POLITECNICO DI TORINO

Master degree course in Ingegneria Elettronica

Master Degree Thesis

FHOG

Design of an optimized FHOG architecture with Vivado HLS

Supervisors
Prof. Maurizio Martina
Dr Walid Walid

Candidate
Pietro Romeo
matricola: 269010

Anno accademico 2023-2024

Summary

Object detection is one of the modern challenges in computer vision. There are many ways to detect classes of objects from static images, and most of them are based on the Histogram of Oriented Gradients (HOG) algorithm. Pixels have sharp variations at the boundary of objects, and this is exploited in HOG, where pixels' gradients are computed to highlight such boundaries. The first step in HOG based object detection is therefore to generate a "Feature Map", a version of the image comprised of gradient vectors that allow to proceed with the actual object detection.

The ability to process videos as well as images with HOG opens many possibilities in terms of possible applications, such as object recognition for cars, robots etc., security footage, smart search to find sequences of interest in movies, just to name a few. Such applications are already possible with the HOG algorithm, which is however outdated and has several limitations, including being limited to a single class of objects at a time.

The goal of this thesis is to provide an optimized hardware implementation on FPGA capable of executing an advanced version of the HOG algorithm, the one proposed by Felzenszwalb and his colleagues, which will be referred to as FHOG, or Felzenszwalb's HOG. FHOG is more complex than HOG, it does not suffer from its limitations and its results are compact and easy to process.

There are many software implementations for the FHOG algorithm that are capable of executing it with high precision,but so far there have been no attempts to implement the FHOG algorithm in hardware. A hardware implementation would extend the possible applications for the algorithm. This work will be useful to anyone who is seeking to implement a detection or tracking system with FHOG at its core. Since there are multiple algorithms that use FHOG, each with its own requirements, an advanced digital design tool, namely HLS or High Level Synthesis, was selected. By using it the

result is not specific to a single application, since HLS' versatility allows to change and adapt the design with very little effort.

The thesis is organized into five chapters, out of which there's an introduction, a conclusion and three core chapters. In Chapter 1, an introduction to the HOG algorithm is presented, summarizing the steps of the algorithm and showing both its potential and limitations. Some of the space in the chapter is also dedicated to introducing Vivado HLS.

In Chapter 2 the FHOG algorithm is explained in detail, highlighting key passages and differences with respect to the original HOG algorithm [2]. By the end of the Chapter, the algorithm is also simulated with MATLAB and its output is compared with the one provided by an already existing function.

In Chapter 3, several optimizations and approximations for the FHOG are discussed and verified. These optimizations mostly have the effect of increasing performance at the cost of precision, and whether the trade off is acceptable or not is going to be determined by applying the author's custom MATLAB implementation to an FHOG based object tracking system: the fast Discriminative Scale Space Tracker (fDSST).

In Chapter 4, an optimized hardware architecture for FHOG is presented and its performance is compared with several other architectures found in literature. The focus of this implementation is going to be high throughput and low memory usage. This is achieved by the use of traditional techniques such as pipelining and unrolling, as well as the use of line buffers to store intermediate results and use them immediately.

Chapter 5 concludes the thesis by giving suggestions on how to use and enrich this work, and summarizing the obtained results.

Contents

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

In 2006 Navneet Dalal, a Ph.D. student of the Institut National Polytechnique de Grenoble (INPG), published a revolutionary thesis [3] focused on the development of a new object recognition technique based on the Histogram of Oriented Gradients (HOG). The paper summarizing this work [2], with over twenty five thousand citations, set a new standard in the field. While revolutionary, HOG based object recognition system suffer from several limitations. In 2010, a paper presenting an improved HOG algorithm [1] was published. This algorithm, now known as FHOG, is a direct improvement over its predecessor. The objective of this work is to present an FPGA implementation of the FHOG algorithm.

1.1 Histogram of Oriented Gradients

While this work is centered upon an FHOG implementation, the author believes it useful to first present an introduction to the HOG algorithm, since it is more simple and what is said about HOG can in most cases be applied to FHOG as well.

The objective of the HOG algorithm is that of extracting features from images, from a computer vision point of view, so that such features can be used to identify classes of objects, for example cars. Cars can vary in shape, size, color, orientation and they can be illuminated from various angles. HOG aims to capture the essential information that are required to identify a car, despite possible variations in its appearance, and it can do so for several other classes of objects. In particular, HOG is specialized in human detection.

There are several applications for HOG, including smart search of objects

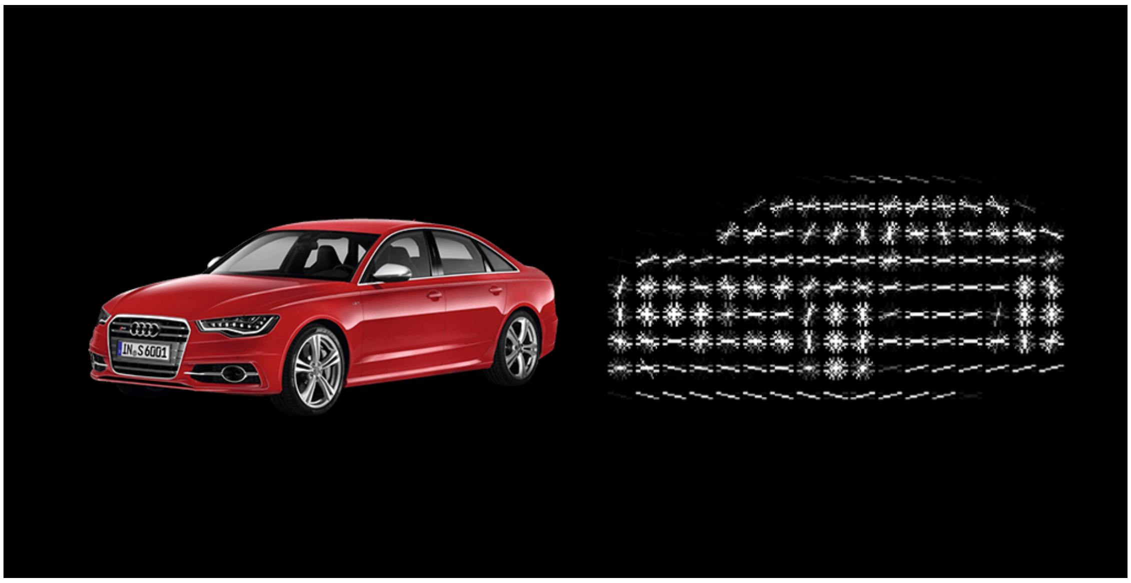


Figure 1.1: An example of a car image with HOG [17].

in both videos and pictures, security cameras, intelligent digital content management softwares, pedestrian detection for smart cars and so on. In order to do that, the HOG feature map alone is not enough, but it is simply a starting point for several types of algorithms that compare the feature map with complex models. Such algorithms include fDSST (a tracking algorithm), and machine learning algorithms based on Support Vector Machines. The focus of this thesis, however, lies only with the feature map generation.

So how is the feature map generated? There are several steps to follow, that can vary according to the object class:

1. For each colour channel, convolve with [-1,0,1] mask along x and y axis, and compute gradients. The channel[1] with the largest magnitude gives the pixel's dominant orientation and magnitude.

2. Divide the image into square regions, or cells.

3. Group cells into overlapping squares of 2x2, or blocks.

4. Create a spatial and orientation histogram, where for each block trilinear interpolation is used by each pixel to votein the histogram using gradient

[1]Coloured pictures have three channels, Red, Green and Blue

magnitude[2].

5. Apply L2-Hys norm or L1-sqrt normalization independently to each block.

For a more in-depth, but simple explanation of the algorithm, see [17], or check Chapter 2, where FHOG is thoroughly explained and compared with HOG. Figure 1.2 also gives a graphical representation of these steps.

There are several variables in the HOG algorithm, including image size, cell size, block size, and number of bins. Regardless, when referring to HOG, it is assumed that a "standard" choice of parameters for human detection is used. Such parameters include Detection window size (64x128), number of orientation bins (9), cell size (8x8), block size (2x2 cells), and normalization type (L2-Hys norm). Figure 1.1 shows which parameters are suitable for which object classes.

Table 1.1: HOG parameters for different object classes [3].

Class	**Window Size**	**Avg. Size**	**Orient. Bins**	**Orient. Range**	**Gamma Compr.**	**Norm. Method**
Person	64x128	H* 96	9	(0-180°)	$\sqrt{RGB}$	L2-Hys
Car	104x56	H 48	18	(0-360°)	$\sqrt{RGB}$	L1-Sqrt
Motorbike	120x80	W* 112	18	(0-360°)	$\sqrt{RGB}$	L1-Sqrt
Bus	120x80	H 64	18	(0-360°)	$\sqrt{RGB}$	L1-Sqrt
Bicycle	104x64	W 96	18	(0-360°)	$\sqrt{RGB}$	L2-Hys
Cow	128x80	W 96	18	(0-360°)	$\sqrt{RGB}$	L2-Hys
Sheep	104x60	H 56	18	(0-360°)	$\sqrt{RGB}$	L2-Hys
Horse	128x80	W 96	9	(0-180°)	RGB	L1-Sqrt
Cat	96x56	H 56	9	(0-180°)	RGB	L1-Sqrt
Dog	96x56	H 56	9	(0-180°)	RGB	L1-Sqrt

H and W stand for Height and Width

[2]Meaning the pixel's magnitude is placed into a bin, according to the angle of the gradient

