Real time HOG implementation

Katsaros Nikolaos
Patsiatzis Nikolaos

Supervisor: Assoc. Prof. Mpellas Nikolaos

University of Thessaly
Department of Electrical and Computer Engineering
Volos, Greece

Team number: xohw18-222

Video available on Youtube

Real time HOG implementation

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.

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 project, 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.

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.

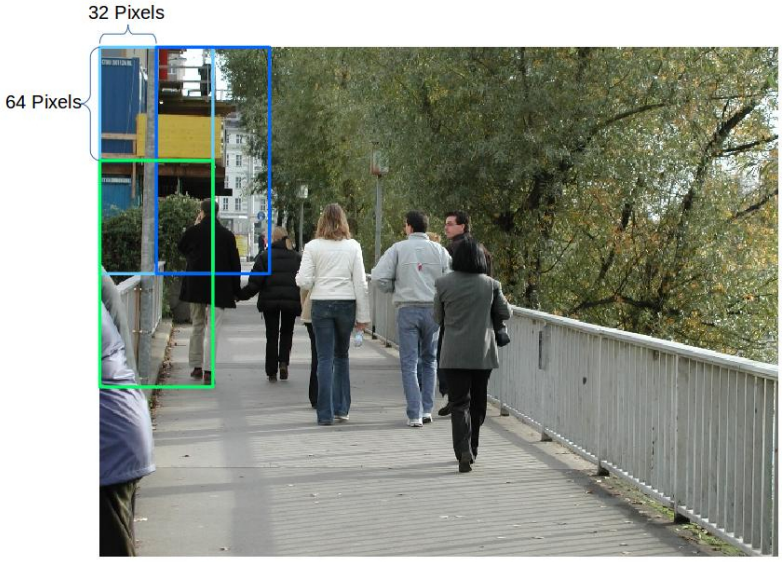

Figure 2.1: Sliding windows inside a frame

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.

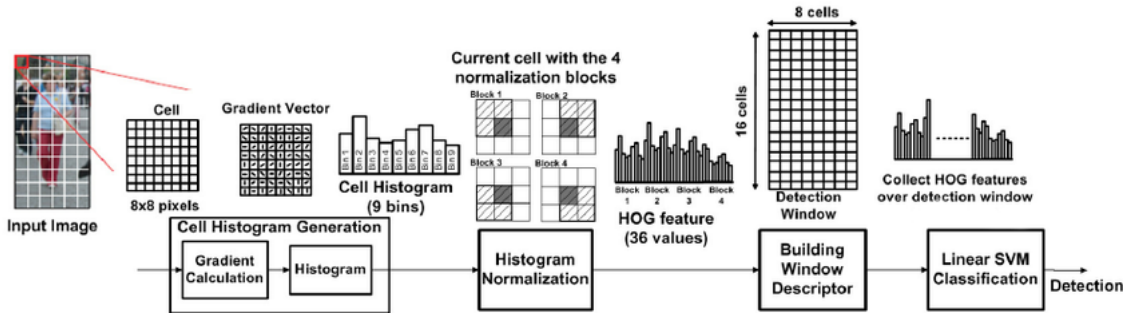

Figure 2.2: Overview of the HOG algorithm [3]

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.

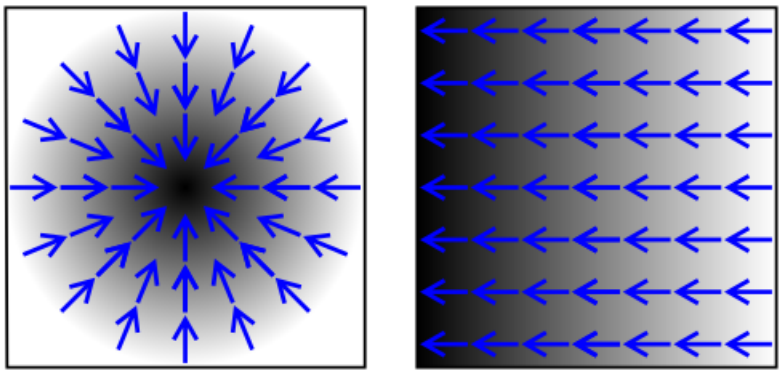

Figure 2.3: Gradient magnitudes and angle representation

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.

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

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

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

The gradient orientation angle can be calculated as:

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

The Fig 2.4 shows the effect of applying the gradient function in a random part of an input image.

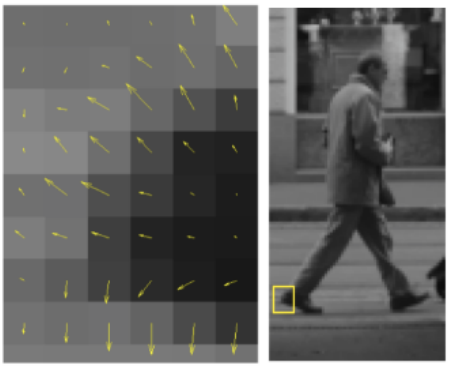

Figure 2.4: Pedestrian gradient example from the INRIA dataset [4].

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°.

