 (2016), pp. 325–337. DOI: `10.1007/s11265-015-1080-7`. URL: `https://doi.org/10.1007/s11265-015-1080-7`.

[4] Navneet Dalal. *INRIA Person Dataset.* URL: `http://lear.inrialpes.fr/data/human`.

[5] Thorsten Joachims. “Advances in Kernel Methods”. In: ed. by Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola. Cambridge, MA, USA: MIT Press, 1999. Chap. Making Large-scale Support Vector Machine Learning Practical, pp. 169–184. ISBN: 0-262-19416-3. URL: `http://dl.acm.org/citation.cfm?id=299094.299104`.

[6] M. Enzweiler and D. M. Gavrila. “Monocular Pedestrian Detection: Survey and Experiments”. In: *IEEE Transactions on Pattern Analysis and Machine*

Intelligence 31.12 (Dec. 2009), pp. 2179–2195. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2008.260.

[7] Sebastian Bauer, Ulrich Brunsmann, and Stefan Schlotterbeck-Macht. "FPGA Implementation of a HOG-based Pedestrian Recognition System". In: 2010.

[8] Michael Imhoff, Ursula Gather, and Katharina Morik. "Development of Decision Support Algorithms for Intensive Care Medicine: A New Approach Combining Time Series Analysis and a Knowledge Base System with Learning and Revision Capabilities". In: *KI-99: Advances in Artificial Intelligence, 23rd Annual German Conference on Artificial Intelligence, Bonn, Germany, September 13-15, 1999, Proceedings.* 1999, pp. 219–230. DOI: 10.1007/3-540-48238-5_18. URL: https://doi.org/10.1007/3-540-48238-5_18.

[9] Michael Hahnle et al. "FPGA-Based Real-Time Pedestrian Detection on High-Resolution Images". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2013, Portland, OR, USA, June 23-28, 2013.* 2013, pp. 629–635. DOI: 10.1109/CVPRW.2013.95. URL: https://doi.org/10.1109/CVPRW.2013.95.

[10] S. Xie et al. "Binarization based implementation for real-time human detection". In: *2013 23rd International Conference on Field programmable Logic and Applications.* Sept. 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645590.

[11] A.-Y Guo et al. "FPGA implementation of a real-time pedestrian detection processor aided by E-HOG IP". In: 28 (Apr. 2017), pp. 87–103.

[12] Kosuke Mizuno et al. "Architectural Study of HOG Feature Extraction Processor for Real-Time Object Detection". In: *2012 IEEE Workshop on Signal Processing Systems, Quebec City, QC, Canada, October 17-19, 2012.* 2012, pp. 197–202. DOI: 10.1109/SiPS.2012.57. URL: https://doi.org/10.1109/SiPS.2012.57.

[13] Navneet Dalal. "Finding People in Images and Videos". PhD thesis. Grenoble Institute of Technology, France, 2006. URL: https://tel.archives-ouvertes.fr/tel-00390303.

[14] Ai-Ying Guo et al. “FPGA Implementation of a Real-Time Pedestrian Detection Processor Aided by E-HOG IP”. In: 2017.

[15] M. Bilal et al. “A Low-Complexity Pedestrian Detection Framework for Smart Video Surveillance Systems”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.10 (Oct. 2017), pp. 2260–2273. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2016.2581660.

[16] Maryam Hemmati et al. “HOG Feature Extractor Hardware Accelerator for Real-Time Pedestrian Detection”. In: *17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014*. 2014, pp. 543–550. DOI: 10.1109/DSD.2014.60. URL: https://doi.org/10.1109/DSD.2014.60.

[17] Piotr Dollár et al. “Pedestrian detection: A benchmark”. In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. 2009, pp. 304–311. DOI: 10.1109/CVPRW.2009.5206631. URL: https://doi.org/10.1109/CVPRW.2009.5206631.

[18] Christian Wojek et al. “Sliding-Windows for Rapid Object Class Localization: A Parallel Technique”. In: *Pattern Recognition, 30th DAGM Symposium, Munich, Germany, June 10-13, 2008, Proceedings*. 2008, pp. 71–81. DOI: 10.1007/978-3-540-69321-5_8. URL: https://doi.org/10.1007/978-3-540-69321-5_8.