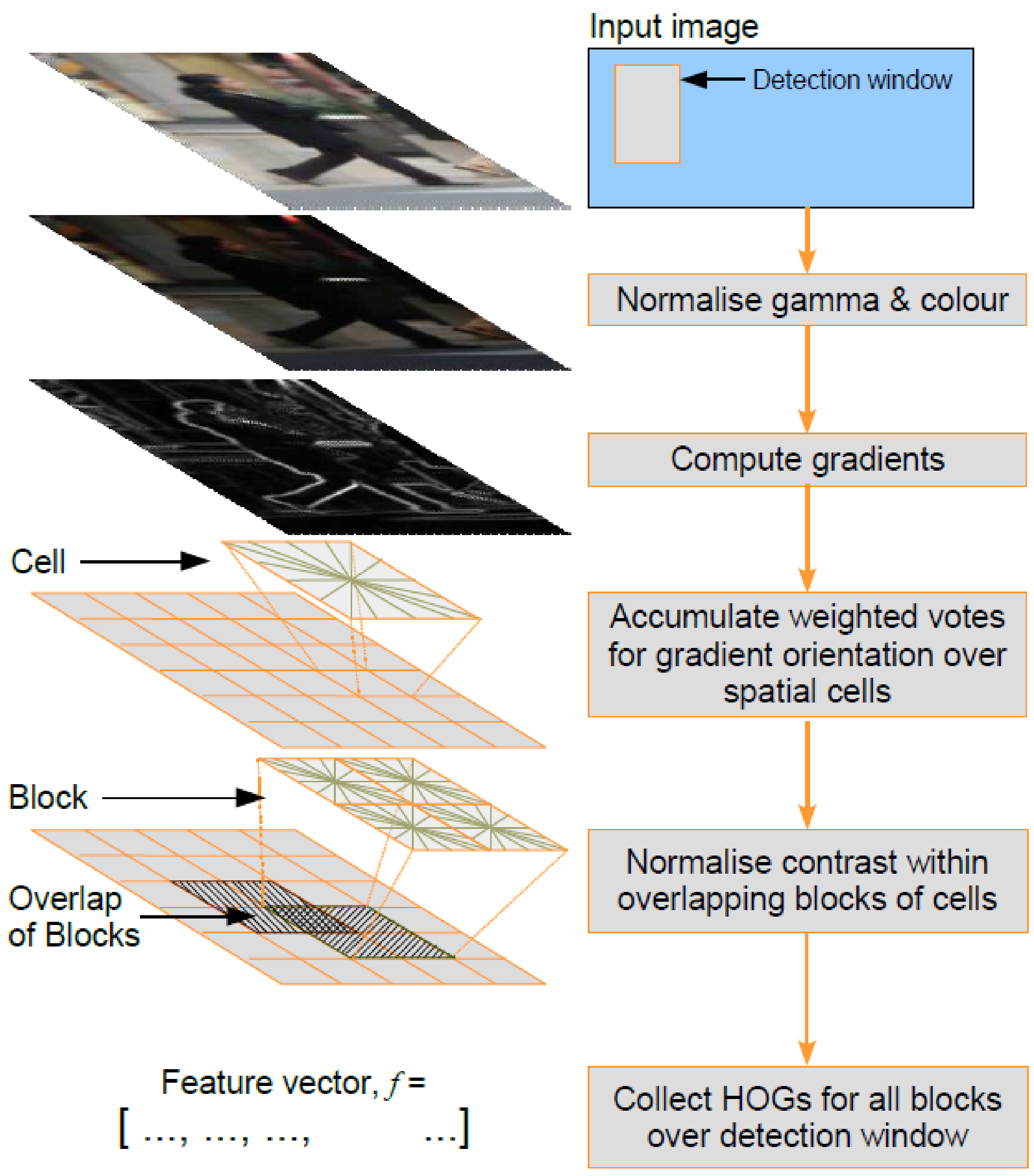

Input image
Detection window
Normalise gamma & colour
Compute gradients
Accumulate weighted votes for gradient orientation over spatial cells
Normalise contrast within overlapping blocks of cells
Collect HOGs for all blocks over detection window
Cell
Block
Overlap of Blocks
Feature vector, f =
[..., ..., ..., ...]

Figure 1.2: A visual representation of the HOG steps [3].

1.2 FHOG

The HOG algorithm suffers from a few limitations. One of which is shown in 1.1: the feature map is specific to an object class, therefore a feature map using human parameters may not work as well for cars or sheep. In other words, HOG feature maps do not capture enough information. FHOG-generated feature maps overcome this problem by merging feature maps that use 9 and 18 bins, and then reducing the size of the output by using a modified version of the Principal Component Analysis. As a result, the output data from the algorithm is slightly smaller than that of HOG, but is more dense in information.

Even though the output is more compact, that does not apply to the core of the algorithm, which is much more complex than HOG. It is perhaps for this reason that no hardware implementation for FHOG has been attempted yet. This work will attempt to do just that. More specifically, it is meant to be a starting point for future FHOG hardware implementations.

1.3 Software of choice: Vivado HLS

Vivado HLS is a tool that translates a C/C++ code into an HDL[3], so that it can be easily loaded into an FPGA[4]. This tool has an incredible potential, as it allows to skip some of the longer steps that characterize a VLSI design, and provide a reasonable -albeit not excessively optimized- implementation of an algorithm. For this reason, HLS is popular both among software engineers who have but basic knowledge of hardware, and among digital designers seeking to test or produce implementations of complex algorithms in a reasonable amount of time, a key resource in the world of electronics. HLS stands for High-level Synthesis and, as the name suggests, allows the designer to work at a higher level of abstraction with respect to an RTL design [8].

While implementing an algorithm with HLS is relatively easy, doing so in an optimized way requires an extensive knowledge of the algorithm, the architecture of an FPGA, and of digital electronics in general.

The FHOG implementation discussed in this work is only a part of the whole object recognition system, which also includes pre-processing and classification. While the author has a decent understanding of these, there are still a lot of arbitrary variables involved, such as the window size and the desired performance considering the system in its whole. By designing the FHOG module with HLS, these variables are but a small inconvenience, since

[3]Hardware Description Language, such as VHDL or Verilog

[4]Field Programmable Gate Array

changing the design to accommodate them is as easy as changing a few lines of code.

To stimulate the interest of the reader, a few famous pictures - processed with an FHOG software - are provided (Figure 1.3). Note that in practice the algorithm is never used this way, but instead processes slices of images at a fixed size.

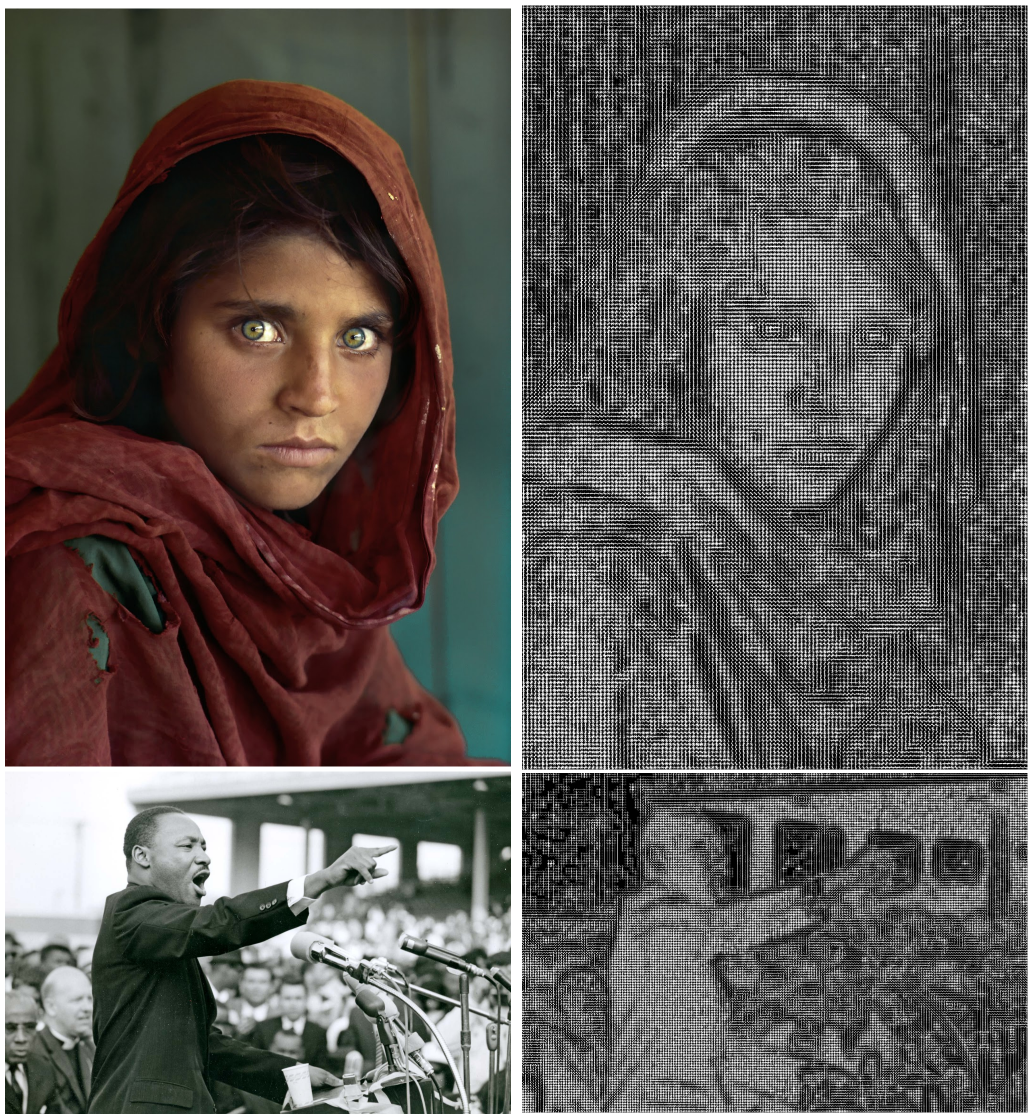


Figure 1.3: Examples of application of the FHOG algorithm. In practice, the FHOG is never used for such large images, but it is applied to a window of fixed size.

Chapter 2

FHOG

2.1 FHOG features

The FHOG and HOG algorithm are based on the same passages, save for a few key differences that allow the FHOG to be suitable for detection of any object class and for multiple parts and mixture models. Previous image detection systems lacked this versatility, and were forced to more specific, single-purpose architectures due to complexity and sheer amount of operations. FHOG manages to address this problem by reducing the size of the feature map drastically, using a simplified but equally efficient version of the Principal Component Analysis.

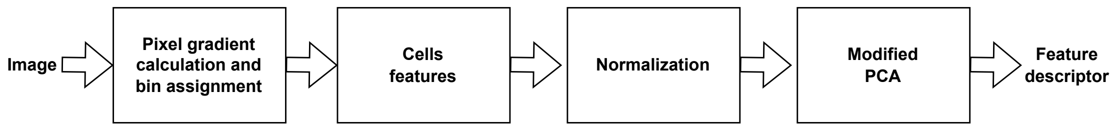


Figure 2.1: Main steps of the FHOG algorithm

The FHOG algorithm is explained thoroughly in this section, the way it is presented in [1]. Most passages are the same as the ones for the HOG algorithm [2], save for a few differences.

2.1.1 Input image

Before diving into the actual algorithm, it is important to know about the image processing that precedes it. FHOG, the way it is presented in [1], is performed by applying a detection window of a fixed size (usually 64x32 pixels) at all locations and scales of an image. The same image is processed several times at different resolutions, and this is obtained by smoothing and subsampling an initial high-resolution image[1].

To improve detection at the borders, padding is performed as well. It consists in adding a layer of pixels (usually 8 or 16) around the image, along both height and width. The added pixels can either be all zeros (zero padding), or they can be a reflection of the pixels close to the borders (reflective padding). Padding will be further discussed in the Normalization step.