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

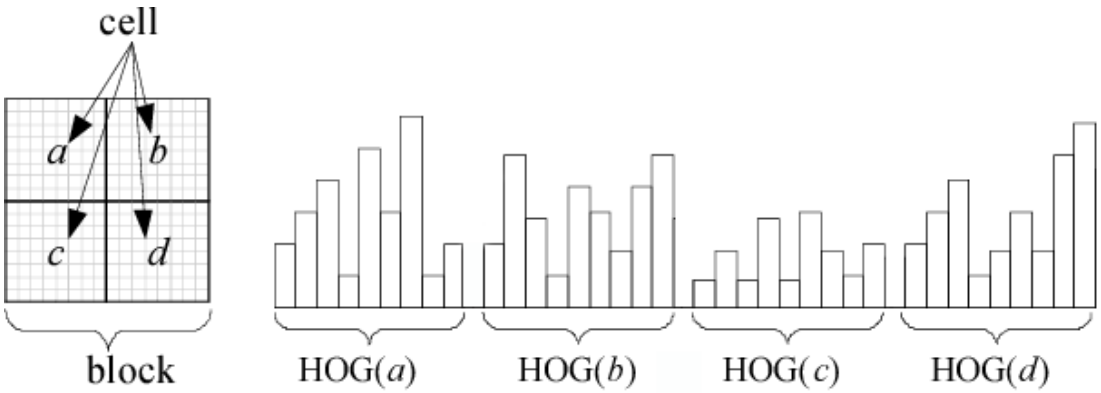

Figure 2.5: The concatenated HOG descriptor

The HOG descriptor is represented by a concatenation of all these blocks as it can be seen in Fig 2.5. 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. 2.6 each detection window is represented by 7x15 blocks. A block consists of 2x2 cells, a cell is represented by a 9-bin histogram, giving a total of (7x15)x(2x2)x9 = 3780 features.

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

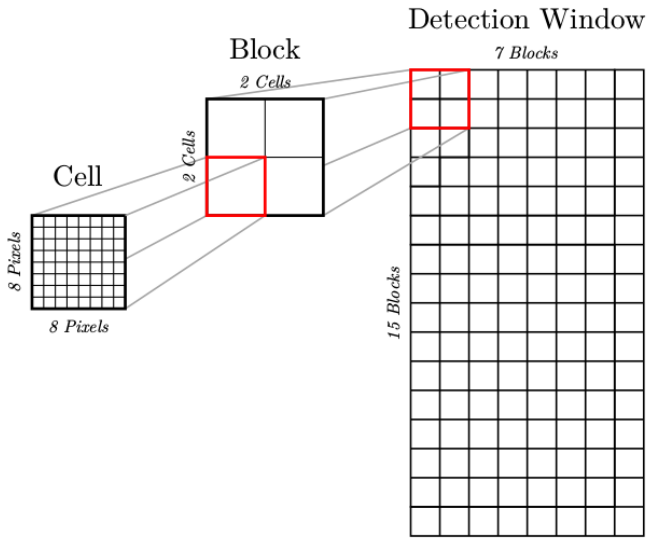

Figure 2.6: Division of the detection window into blocks and cells

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.

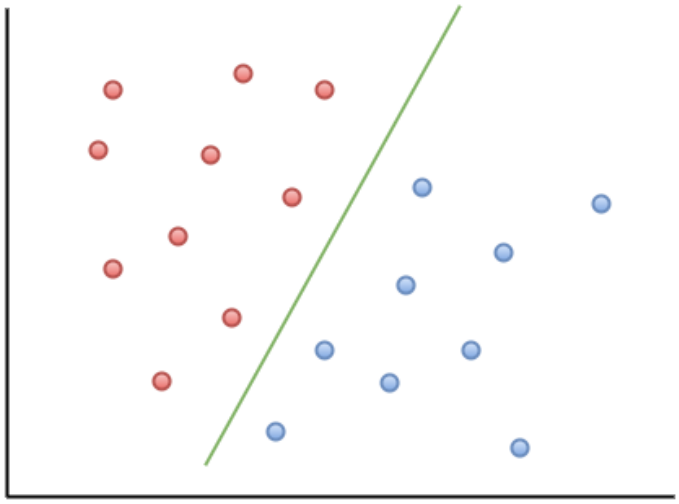

Figure 2.7: Linearly separable data points that belong to one of two classes, red and blue

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:

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

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 2.8: 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. 2.8). 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. 2.8.

Chapter 3

Design

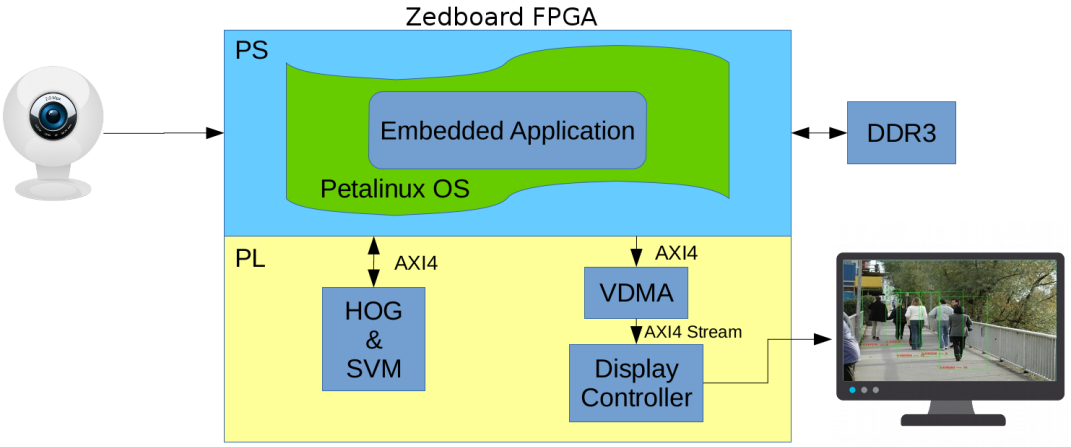

Figure 3.1: System Overview

3.1 Design 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.

3.1.1 Petalinux

As an embedded OS, Petalinux 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. Using Ubuntu on Zedboard, we had the opportunity to perform the development and debug process on the platform without the need of recompiling the kernel for every modification on the embedded application.