[19] M. Hiromoto and R. Miyamoto. “Hardware architecture for high-accuracy real-time pedestrian detection with CoHOG features”. In: *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. Sept. 2009, pp. 894–899. DOI: 10.1109/ICCVW.2009.5457609.

[20] Kosuke Mizuno et al. “A Sub-100 mW Dual-Core HOG Accelerator VLSI for Parallel Feature Extraction Processing for HDTV Resolution Video”. In: *IEICE Transactions* 96-C.4 (2013), pp. 433–443. URL: http://search.ieice.org/bin/summary.php?id=e96-c_4_433.

[21] Kazuhiro Negi et al. “Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm”. In: *2011 International Conference on Field-Programmable Technology, FPT 2011, New Delhi, India, December 12-14, 2011*. 2011, pp. 1–8. DOI: 10.1109/FPT.2011.6132679. URL: https://doi.org/10.1109/FPT.2011.6132679.

[22] Aliaksei Kerhet et al. “Distributed video surveillance using hardware-friendly sparse large margin classifiers”. In: *Fourth IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2007, 5-7 September, 2007, Queen Mary, University of London, London, United Kingdom*. 2007, pp. 87–92. DOI: 10.1109/AVSS.2007.4425291. URL: https://doi.org/10.1109/AVSS.2007.4425291.

[23] Ryoji Kadota et al. “Hardware Architecture for HOG Feature Extraction”. In: *Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2009), Kyoto, Japan, 12-14 September, 2009, Proceedings*. 2009, pp. 1330–1333. DOI: 10.1109/IIH-MSP.2009.216. URL: https://doi.org/10.1109/IIH-MSP.2009.216.

[24] Mateusz Komorkiewicz, Maciej Kluczewski, and Marek Gorgon. “Floating point HOG implementation for real-time multiple object detection”. In: *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*. 2012, pp. 711–714. DOI: 10.1109/FPL.2012.6339159. URL: https://doi.org/10.1109/FPL.2012.6339159.

[25] Vinh Ngo et al. “A pipeline hog feature extraction for real-time pedestrian detection on FPGA”. In: *2017 IEEE East-West Design & Test Symposium, EWDTS 2017, Novi Sad, Serbia, September 29 - October 2, 2017*. 2017, pp. 1–6. DOI: 10.1109/EWDTS.2017.8110057. URL: https://doi.org/10.1109/EWDTS.2017.8110057.

[26] Vinh Ngo et al. “A High-Performance HOG Extractor on FPGA”. In: *CoRR* abs/1802.02187 (2018). arXiv: 1802.02187. URL: http://arxiv.org/abs/1802.02187.

[27] Christian Wojek, Stefan Walk, and Bernt Schiele. "Multi-cue onboard pedestrian detection". In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. 2009, pp. 794–801. DOI: `10.1109/CVPRW.2009.5206638`. URL: `https://doi.org/10.1109/CVPRW.2009.5206638`.

[28] Victor Prisacariu and Ian Reid. *fastHOG - a real-time GPU implementation of HOG*. Tech. rep. 2310/09. Department of Engineering Science, Oxford University, 2009.

[29] Mariana-Eugenia Ilas and Constantin Ilas. "A New Method of Histogram Computation for Efficient Implementation of the HOG Algorithm". In: *Computers* 7.1 (2018), p. 18. DOI: `10.3390/computers7010018`. URL: `https://doi.org/10.3390/computers7010018`.

[30] Xilinx. *Zedboard evaluation kit*. URL: `http://zedboard.org/product/zedboard`.

[31] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[32] Xilinx. *Vivado Design Suite - User Guide - ug902 - High-Level Synthesis*. URL: `https://www.xilinx.com/support/documentation/sw%5C_manuals/xilinx2016%5C_4/ug902-vivado-high-level-synthesis.pdf`.

[33] Xilinx. *Vivado Design Suite - User Guide - ug908 - Vivado IDE*. URL: `https://www.xilinx.com/support/documentation/sw%5C_manuals/xilinx2016%5C_4/ug893-vivado-ide.pdf`.

[34] Xilinx. *PetaLinux Tools - Documentation - ug1144 - Reference Guide*. URL: `https://www.xilinx.com/support/documentation/sw%5C_manuals/xilinx2016%5C_4/ug1144-petalinux-tools-reference-guide.pdf`.