2.1.2 Pixel-level features

Given an input image made out of three pixel maps, one for each colour channel, gradient magnitude is calculated for each triplet of pixels at the same position in the different channels. This is done with the use of the [-1,0,1] mask and its transpose, meaning for a given pixel, the gradient in the x direction Gx is obtained by subtracting the pixel to its left from the pixel to its right, and similarly the gradient in the y direction Gy is calculated by subtracting the pixel on its top from the pixel to its bottom. Gradient magnitude is then calculated from these values by using Pythagoras' theorem. This step is illustrated in Figure (2.2).

Once this is done for the three pixels representing the three colours, the one with the highest magnitude is chosen, and its phase is calculated. It is important to note that in many implementations these steps are performed for black and white images only, which only have one channel. This is also the approach followed in algorithm (1).

In some implementations, gradients in the x and y directions are divided by two, according to the definition of derivative. Also, pixels at the border of the image are handled differently, with the so-called *uncentered derivative*, meaning if one of the pixels needed to calculate the gradient is not found, the central pixel is used instead but the division by two is not performed.

[1]This approach may vary according to the detection algorithm. For example fDSST [10] processes the image once at high resolution

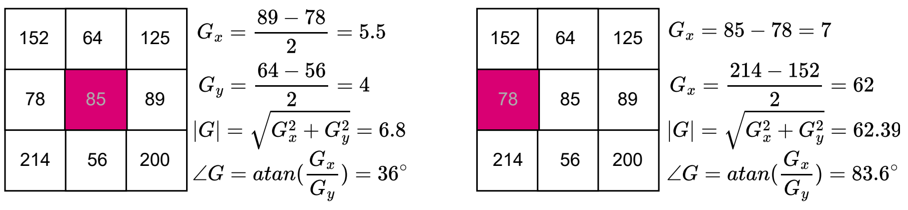

152 64 125
78 85 89
214 56 200
$G_x = \frac{89-78}{2} = 5.5$
$G_y = \frac{64-56}{2} = 4$
$|G| = \sqrt{G_x^2 + G_y^2} = 6.8$
$\angle G = atan(\frac{G_x}{G_y}) = 36°$
152 64 125
78 85 89
214 56 200
$G_x = 85 - 78 = 7$
$G_x = \frac{214-152}{2} = 62$
$|G| = \sqrt{G_x^2 + G_y^2} = 62.39$
$\angle G = atan(\frac{G_x}{G_y}) = 83.6°$

Figure 2.2: Example of pixel gradient calculation. The gradient of pixel 78 in the x direction is represented here as an example of how uncentered derivatives work. Since there's no value at the pixel's left, the gradient is computed as the difference between the pixel at the right and the pixel 78 itself. Division by two is not performed in this case.

Now comes the interesting part: the bin assignment. This can be done in two different ways: either contrast-insensitive or contrast sensitive. Both ways involve the discretization of the phase into "bins" of 20°. By using the contrast sensitive way, gradient magnitudes are assigned into one of 18 bins ranging from 0° to 360°; while in the contrast insensitive approach, they are assigned into one of 9 bins ranging from 0° to 180°; in both cases at 20° steps. This assignment is done by generating an empty vector of 9 or 18 components for each gradient magnitude, and then adding the magnitude at the corresponding bin. Using the phase and magnitude derived in the left part of Figure(2.2), the result would be a sparse vector with the third component (the bin corresponding to 40°) equal to 6.8 (for both contrast approaches). Bin assignment in FHOG is done with equations (2.1) and (2.2), where $p = 9$, $q = 18$, and $\theta(x, y)$ being the phase expressed in radians, and $\theta_{deg}(x, y)$ the one in degrees.

$$B_1(x,y) = round\left(\frac{p\theta(x,y)}{\pi}\right) modp = round\left(\frac{\theta_{deg}(x,y)}{20}\right) mod9 \tag{2.1}$$

$$B_2(x,y) = round\left(\frac{q\theta(x,y)}{2\pi}\right) modq = round\left(\frac{\theta_{deg}(x,y)}{20}\right) mod18 \tag{2.2}$$

Each class of objects is better detected by using one or the other method. As stated in [2]: *"For humans, the wide range of clothing and background colours presumably makes the signs of contrasts uninformative. However note that including sign information does help substantially in some other object recognition tasks, e.g. cars, motorbikes."*. The FHOG algorithm uses both approaches on contrast, thus making it suitable for all object classes.

Algorithm 1: Pixel Level feature maps

```
inputs  : $p(x,y)$, $0 <= x <= h-1$, $0 <= y <= w-1$
outputs: $F(x,y)$ h $x$ w matrices for sensitive features whose entries
         are sparse vectors of size 18.
1  $q = 18$ ;
2  $F = zeros(h,w,q)$ ;
3  for  $x_0 = 2 : w-1$ do
4      for  $y_0 = 2 : h-1$ do
5          $G_x = (p(x0+1,y0) - p(x0-1,y0))/2$ ;
6          $G_y = (p(x0,y0+1) - p(x0,y0-1))/2$ ;
7          $M = \sqrt{{G_x}^2 + {G_y}^2}$ ;
8          $O = atan(\frac{G_y}{G_x})$ ;
9          $B = round\frac{q*O}{2\pi}modq$ ;
10         $F(x0,y0,B) = r(x0,y0)$
```

Notes

- In the algorithm, only internal pixel gradients are computed. Border pixel gradients are computed with the aforementioned uncentered derivatives, but the resulting code would be too long and tedious to show here.
- Table 1.1 clearly shows how bin assignment affects different classes of objects. This further reinforces the use of both bin assignment types in the FHOG algorithm. In the same table, an optimal window size is proposed for each class. This is a problem, since in [1] there's no indication on the window size to be used. For the purpose of this work, the "human window" is going to be used, however keep in mind that other object classes may require different window sizes.

2.1.3 Cell-level features

In this step, pixel features are used to produce cell features. The most basic way to do so consists in grouping pixels into square cells of size k by k, and averaging them, however both FHOG follows a more complex approach: the bilinear interpolation [2]. With this approach, pixels within a *Block*, a square of 2x2 cells, contribute to each of the four cell features in the block, according to vicinity. Blocks overlap too, meaning a single pixel contributes to the nine cells included in the four blocks that encapsulate the pixel's cell.

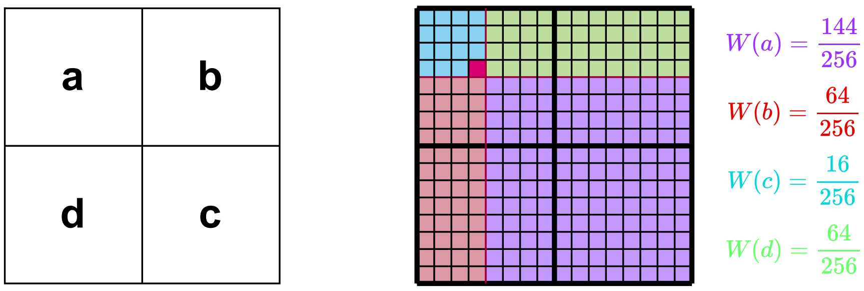


Figure 2.3: The block entity is made out of cells a,b,c,d. Given a pixel inside the block, its contribution to each of the four cells is equal to the pixel feature times the respective weight factor, which is calculated geometrically as the ratio between the highlighted areas and the total area of the block. The sum of all four weight factors is always equal to one.

2.1.4 Normalization

In this step, cell features (both contrast sensitive and contrast insensitive) are normalized with respect to the four blocks that surround it. To do so, for each cell four normalization factors are going to be calculated for each $\gamma, \delta \in \{-1,1\}$ (2.3). Then the cell feature vector is divided by these factors one at a time, and the four resulting vectors are concatenated to obtain a primitive FHOG feature map (2.4), which is a 4x9 or 4x18 matrix for each cell. In the division, a truncation factor $\alpha = 0.2$ is used to make sure the division result for each component of the vectors is never higher than α. This step is computed for the internal cells only, meaning the external cells added with padding are only used to calculate normalization factors.

[2]The HOG algorithm uses instead the so-called trilinear interpolation. The FHOG authors in [1] considered it to be superfluous though. This is the only instance where HOG follows a more complex approach.

Algorithm 2: Cell Level feature Maps

```
inputs  : $F(x, y)$
outputs: $C(x, y)$, $\frac{h}{8}x\frac{w}{8}$ sized matrix whose entries are vectors of size 18
1  C = zeros(h/8, w/8, 18) ;
2  - - Block selection
3  for $i$ = 1:(h/8-1) do
4      for $j$ = 1:(w/8-1) do
5          - - Pixel Selection
6          for $m$ = 1:16 do
7              for $n$ = 1:16 do
8                  $W(a) = (16 - i) * (16 - j)/256$ ;
9                  $W(b) = (16 - i) * j/256$ ;
10                 $W(c) = (i) * (j)/256$ ;
11                 $W(d) = (i) * (16 - j)/256$ ;
12                 $C(i, j, :)+ = W(a). * F((i - 1) * 8 + m, (j - 1) * 8 + n, :)$ ;
13                 $C(i, j + 1, :)+ = W(b). * F((i - 1) * 8 + m, (j - 1) * 8 + n, :)$ ;
14                 $C(i + 1, j + 1, :)+ = W(c). * F((i - 1) * 8 + m, (j - 1) * 8 + n, :)$
                    ;
15                 $C(i + 1, j, :)+ = W(d). * F((i - 1) * 8 + m, (j - 1) * 8 + n, :)$ ;
```

Since both square roots and divisions are performed, this could be one of the more computationally heavy steps. Fortunately, the normalization factors are the same for contrast sensitive and contrast insensitive features. They simply are calculated from contrast insensitive features and are used in both cases. The equations below are as presented in [1].

$$N_{\gamma,\delta}(x,y) = \sqrt{||C_1(x_0,y_0)||^2 + ||C_1(x_0+\gamma,y_0)||^2 + ||C_1(x_0,y_0+\delta)||^2 + ||C_1(x_0+\gamma,y_0+\delta)||^2} \tag{2.3}$$

$$H(x,y) = \begin{pmatrix} T_\alpha(C(x,y)/N_{-1,-1}(x,y)) \\ T_\alpha(C(x,y)/N_{+1,-1}(x,y)) \\ T_\alpha(C(x,y)/N_{+1,+1}(x,y)) \\ T_\alpha(C(x,y)/N_{-1,+1}(x,y)) \end{pmatrix} \tag{2.4}$$

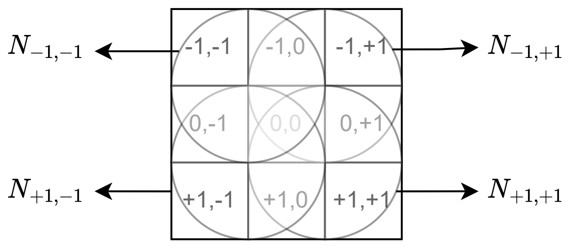


Figure 2.4: Visual representation of which cells are hit by each Normalization Factor