3.1.2 Webcam Interface

Since we develop a computer vision application, it is obvious that a video input is needed. In order to interface a USB webcam, we need to install and include a few kernel modules to the Petalinux kernel image, more specifically the USB Video Class (UVC) drivers under the USB support section of the Petlinux kernel configuration. After setting the configurations, we also need to edit the device tree so that USB hardware on the Zedboard is identified by the Linux kernel.

3.1.3 Custom Hardware Interface

In addition, the operating system needs to be able to access the implemented hardware, that being HOG & SVM as shown in Fig. 3.1. 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.

3.1.4 Display Controller

In order to render pedestrian detections output on a VGA monitor a VGA controller was needed. We used the Display Controller, which is a 3^{rd} party IP provided by Digilent, whose functionality will be covered in Sec. 3.2.

3.2 Hardware

3.2.1 Initial Hardware Implementation

The initial implementation of the algorithm was based on the proposed one by Dalal & Triggs except for the histogram interpolation. 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. However, this initial implementation was not able to achieve real-time performance. Nonetheless, 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 following sections present 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. Additionaly, other optimizations are showcased, which aim to **increase the throughput of the design**.

3.2.2 HOG Approximations

Histogram Binning

An important source of complexity (esp. for hardware implementation) is the precise computation of complex functions (such as trigonometric functions or the square root). 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 in lookup tables) 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. 3.2. This way the computation of *arctan* is replaced by simple integer multiplications.

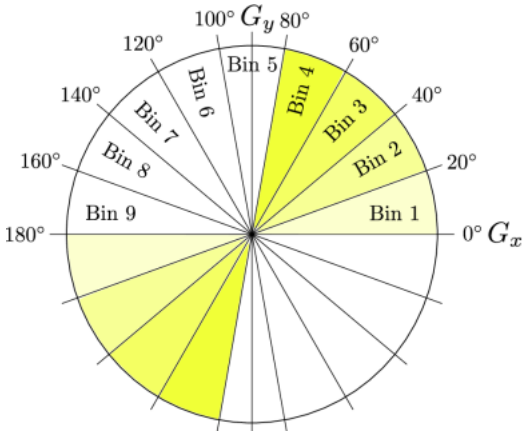

Figure 3.2: Angular quantization into 9 evenly spaced orientation bins over 0°-180°(unsigned gradient).

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

```
1: procedure ANGULAR_QUANTIZATION(Gx, Gy)
2:     Input = Gx, Gy
3:         /*Quadrants I & III*/
4:     if (Gx > 0 and Gy > 0) or (Gx < 0 and Gy < 0) then
5:         if (|Gy| < tan(20°) * |Gx|) then
6:             bin = 1
7:         else if (|Gy| < tan(40°) * |Gx|) then
8:             bin = 2
9:         else if (|Gy| < tan(60°) * |Gx|) then
10:            bin = 3
11:        else if (|Gy| < tan(80°) * |Gx|) then
12:            bin = 4
13:        else
14:            bin = 5
15:        end if
16:        /*Quadrants II & IV*/
17:    else
18:        if (|Gy| < tan(20°) * |Gx|) then
19:            bin = 5
20:        else if (|Gy| < tan(40°) * |Gx|) then
21:            bin = 6
22:        else if (|Gy| < tan(60°) * |Gx|) then
23:            bin = 7
24:        else if (|Gy| < tan(80°) * |Gx|) then
25:            bin = 8
26:        else
27:            bin = 9
28:        end if
29:    end if
30:    Return bin
31: end procedure
```

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} False Positives Per Window (FPPW). Due to those reasons we decided to transition from RGB images to grayscale ones.

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 (4.4% at 10^{-4} FPPW), 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.

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. Our experiments show that this step leads to 4.1% at 10^{-4} FPPW.

For our implementation only the odd columns of each cell contribute to the HOG descriptor as shown in Fig. 3.3.

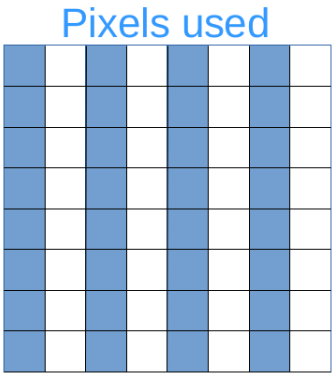

Figure 3.3: Overview of a cell's pixels used

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 1% at 10^{-4} FPPW accuracy loss.

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

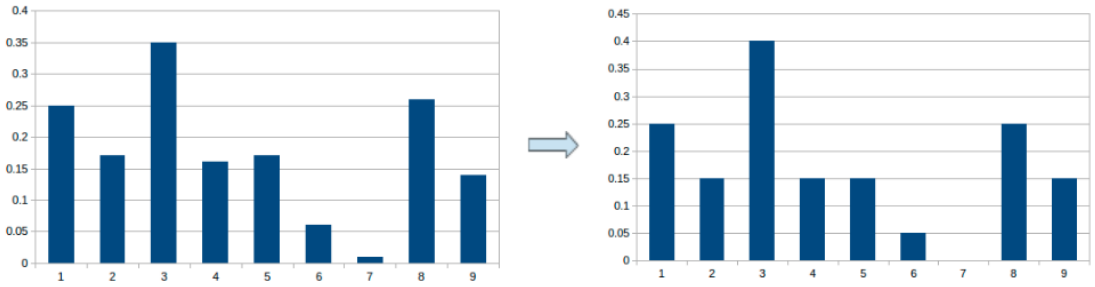

Figure 3.4: Small scale example of the quantized normalization method

3.2.3 Gradient and Histogram merge

The programming style in FPGAs should be 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.

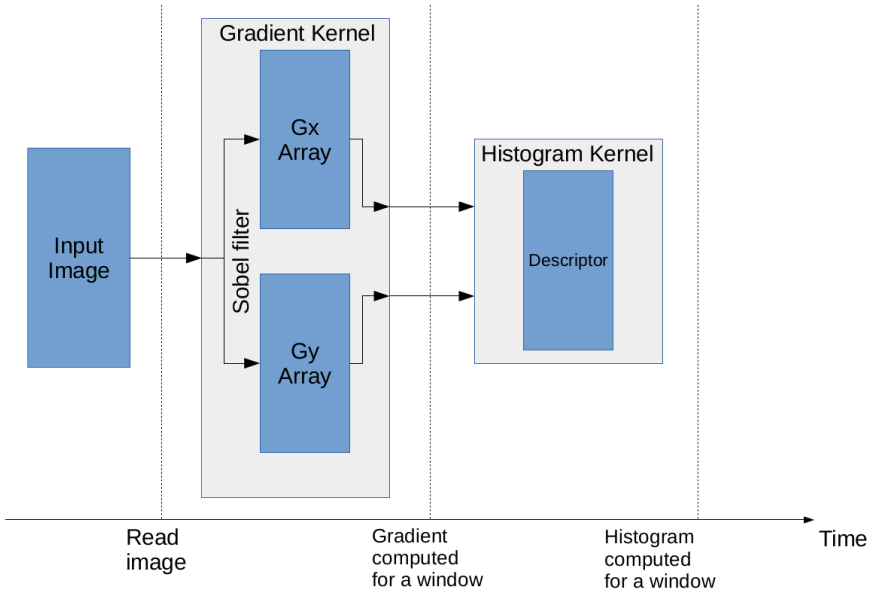

Figure 3.5: Initial Hardware implementation

As it can be seen from Fig. 3.6, 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.

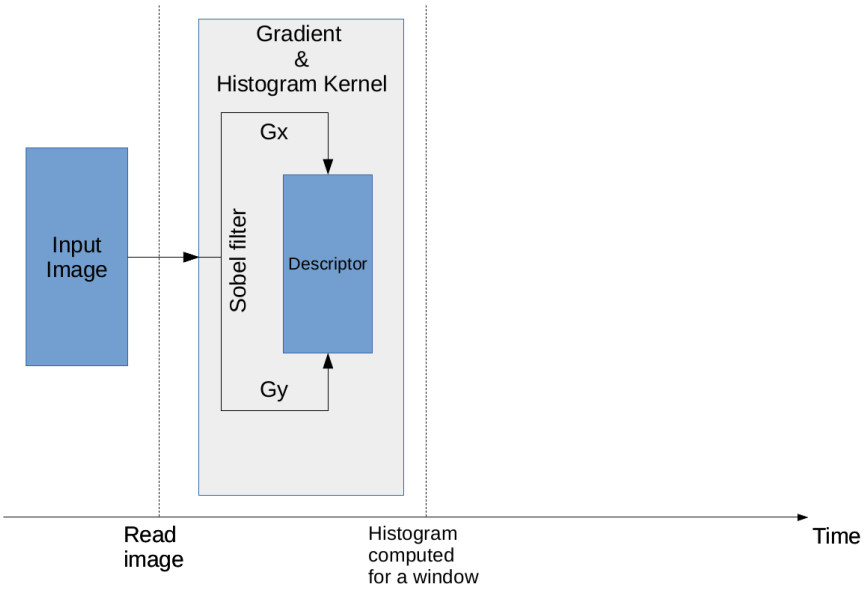

Figure 3.6: Hardware stream-like implementation

3.2.4 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, 4.4% at 10^{-4} FPPW.

3.2.5 Arbitrary precision and throughput

A thorough bitwidth analysis was performed and 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 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.

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 3.7:

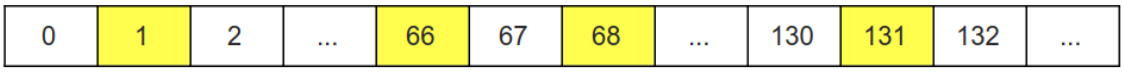

Figure 3.7: Initial input image array

An overview of the partitioned array and its access pettern can be seen in Fig 3.8.

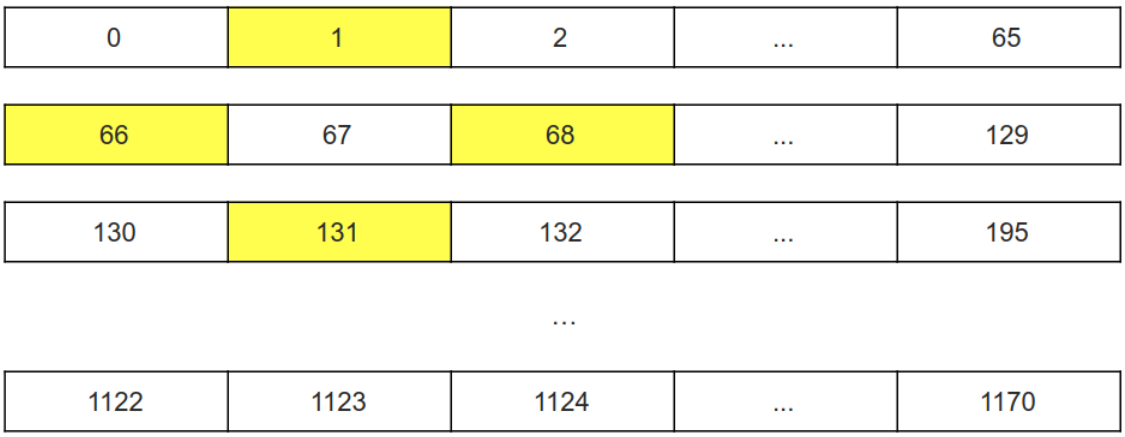

Figure 3.8: Partitioned image array

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.