This step is slightly differently in HOG [3], where rather than using cell descriptors, the author uses *Block Descriptors*, or concatenations of the four cell descriptors included in a 2x2 square of cells. Blocks are then divided by their normalization factors. With the block notation, equation 2.3 can also be expressed as in 2.5. In FHOG, both blocks and cells are needed, which makes padding compulsory to make sure border cells are correctly normalized. Both in HOG and FHOG, a small factor ϵ is added to N to avoid division by zero.

$$N_{\gamma,\delta}(x_0,y_0) = \sqrt{||B(x_0+\gamma,y_0+\delta)||^2 + \epsilon} \tag{2.5}$$

Algorithm 3: Normalization

```
Inputs  : $C(x, y)$
Outputs: $H(x, y), H_2(x, y)$, each entry of these matrices is a 4x9 or 4x18
         sized matrix
1  - - Computation of contrast insensitive features by folding ;
2  C2 = zeros(size(C,1),size(C,2),9) ;
3  C2(:,:,1:9) = C(:,:,1:9) + C(:,:,10:18) ;
4  for $i$ = 1 : size(C,2)/8-1 do
5      for $j$ = 1 : size(C,2)/8-1 do
6          $B(i, j, :) = [C2(i, j, :), C2(i, j + 1, :), C2(i + 1, j + 1, :), C2(i + 1, j, :)]$;
7          $B_{norm}(i, j, :) = \sqrt{||B(i, j, :)||^2 + \epsilon}$ ;
8  - - Initialization of output matrices ;
9  H = zeros(size(C,1)-2,size(C,2)-2,4,18) ;
10 H2 = zeros(size(C,1)-2,size(C,2)-2,4,9) ;
11 for $i$ = 2:(size(C,1)-1) do
12     for $j$ = 2:(size(C,2)-1) do
13         $H2(i - 1, j - 1, :, :) = [min(C2(i, j, :)./B_{norm}(i - 1, j - 1), 0.2); min(C2(i, j, :)./B_{norm}(i - 1, j), 0.2); ...]$;
14         $H(i - 1, j - 1, :, :) = [min(C(i, j, :)./B_{norm}(i - 1, j - 1), 0.2); min(C(i, j, :)./B_{norm}(i - 1, j), 0.2); ...]$;
```

2.1.5 Modified PCA

PCA [18] is an eigenvector based technique used in statistics to reduce the size of large datasets. Performing this technique requires costly projection steps [1]. By applying it many times, Felzenswab and his team found that the resulting eigenvectors (approximately) followed a specific pattern, so they aimed to generate such patterns directly without going through the whole PCA.

In the end, with this technique the size of each $H_1(x_0, y_0)$ is reduced from 36 to 13, and each $H_2(x_0, y_0)$ is reduced from 76 to 22. This is achieved without loss in information. The procedure consists in summing the elements along the column for each orientation bin, and then summing the elements along the row for each normalization type (the latter is only done for contrast insensitive features, leading to 31 components, 9 + 18 + 4, rather than 35). This can be represented as a dot product between H and matrices that only have unitary elements along a line (u_k) or a column(v_k). In the algorithm that follows, we'll simply sum the elements along row or column though. Moreover, the result has a very intuitive meaning: *we'll have nine contrast insensitive orientation features, eighteen contrast sensitive orientation features and four features that reflect the overall gradient energy in different areas around the cell* [1].

2.2 Algorithm Verification

The main issue with the implementation of such a complex algorithm lies with the lack of details provided in most papers, and with the differences that are present in existing software implementations. Before the actual implementation was designed, the algorithm was therefore recreated manually in Matlab, and its output was compared with an already existing tool -the one freely provided by pdollar [6]-. The tool is available to all, and can be downloaded directly from Matlab. The tool is widely used and its author has published several papers on the topic, including [12], [13] and [14]. The verification of the code and the algorithm is done firstly by comparing the output images, and secondly by applying the Matlab script into an already existing FHOG software recognition system and verifying whether it is still able to recognize objects.

Algorithm 4: Modified PCA

```
Inputs  : $H_1(x,y)$, $H_2(x,y)$
Outputs: $G(x,y)$, a matrix containing 31 components for each block
1  $p = 9$ ;
2  $q = 18$ ;
3  – Cell selection for $x_0 = 0$ to $rowsof(H_1) - 1$ do
4      for $y_0 = 0$ to $columnsof(H_1) - 1$ do
5          – computation of last 4 normalization components
6          for $i = 0$ to 3 do
7              for $j = 0$ to $p - 1$ do
8                  $G(x_0, y_0)(p + q - 1 + i) += H_1(x_0, y_0)(i, j)$ ;
9          – computation of 9 contrast insensitive features
10         for $j = 0$ to $p - 1$ do
11             for $i = 0$ to 3 do
12                 $G(x_0, y_0)(j) += H_1(x_0, y_0)(i, j)$ ;
13         – Computation of 18 contrast sensitive features
14         for $j = 0$ to $q - 1$ do
15             for $i = 0$ to 3 do
16                 $G(x_0, y_0)(p + j) += H_2(x_0, y_0)(i, j)$ ;
```

2.2.1 Produced images

Several images representing the FHOG of a test image are shown below. The visual analysis of these histograms provides an immediate, despite unreliable, feedback on how the author's code fares compared to the code provided by the pdollar tool. Despite all passages of the FHOG algorithm have been followed meticulously, initial results would show too bright images. Later on, this was fixed by dividing the final descriptor by two. The author thinks that it was necessary because of the application of several filters in pdollar's implementation -whereas they are not present in mine- and possibly to other unknown arbitrary choices that were done on both sides.

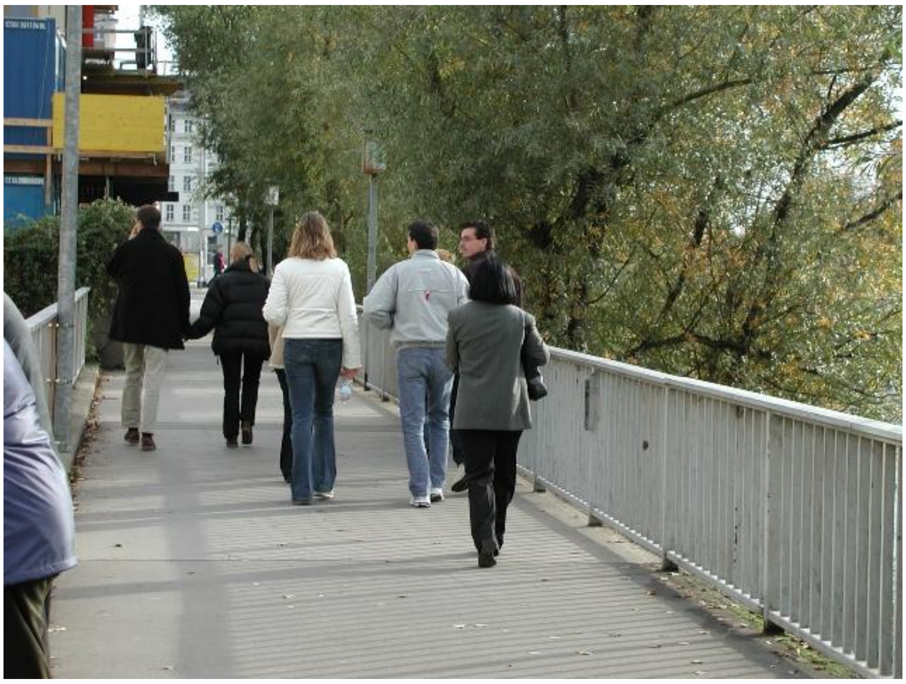
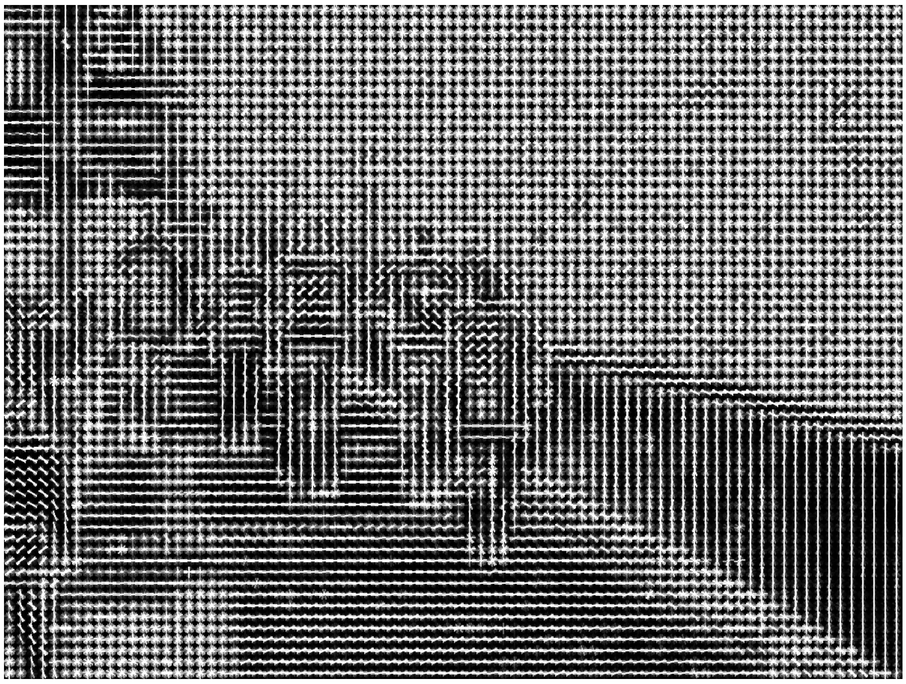


Figure 2.5: Test Image [5]


(a) *FHOG provided by the tool*

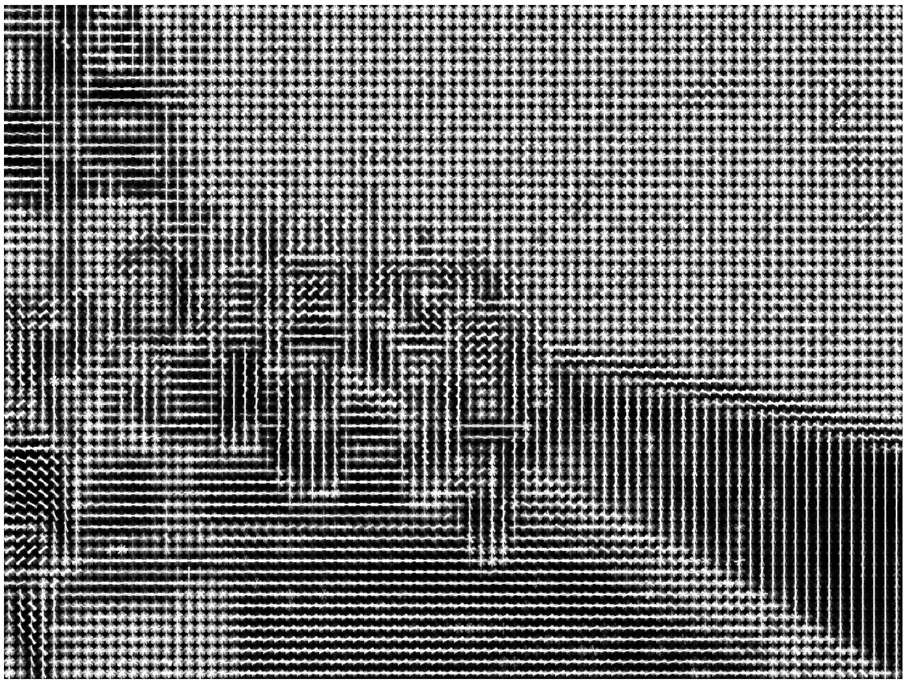

(b) *FHOG produced with my implementation*

Figure 2.6

Chapter 3

Algorithm Optimizations

Now that the algorithm has been analyzed, it is time to do a few considerations on its implementation. In this Chapter, several optimizations for the algorithm are going to be discussed in terms of trade off between speed and precision. Then, their behavior is tested with the help of an object detection software, the fDSST.

The FHOG algorithm is quite heavy, computationally speaking. Fortunately, it is also redundant: several performance optimizations can be done with small losses in terms of precision. Most of these optimizations are the ones proposed by [5] in their HOG implementation. Keep in mind that the objective of this thesis is that of producing a fast FPGA implementation. While software based existing FHOG implementation can already achieve the highest precision, the goal here is to possibly improve performance to a point where images can be processed as fast as the frames of a video without the aid of a high-end computer processor. Precision is therefore not too important. If an object is not recognized in a specific frame, chances are it will be recognized in the next frame. The goal of these optimizations is therefore going to be that of improving performance in terms of latency, throughput and complexity rather than precision.

3.1 Performance evaluation

The impact on detection of the many choices present in this algorithm is discussed in depth in [3]. The unit of measurement is the percentage at 10^{-4} False positives per Window (FPPW). As stated in [3]: *We often quote the performance at* 10^{-4} *False Positives per Window, the maximum false positive* *we consider to be useful for a real detector given that* 10^3 *to* 10^4 *windows are tested for each image.* The error evaluations present in [3] however were made for the HOG algorithm, and therefore do not necessarily have the same value for the FHOG. Their value will therefore be considered more of an indication, and whether the following approximations will still hold will be verified through the fDSST software that will be introduced later in this chapter.

3.2 Black and white input images

As discussed previously, the use of black and white, single channel images rather than colored, three channels images has large benefits in terms of performance with negligible precision loss.

Firstly, magnitude computations are reduced to one third compared to RGB images, where magnitudes are instead computed for each pixel across all three channels. Secondly, the comparison between magnitudes is skipped entirely. In terms of precision loss, moving from RGB to grayscale colors reduces the accuracy only by 1,5% at 10^{-4} False Positives Per Window (FPPW), as shown by Dalal & Triggs.

3.3 Pixel level optimizations

3.3.1 Column skip

In [5], only oddly numbered pixel columns contribute to the cell descriptor. At first glance, this may be too much of an approximation. Also, in [5] it is stated that: *it would just decrease the classification score margin between a pedestrian and non-pedestrian sample.* Their system is however an HOG, where human detection is the sole focus, while FHOG aims to detect all object classes. It will be later demonstrated that this approximation still holds in FHOG, for non-human objects. Skipping cells has a large impact on FHOG, halving the operations required until the cell descriptors are produced.

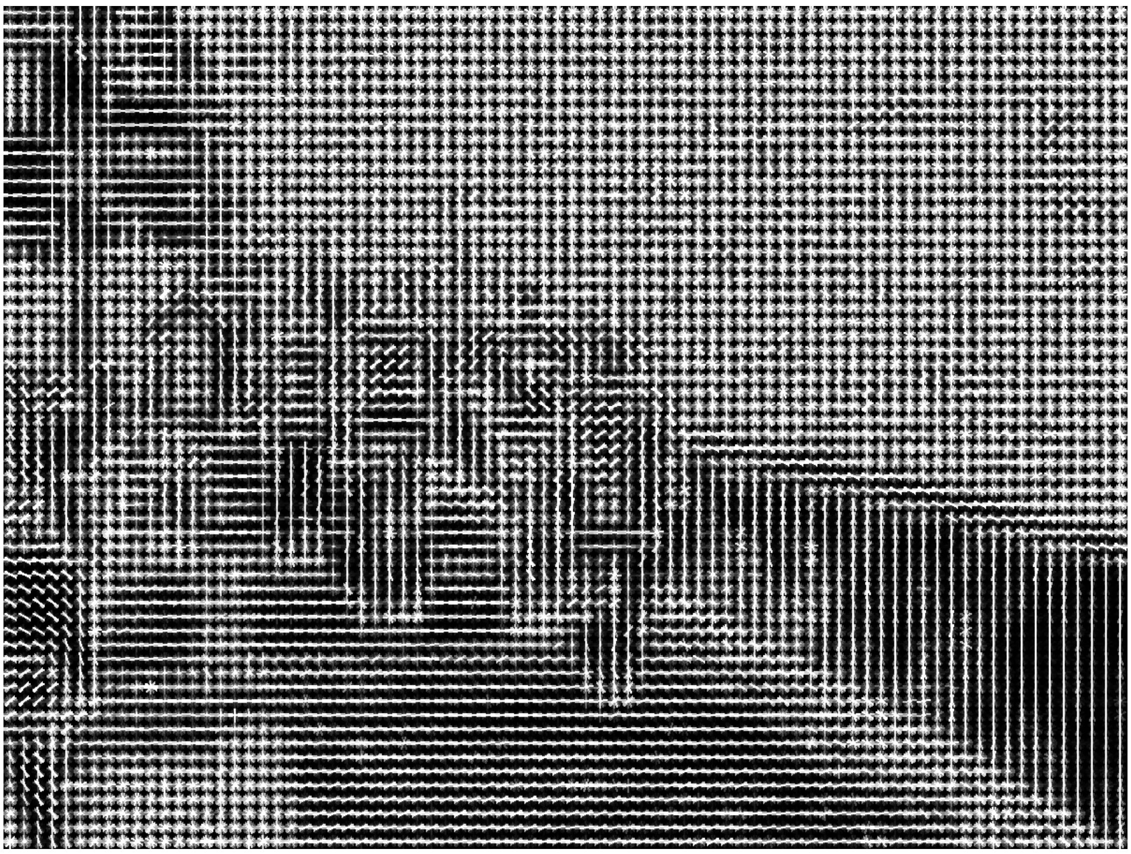


Figure 3.1: Visual FHOG obtained by performing column skip. The human shapes are still recognisable despite having used only half the pixels.

3.3.2 Magnitude calculation with subtraction

Magnitude calculation includes the costly square root operation, which could slow down the algorithm considerably. A possible solution to speed it up is that of computing magnitudes as $|G_x - G_y|$ instead of $\sqrt{G_x^2 + G_y^2}$, as proposed in [5]. Despite at first glance it may look too much of an approximation, doing some tests with the calculator upon the data generated with the MATLAB code showed remarkable results. Alternatively, integer square root algorithms could be used, such as the one proposed in [15].

3.3.3 Phase calculation and bin assignment

Phase calculation includes the arcotangent operation. It is a very costly operation, and digital designer tend to avoid it, especially in FPGAs. Since the phase is discretized into bins, there's no need to have a precise value for the phase. Most implementations ([16],[5],[4],[7]) therefore store the values of the tangents for all bins, and proceed with algorithms similar to algorithm 5, which relies on Look Up Tables (LUTs).

This method is further improved in [7] by approximating the tangent values to fractions. For example, $tan(20°) \approx 4/11$. This would allow to work with integers and avoid floating point arithmetics. In [5], they use LUTs to store the whole multiplication between the tangent and G_x. This allows to skip the multiplications entirely, however 256 entries are required for each tangent value (since G_x varies from 0 to 255). In any case, not all tangents have to be stored: in fact, only those for the first quadrant - from zero to eighty degrees - are required, due to the basic properties of the tangent. Note that for simplicity, the bin assignment in Algorithm 5 is contrast insensitive. To switch to contrast sensitive - which is the case in FHOG - one simply has to create four different cases, one for each quadrant.

3.4 Cell level features

3.4.1 Non Overlap of Blocks

Due to bilinear interpolation and overlapping blocks, the number of operations to be performed in the cell features step is huge. According to [5], the block overlap is not strictly necessary, as it has a minimal impact on the precision of the algorithm (4.4 % at 10^{-4} FPPW). In other words, when performing bilinear interpolation, instead of having each cell contribute to the four blocks around it, each cell would contribute to a block only.

3.4.2 Derivation of Cell Features by Averaging

As mentioned in the previous chapter, the most basic way to derive cell features is that of averaging[1] its pixel features. Should the precision loss be acceptable, this would be a huge increase in terms of performance, reducing the number of operations before cell features by a factor of four.

[1]Since the elements in a cell are a power of two (usually 64), the averaging consists in a sum and a shift, rather than a sum and a division, which would be much costlier.

Algorithm 5: LUT bin assignment

```
inputs  : $G_x(x,y)$ $G_y(x,y)$
outputs: Bin assignment
– Quadrants I and 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
        $bin = 5$
–Quadrants II and IV
else
    if $(|Gy| < tan(20)|Gx|)$ then
        $bin = 9$
    else
        if $(|Gy| < tan(40)|Gx|)$ then
            $bin = 8$
    else
        if $(|Gy| < tan(60)|Gx|)$ then
            $bin = 7$
    else
        if $(|Gy| < tan(80)|Gx|)$ then
            $bin = 6$
    else
        $bin = 5$
```

3.4.3 Derivation of contrast insensitive map

One free optimization that can be done is that of deriving the contrast insensitive feature map from the contrast sensitive one. This can be easily done by summing the last nine components of the contrast sensitive features to the first nine. With this optimization, the amount of operations to be performed before normalization is reduced drastically, and without any loss of information.