3.2.6 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. 3.9. 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.

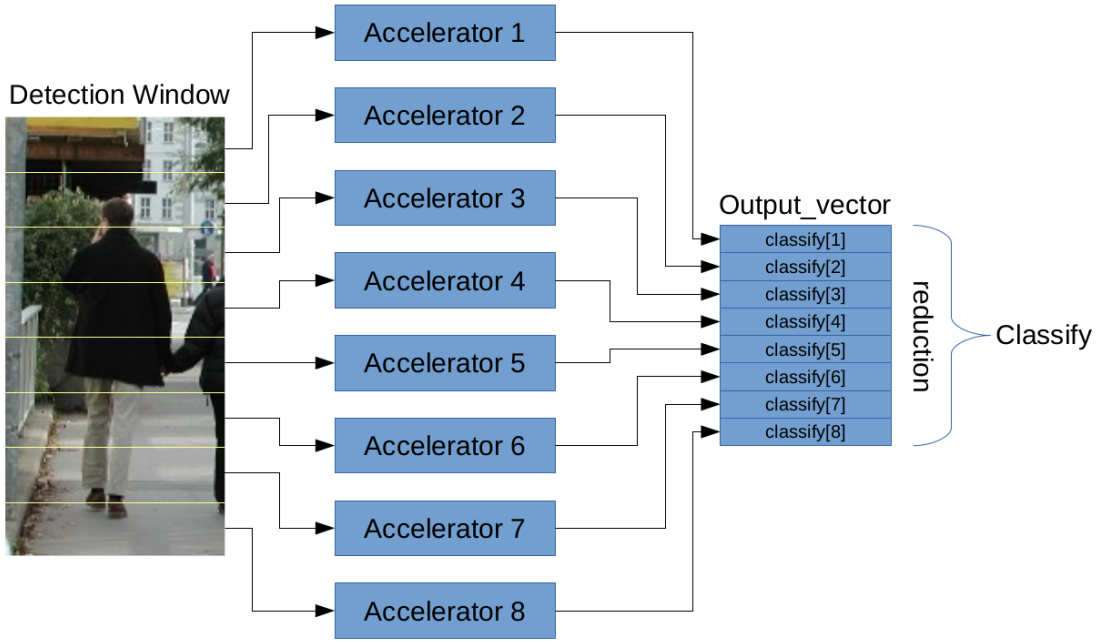

Figure 3.9: Abstract overview of the implemented hardware input & output

In Fig. 3.10 an overview of a single accelerator's flow is presented.

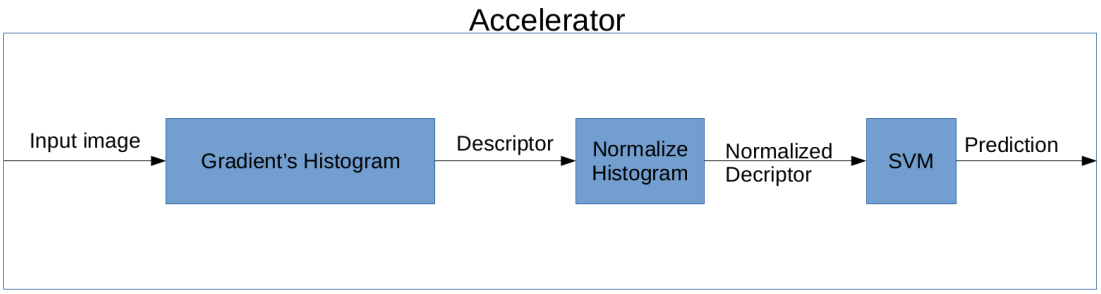

Figure 3.10: Abstract overview of the accelerator's flow

3.2.7 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, the available LUTs wouldn't suffice since the LUT utilization in the previous step was over 60% of the total. 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. 3.11, 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.