3.5 Normalization

There are a few obvious optimizations to be done here. Firstly, in the previous algorithm, cell norms are calculated more than once for each cell. It would make more sense to calculate the needed norms and then proceed to calculate normalization factors. Secondly, the square of a norm is actually more simple than the norm itself. Rather than calculating the square of the norm, it is more optimal to compute the sum of the squared components for each cell feature. Thirdly, and as previously stated, the computation of the square root can be substituted with the computation of its inverse, so as to avoid the subsequent division. In truth however, it would be better to avoid the square root entirely.

3.5.1 Quantized Normalization

In [5], a method called the "quantized normalization" is proposed. With it, each of the 4x9 (or 4x18) elements of the descriptor is quantized into one of eight values, according to the block average. This method (see eq. 3.1) would allow to skip the computation of both the squares and the square root. This method is incredibly effective, however it must be noted that it is created for HOG, where rather than having cell descriptors we have block descriptors, and normalization is done contrast-insensitively only. For this method to work in FHOG as well, a few tweaks have to be performed.

$$normalized\,element = \begin{pmatrix} 0.4,\ if\ element > 2 * block\,average \\ 0.35,\ if\ element > 7 * block\,average/4 \\ 0.3,\ if\ element > 3 * block\,average/2 \\ 0.25,\ if\ element > 5 * block\,average/4 \\ 0.20,\ if\ element > block\,average \\ 0.15,\ if\ element > 3 * block\,average/4 \\ 0.1,\ if\ element > block\,average/2 \\ 0.05,\ if\ element > 1 * block\,average/4 \\ 0,\ else \end{pmatrix} \tag{3.1}$$

The first observation to be done concerning the adaptation of quantized normalization to FHOG is when it comes to the generation of the contrast sensitive features: its average is half of the corresponding contrast insensitive feature's average. This allows to compute the CS block average by simply shifting the CI block average. One might also wonder whether the comparison in eq.(3.1) still holds for CS features, and it does since both average and elements are smaller. Tests performed with MATLAB yielded positive results in terms of precision, but the added logic due to having to work with both contrast sensitive and contrast insensitive features reduced the performance advantages.

3.5.2 Other Norm types

A few papers [4][7], avoid squares by using the L1 - sqrt norm instead of the L2-Hys norm (3.2).

$$v \leftarrow \sqrt{v/(||v||_1 + \epsilon)} \tag{3.2}$$

Compared to the L2- Hys norm, the squares are not computed, and the square root is performed at the end. Also, according to [3], the two norms have an equivalent performance [2] . Alternatively, L1 norm (L1-sqrt norm but without the square root) could be used too, albeit with a loss of performance (5.5% at 10^-4 FPPW). Using L1 norm has several other advantages, which will be discussed in the next chapter. Note that both L1-sqrt and L1 norms do not include clipping.

[2]More precisely, L1 sqrt norm is more suitable for certain object classes, while L2 Hys is more suitable for others. Since the FHOG object detection works for all classes, the two choices seem to be equivalent. See Table 1.1 for more details.

3.6 Optimizations summary

The following table summarizes the optimizations and how they behave in terms of estimated precision loss. N/A stands for Not Available.

Table 3.1: List of optimizations

Optimization	**Section**	**Precision Loss (% at $10^{-4}FPPW$)**
B/W images	Pre-processing	1.5
Subtractions	Pixel features	N/A
LUT tangents	Bin Assignment	none
Column skip	Pixel features	4.1
Non Overlap	Cell Features	4.4
Average	Cell Features	N/A
L1-sqrt Norm	Normalization	none
Quantized norm	Normalization	1
L1 Norm	Normalization	5.5

3.7 Optimizations Verification

Most of the previously discussed optimizations were used only in HOG implementations, and while they should work in FHOG as well, some testing is still required. Visual and numerical comparison is not enough for this task, however a full verification of the algorithm would require a large amount of time and resources since it implies extensive testing on very large datasets, which goes beyond the scope of a master thesis. Therefore it was decided to apply the Matlab script to an already existing FHOG-based object detection software and verifying if it would still work. This is going to be especially useful to verify if the precision loss brought by the optimizations is still acceptable.

3.7.1 Fast Discriminative Scale Space Tracker

One of the more advanced object tracking systems is the Fast Discriminative Scale Space Tracker, theorized by M. Danelljan et al in [10] and implemented in [9]. The goal of this method is to track the movements of an object of which the initial position and shape is known. Difficulties in doing so include orientation changes, as well as shape change due to perspective. While it is not the goal of this thesis to dive into the intricacies of another complex algorithm, suffice it to say that the fDSST employs FHOG for image representation.

The implementation of the fDSST is a MATLAB script, therefore compatible with the script produced in Chapter 2 to verify the full algorithm. Also, it relies the same FHOG function that was used in the visual verification step [6] in Chapter 2. Since the fDSST tool uses an FHOG with variable parameters, the author's code was modified to be able to work with any cell size while keeping the algorithm's structure intact.

The test sequence of the fDSST script is a video of a man holding a plush dog (Figure 3.2), and the objective of the tracker is to keep identifying the plush's face through the video. When substituting the FHOG originally used in the fDSST implementation with the author's, the task of tracking the plush remains successful, however since the code was not optimized for performance, each frame would take a few seconds to process, much more than in the original fDSST code. The processing speed improves drastically when applying some of the optimizations.

Once all of the 1350 frames of the video are processed, the code also provides the user with several metrics in the command window to evaluate performance and precision. These metrics are: Frames per Second, Overlap Precision, Distance Precision, and Center location error. As stated in [10]: *The OP score is computed as the percentage of frames in a video where the intersection over-union overlap with the ground truth exceeds a certain threshold. In the tables we report the OP at a threshold of 0.5, which corresponds to the PASCAL evaluation criterion. The DP score is defined as the percentage of frames in a video where the euclidean distance between the tracking output and ground truth centroids is smaller than a threshold. A threshold of 20 pixels is used in this work.*

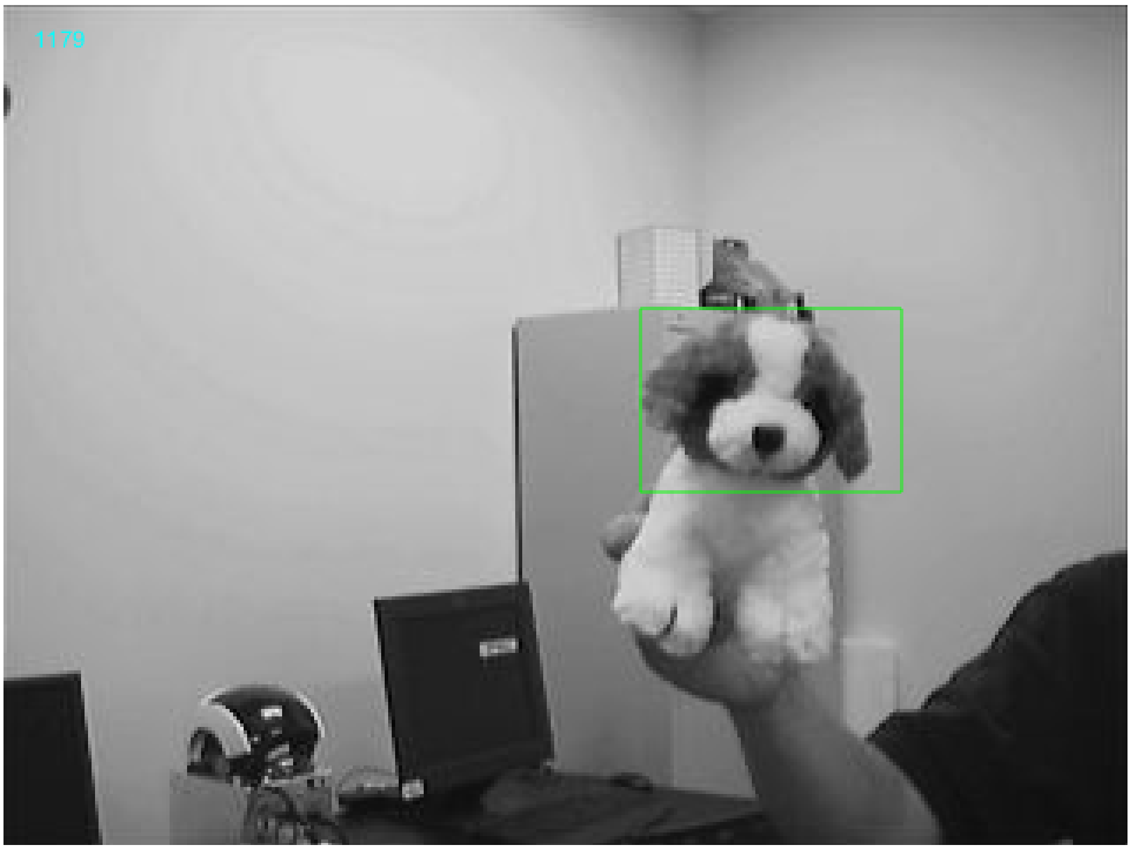


Figure 3.2: A frame of the plush dog sequence. The green frame always follows its face. The number on the top left refers to the frame.

3.7.2 Verifying

In order to test the optimizations, the FHOG script was readapted so that each optimization can be switched on or off. This was done by creating a separate function for each step of the algorithm (such as pixel features, cell features etc...), where the function decides how to perform the step according to a parameter. The parameters for each step are set into the main script, which now only acts as a wrapper for all of the functions. With this change, it is now possible to evaluate each combination of optimizations with ease, as well as adding the possibility to evaluate their performance impact by using MATLAB's profiler[3] (table 3.2).

[3]A built-in tool that measures the time required to complete each function included in a script, as well as the number of calls for each function.

The results of the profiler are however to be taken as an inaccurate indication because of how reliant these timings are on the owner's computer, as well as how MATLAB behaves much differently from a specialized hardware FPGA implementation. Still, by running the profiler on the unaltered FHOG, it immediately appears clear how the computation of cell features is by far the heaviest task. Optimizing this step is therefore going to be vital towards improving the algorithm's performance. Note that the function that calculates the weight factors is called approximately 1.2 million times[4], and with each call, four factors are calculated. Considering that each factor is then multiplied and summed, it can be estimated that the Cell Features step by bilinear interpolation and overlap takes roughly 10 million operations to be completed.

Function	Time (%)
Cell Features	96.65
Pixel Features	2.3
Normalization	0.76
Modified PCA	negligible

Table 3.2: Results provided by MATLAB's profiler on the full algorithm, for the test image.

In Table (3.3), each optimization is evaluated through the fDSST in terms of frame per second and precision. The script itself returns these data by writing them in the command window at the end of a simulation. Due to the high number of possible optimizations, not all combinations have been tried. Instead, the performances are evaluated for the full, unaltered algorithm [5]; and then for each optimization individually applied to the full algorithm. Once this is done, a selection of the best optimizations is evaluated, and will be used as the algorithm to be applied in the FPGA implementation. Note that in all cases, the use of black and white images and folding are applied by default, and the LUT tangents are not verified due to how the method is already proven by several papers and the theory.