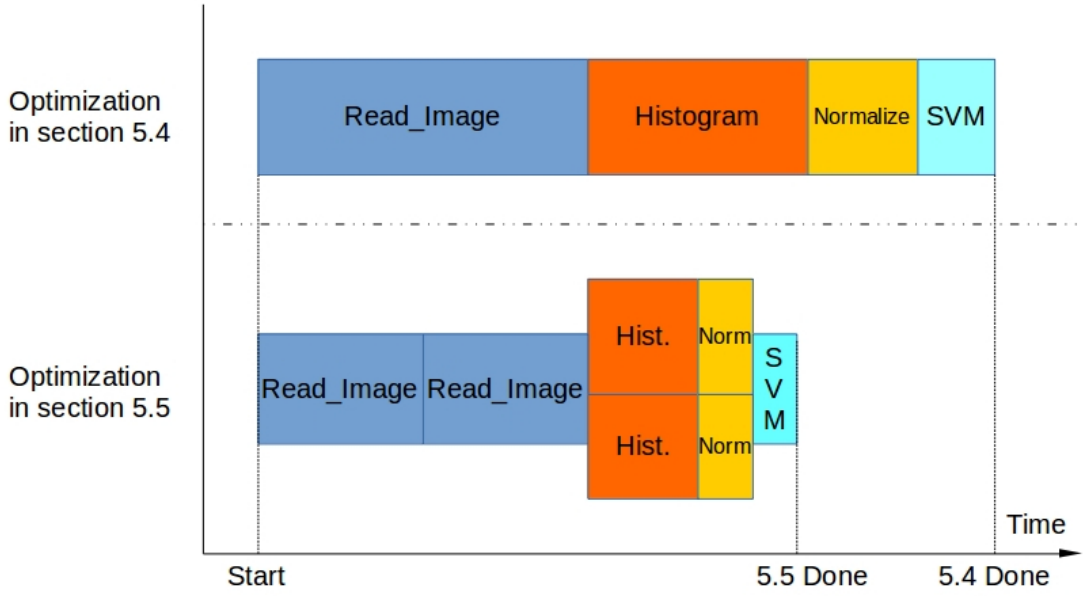

Figure 3.11: Parallel data processing

3.2.8 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. 3.12:

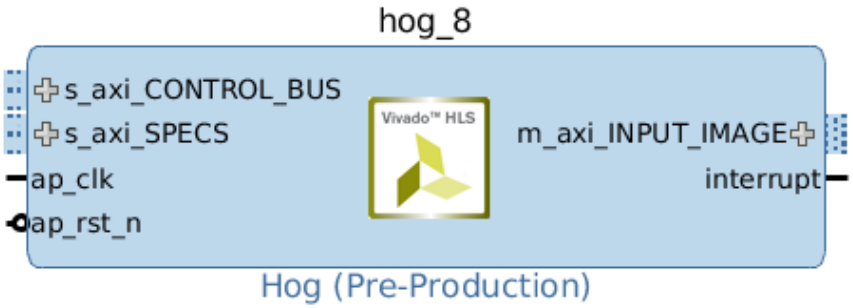

Figure 3.12: Overview of the accelerator's IP ports

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.

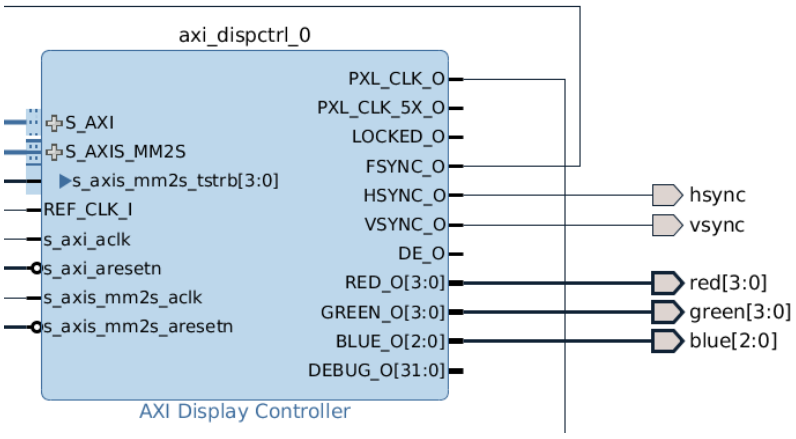

Figure 3.13: Overview of the Display Controller IP

In Fig. 3.13 there is an overview of the AXI Display Controller, a 3^{rd} party IP from Digilent. This IP is used to render the final output of our application to a VGA

monitor through the R,G,B, hsync and vsync external ports. It is initialized and monitored through the S_AXI AXI4 Lite port, while the input is read through the S_AXIS_MM2S AXI4 Stream port. Basically, he display controller accesses the output image data through a Video DMA and renders the image to a VGA monitor.

Initially, we had designed our own VGA controller, which stored 640x480 images into a Block RAM. However, as we were instantiating more accelerators, the resources available did not suffice. Therefore, we decided to use this IP since its stream like implementation had a small memory footprint.