[4]The test image is 496x656

[5]As a reminder for the reader, it calculates pixel features by default, cell features with bilinear interpolation and overlapping blocks and normalization with L2-Hys norm

Table 3.3: Evaluation of Optimizations through the fDSST.

Optimization	**Frames Per second**	**Center Location Error**
Original Algorithm*	208	2.9 pixels
Full Algorithm	0.688	2.93 pixels
Column skip	1.28	2.79 pixels
Subtractions	0.67	3.07 pixels
Non Overlap	1.72	2.81 pixels
Average	4.8	2.46 pixels
L1-sqrt Norm	0.681	3.08 pixels
Quantized norm	0.691	2.93 pixels
L1 Norm	0.697	3.41 pixels
Best**	9.01	2.72 pixels

*Meaning the one the fDSST script originally employs [6]. **Includes Cell skip, Subtractions, Average and L1 norm

Table 3.3 provides with some concerning data. Why is the original script that much faster than the one developed in this thesis? Why is it that some optimizations (mostly those for the Normalization) do not improve performance? Why do some of the optimizations improve the Center location Error even when they are supposed to instead lower precision? As to the first question, It must be reminded that pdollar's implementation [6] was made for the sole purpose of being very fast in MATLAB, whereas the script developed in this work serves only to test whether the optimizations work or not, which, according to the simulations, they do. Note that the Overlap and Distance Precision metrics have not been included in the table, and that's because in all of the simulations these values were always 100%. To answer the second question, it first must be pointed out that the Frames per Second metric slightly changes at every simulation, since it is strongly dependant on owner's computer. Also, as shown in Table 3.2, the normalization step takes less than 1% of the computation time, which is why the optimizations for the normalization are not so impactful. Since it is not possible to measure their performance with precision, the Normalization optimizations will be tested again on HLS. As to the third question, the center location error value seems to change slightly at every simulation, so, similarly to the Frames per Second, it is not a reliable metric. If these metrics do not matter, was there a purpose to these simulations, aside from proving that the optimizations work?

While the Frames per Second of the MATLAB script is not reliable data for the purpose of the hardware implementation, it still gives useful indication on how some optimizations affect the algorithm, especially those concerning the cell features, which increase performance by a very large amount, since they drastically reduce the number of operations. At the same time, the Center location Error metric, while not precise, demonstrates how the optimizations have a small impact on precision and do not behave unexpectedly.

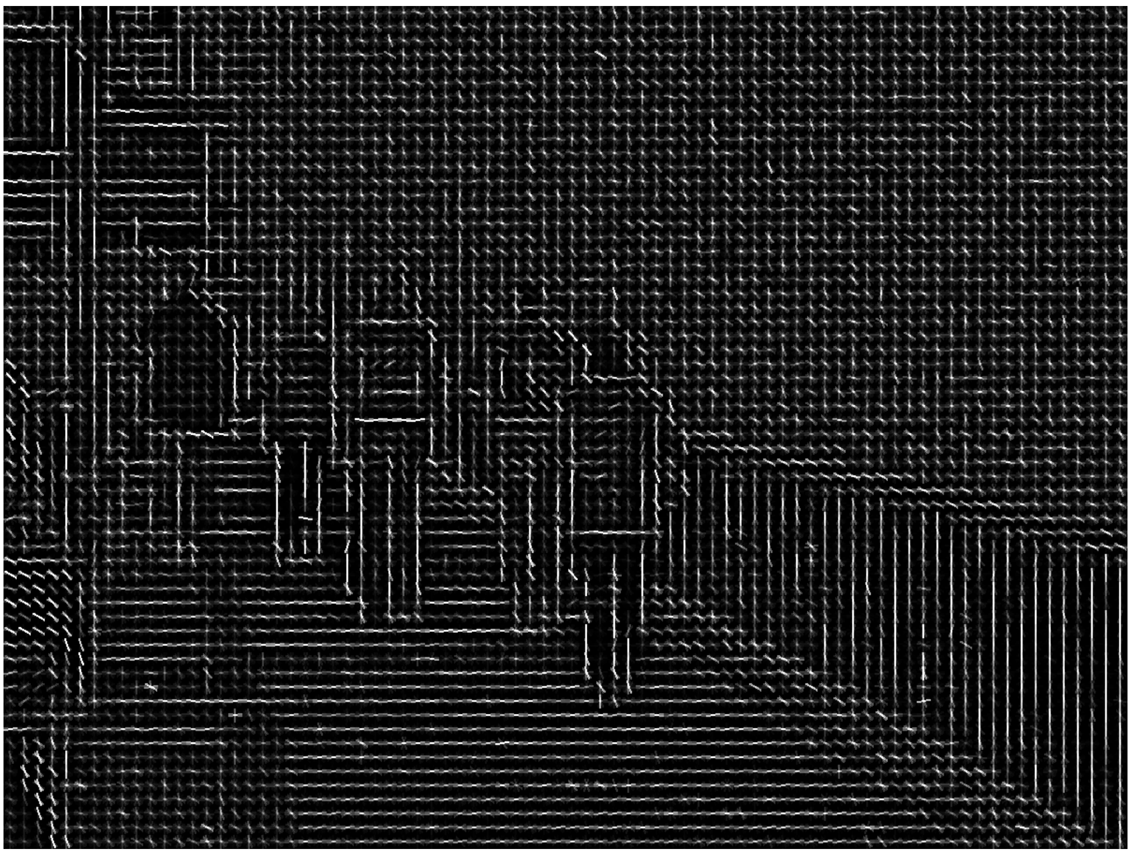
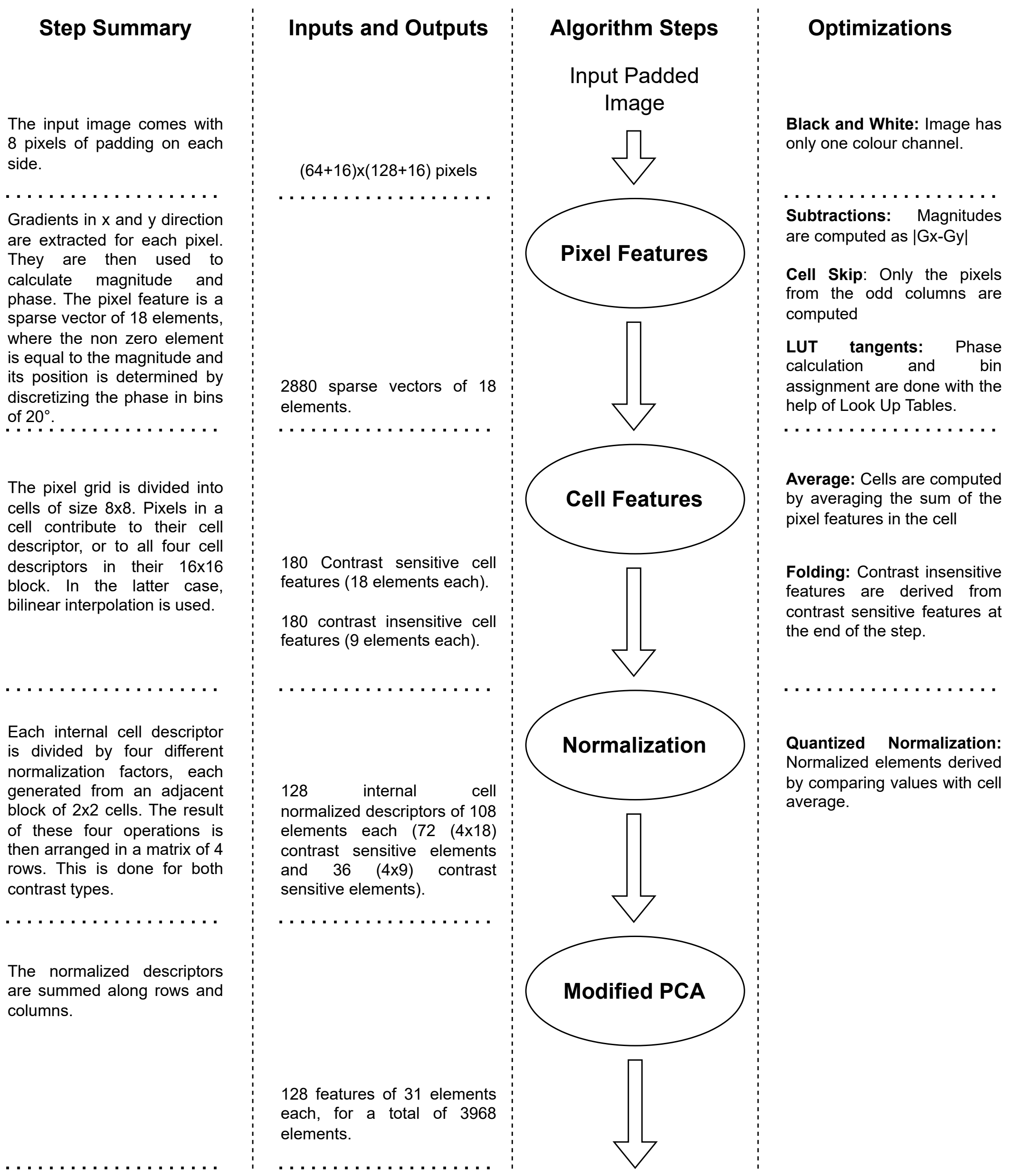


Figure 3.3: Visual output obtained by applying the selection of the best optimizations from Table 3.3 to the test image.

3.8 Algorithm Summary



Step Summary	Inputs and Outputs	Algorithm Steps	Optimizations
The input image comes with 8 pixels of padding on each side.	(64+16)x(128+16) pixels	Input Padded Image	**Black and White:** Image has only one colour channel.
Gradients in x and y direction are extracted for each pixel. They are then used to calculate magnitude and phase. The pixel feature is a sparse vector of 18 elements, where the non zero element is equal to the magnitude and its position is determined by discretizing the phase in bins of 20°.	2880 sparse vectors of 18 elements.	**Pixel Features**	**Subtractions:** Magnitudes are computed as |Gx-Gy| **Cell Skip**: Only the pixels from the odd columns are computed **LUT tangents:** Phase calculation and bin assignment are done with the help of Look Up Tables.
The pixel grid is divided into cells of size 8x8. Pixels in a cell contribute to their cell descriptor, or to all four cell descriptors in their 16x16 block. In the latter case, bilinear interpolation is used.	180 Contrast sensitive cell features (18 elements each). 180 contrast insensitive cell features (9 elements each).	**Cell Features**	**Average:** Cells are computed by averaging the sum of the pixel features in the cell **Folding:** Contrast insensitive features are derived from contrast sensitive features at the end of the step.
Each internal cell descriptor is divided by four different normalization factors, each generated from an adjacent block of 2x2 cells. The result of these four operations is then arranged in a matrix of 4 rows. This is done for both contrast types.	128 internal cell normalized descriptors of 108 elements each (72 (4x18) contrast sensitive elements and 36 (4x9) contrast sensitive elements).	**Normalization**	**Quantized Normalization:** Normalized elements derived by comparing values with cell average.
The normalized descriptors are summed along rows and columns.	128 features of 31 elements each, for a total of 3968 elements.	**Modified PCA**	

Figure 3.4: A summary of the FHOG algorithm, highlighting the various optimizations that will be used in the implementation, as well as the size of inputs and outputs of the various steps.