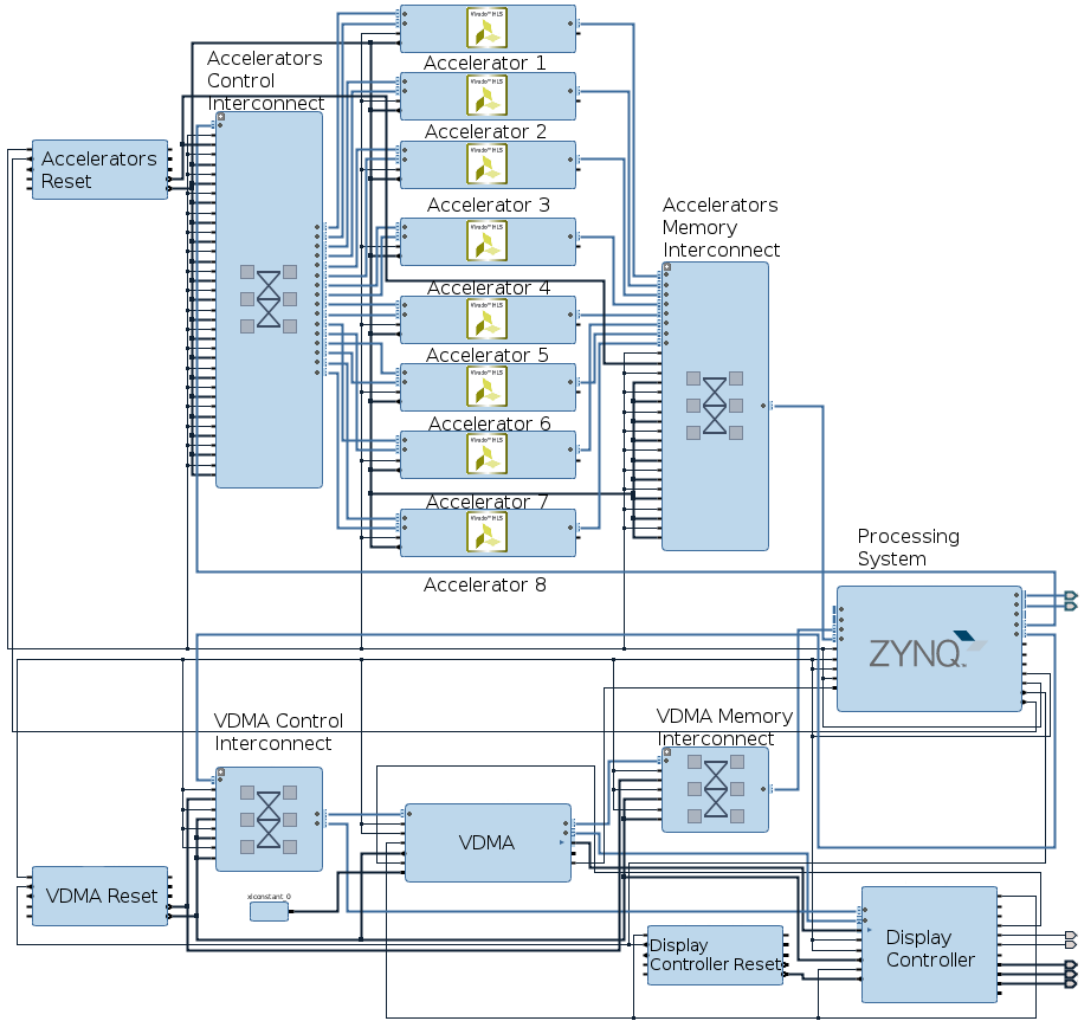

Figure 3.14: Overview of the block design

As shown in Fig. 3.14, one AXI interconnect for the accelerators data is instantiated 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 sys-

tem supports up to 4 separate clocks. In this case, a 190 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.

3.3 Software

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. The fact that we use an Ubuntu image for our Petalinux kernel gives us the opportunity to use frameworks such as OpenCV, OpenMP etc.

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.

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. For all the functionality of the controller we used the provided drivers from Digilent.

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

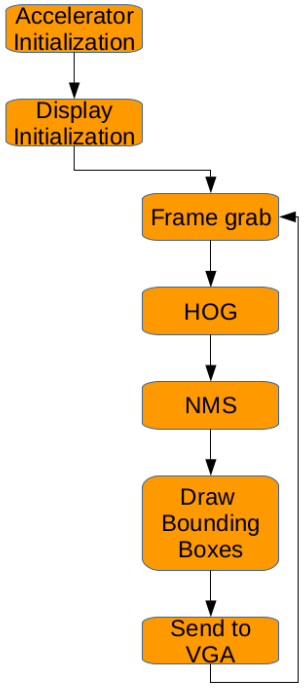

Figure 3.15: Application flow

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%. We have enhanced this function with one more functionality. More specifically, we draw only windows with 2 or more overlaps in order to ignore any random false positives across the scales.

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.

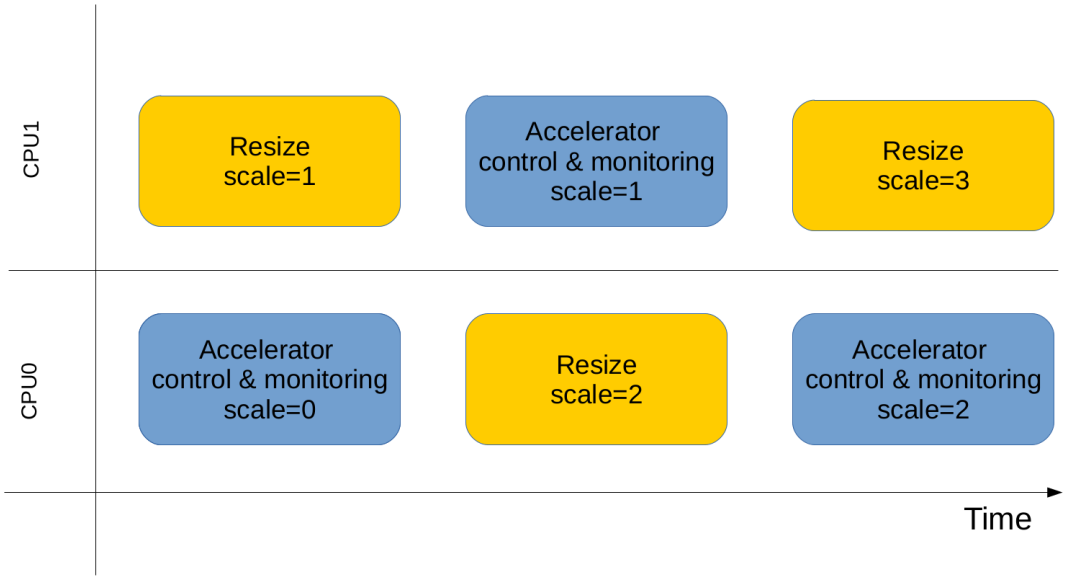

Figure 3.16: Time representation of the double buffering

3.4 Design Reuse

The project is well documented with the appropriate information inside the code. We have created a software wrapper that invokes the accelerators and it is parameterizable in order to suit the needs of any project of this kind. All our IPs and project files are **open-source** and available on Github

Chapter 4

Results

4.1 Challenges

The Histogram of Oriented Gradients is a relatively resource heavy algorithm. Therefore, one of our main challenges was to decrease the memory footprint while at the same time increasing the throughput of our design. Another challenge was the integration of our implemented hardware in an embedded system running on Petalinux. That's because there are several steps that should be performed in order for our accelerators to be acknowledged as peripherals of the system.

4.2 Implementation details and results

In Fig. 4.1 the resource utilization from Vivado HLS and the performance for each optimization step described in Sec. 3.2 are shown. The performance is measured in windows per second which are calculated as:

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

The clock period of the implemented hardware is 5.25ns which means that the clock speed is 190MHz.

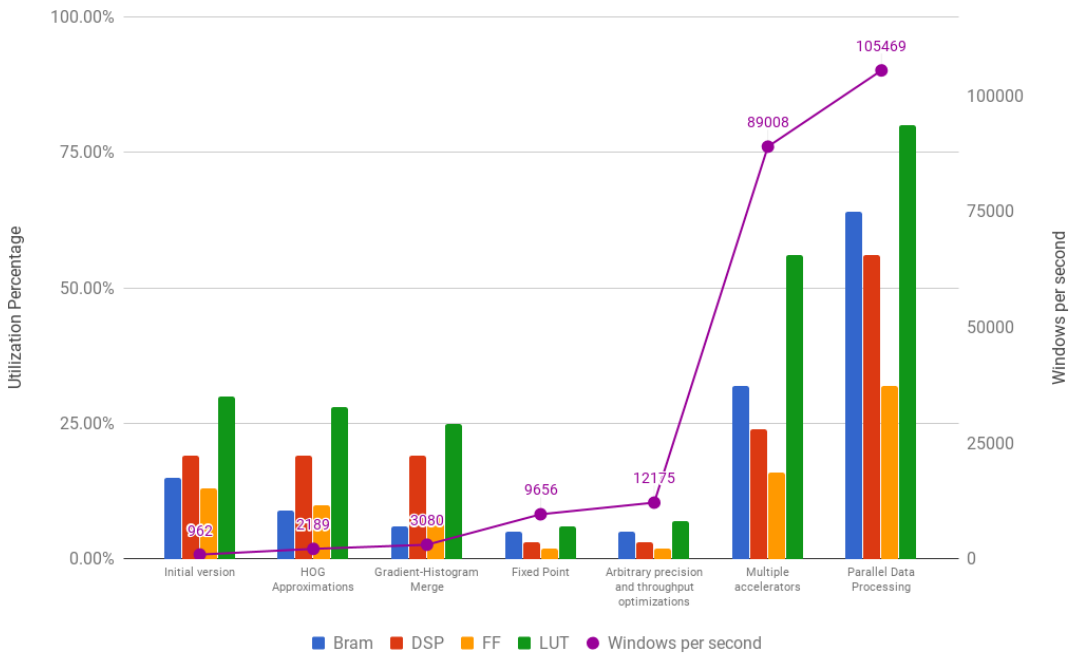

Figure 4.1: Resources vs Latency for the all optimization steps

Fig. 4.2 shows the Post-Implementation resource utilization for the whole system. As it can be seen, Vivado managed to reduce the utilizations that were estimated in Vivado HLS.

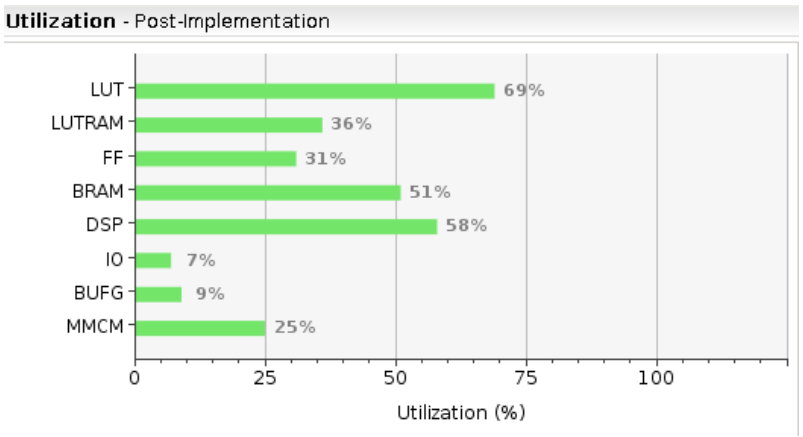

Figure 4.2: Post-Implamentation utilization

4.3 Embedded System Performance

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 4.1. The **Initial Implementation** is the version of the HOG algorithm without any software optimizations executed on the ARM. The **Optomized Software** is version of the algorithm with all the approximations that we described. 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 **Embedded Platform** utilizes the hardware accelerators for the HOG extraction and classification as described in Sec. 3.2.

Initial Implementation	Optimized Software	Embedded Platform
30s	1.450s	0.080s

Table 4.1: Performance comparison

It is clear that the approximation techniques performed on the software as well as the use of the OpenMP framework yield a more that 20x better performance than the Initial Implementation. However, with hardware optimization techniques on the FPGA, the system was able to perform in real-time achieving **384x acceleration** from the initial implementation, while experiencing an **accuracy loss of just 1%** on the Dailmer Test-set.

Chapter 5

Conclusion

In this project 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. 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] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.