Chapter 4

HLS implementation

Most of the optimizations that have been analyzed in the previous chapter switch the original operations with simpler ones, in order to lighten the computational load, while at the same time reducing precision. In other words, they change the output of the algorithm, and that's the reason why they needed to be tested extensively. In this chapter more optimizations are present, however they do not change the output of the algorithm but improve performance by adding pipelines, performing operations in parallel and optimizing memory usage.

The goal of this chapter - and of this thesis - is to derive an FPGA architecture capable of executing the FHOG algorithm. This task presents a few challenges given the complex nature of the algorithm, specifically the presence of three and four dimensional matrices; and the large amount of operations to be performed. As a consequence, memory and resource utilization threaten to be high, to the point where it may be possible for them to exceed the FPGA capacity if not handled correctly. It is important to stress again how this implementation is oriented towards high speed and reduced area (meaning resource and memory usage) rather than precision.

4.1 Vivado HLS

The use of High Level Synthesis instead of a more traditional RTL design means that it won't be possible to control every detail of the implementation. Instead, HLS provides the designer with built-in functions, or pragmas, that allow to control some of the more traditional digital electronics design choices such as pipeline depth. Along with these pragmas, HLS provides some custom libraries, that can be used along with the standard C++ libraries to provide the best possible hardware implementations. The main pragmas and libraries are going to be discussed in this section, due to their key role in this work. For a full description, consult [11].

4.1.1 Pragmas

Pragmas, or directives, are lines that allow the designer to directly control some of the more important aspects of an FPGA implementation. The more common ones are presented in the following list.

- **HLS Pipeline:** indicates that a pipeline of the specified depth is to be used.
- **HLS Unroll:** performs unrolling within a loop. In other words, crates several branches that work in parallel to compute the loop. The number of branches can be specified.
- **HLS Dependence:** allows to specify whether there are or not data dependencies within a loop.
- **HLS Loop Flatten:** specifies whether to flatten the loop or not. When the loop is not flattened, the resulting hardware for that part of the code will follow the code structure, thus making debugging easier.
- **HLS Array Partition:** partitions large arrays into multiple smaller arrays or into individual registers, to allow parallel access to data.

4.1.2 HLS libraries

When coding with HLS, it is best to use the functions included in the HLS libraries, since they better translate into hardware when compared to their C++ counterparts. Some of the more important HLS libraries are summarized In the following list.

- **hls_math.h:** this library is meant to be a replacement of the more traditional math.h library commonly used in C++ to perform arithmetical operations. While the functions provided by both libraries are the same, using the HLS library makes sure that the operations are implemented in a more optimized way.

- **hls_video.h:** this library provides several tools that are useful in image processing. Among them, the line buffer allows to optimize window based operations and filtering of images. The functions provided by this library allow to create such buffers and load them during the process with the appropriate data. Due to the high memory usage of the algorithm, this library will be key in improving its performance.

- **hls_stream.h:** with this library it is possible to treat large data as stream interfaces, by using FIFO memories. It is a common practice to use stream interfaces in image and video processing, as it is the case for FHOG.

- **ap_int.h:** this library allows the user to define custom precision integer[1] data types (ap stands for Arbitrary Precision) by specifying the number of bits to be used. Choosing the precision correctly can have positive effects both on memory usage and performance.

4.2 Inputs and Outputs

In order to build the FHOG module, the first step is to correctly define inputs and outputs. For the purpose of the implementations, it was decided to use the standard "human detection window", which is the one more commonly used in HOG implementations. While FHOG is capable of working with any type of window, it was decided to work with the standard window for the purpose of comparing performances. As for the input, the module receives a black and white image of size 64x128 pixels (excluding padding), which translates into eight cells horizontally and sixteen cells vertically, for a total of 128 cells and 105 overlapping blocks. The output is going to be an array of thirty two elements per cell, for a total of 4096 elements.

The input needs further discussion due to padding. While padding can be avoided in HOG implementations, even though it is not recommended, it is unavoidable in FHOG due to how normalization works: it in fact requires to have four blocks around each cell of the original image, meaning without padding it wouldn't be possible to process cells on the border. There's also the issue of gradient calculation for border pixels, which in theory are handled with uncentered derivatives, but in practice would be better to handle with

[1]Similarly, the ap_fixed.h library allows to define custom floating point types.

centered derivatives in order to process all pixels the same way and avoid the use of additional logic. As a consequence, the input image will be padded with a reflective cell on all sides, plus a pixel on all sides. The input will therefore be a 82x146 image. With these numbers, the number of blocks is 153 and the cells are 180, but the output remains the same. In this work, cells from the original image will be referred to as "internal cells".

4.3 Architecture

In [5], which is an HLS implementation of the HOG algorithm, the authors chose to control the implementation in a very direct way, thus not exploiting the HLS libraries a lot. Still, they managed to achieve a very respectable architecture, and it was possible because of the relative simplicity of the HOG algorithm. As already stated numerous times, the FHOG adds complexity in several ways: firstly, it uses both contrast sensitive and insensitive feature maps; secondly, it is cell-based rather than block-based [2]; lastly it adds the modified PCA step. Therefore while this work uses many of the optimizations present in [5], it will have to rely on the more advanced functionalities provided by HLS, namely the use of data streams and line buffers, to reduce memory use and improve concurrency, and heavy use of pipelining and unrolling.

The use of buffers to store intermediate data naturally leads to a division of the architecture into stages. It was decided to go with three stages working in sequence, according to the diagram in Fig 4.1.

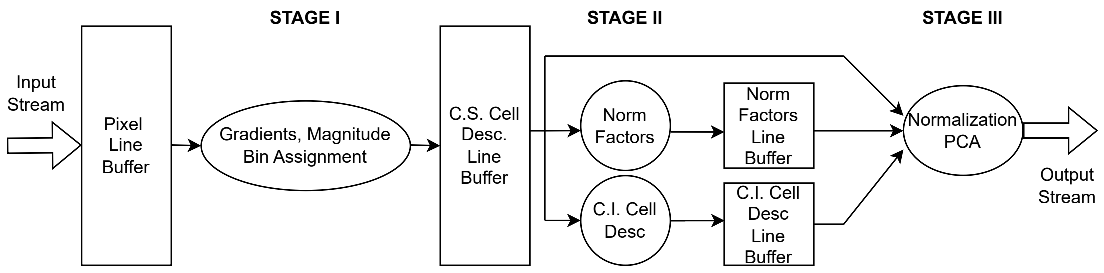


Figure 4.1: Block diagram of the FHOG architecture.

Why streams and line buffers? Data streams are commonly used in image

[2]More precisely, it needs both cell and block descriptors, while HOG can work with blocks only

and video processing, where large amount of data is serialized. In order to begin processing, the FHOG algorithm does not need the whole input image, but only a part of it, which is read from the stream and stored into the Pixel Buffer as needed. Having the input stored in a large memory, on the other hand, would have slowed the execution considerably. The stream interface is also very beneficial in the output, since any block reading it does not have to wait for the whole image to be processed.

4.3.1 First Stage

Pixel gradients, bin assignment and contrast sensitive cell features have been merged into one step, which is the first stage. In other words, each pixel is processed and then fed to the cell descriptor without storing intermediate variables. Both pipelining and unrolling have been applied to this step by the use of pragmas, and of course, all of the optimizations that were discussed in the previous chapter are used here: column skip, magnitudes as subtractions and LUT tangents.

The input image and the output contrast sensitive cell descriptor are stored in line buffers. The input buffer is three lines deep and as wide as the image plus the padding. Once the first three lines are loaded into the buffer, the gradients, bins and magnitude for the middle line are computed and the corresponding cell descriptor is incremented by the magnitude at the bin position. At this point, the input buffer is shifted up, discarding the top line of pixels and adding a new line at the bottom. The process continues until eight lines of pixels are processed, at which point the contrast sensitive cell descriptors for the first line of cells are ready and written to the bottom of the output buffer. The output buffer contains three lines of cell descriptors. Once the output buffer is filled for the first time, the second stage begins to process its content.

4.3.2 Second Stage

In the second stage, normalization factors and contrast insensitive feature map are calculated. In order to avoid the quantized normalization's complex logic, it was decided to use the L1 norm (without square root) to generate the normalization factors. Since the division is a costly operation in hardware, rather than using the original factors, their inverse is used instead, so that one division is performed for each block instead of four. During normalization (performed in stage III), cell descriptors are in fact multiplied by the inverse of the normalization factors.

The contrast insensitive cell descriptor branch operates on the second line of the input buffer, skipping the first and last cell. The normalization factors branch uses instead the first three lines of the buffer to generate the normalization factors for the first two lines of blocks[3] (the ones required to normalize the cells from the second line). This is only true for the first iteration, since the next iterations only compute one line of normalization factors from the second and third line of the input buffer. The output buffers in this stage are two. One for the contrast insensitive cell descriptor, one line deep; and one for the Normalization factors, two lines deep. The input buffer containing contrast sensitive cell descriptors is carried over to the third stage.

This stage is done so that the third stage receives all the data it needs to proceed with normalization and PCA for a line of cells. Hopefully the timing diagram in Figure4.3 can clarify any doubts the reader may have.

4.3.3 Third Stage

The third stage receives in input both cell descriptor buffers and the normalization factors buffer. It performs normalization and PCA. In order to reduce the size of intermediate variables, the outputs for this stage are produced bin by bin rather than cell by cell. Also, as soon as an element of the output is ready, it is sent directly into the output stream.

4.3.4 Buffer sizes

The size of all buffers should further clarify their behavior, and is reported in table 4.1. Their size in terms of Bytes (or bits) was calculated as well.

The total memory use of buffers amounts to 10.878 bits, or 1360 Bytes. The amount of memory that is going to be used in the full architecture is going to be much higher than that since it will also include input and output FIFOs to handle the streams, several internal variables and constants, and the LUTs for tangent computation. Note that in most cases HLS completely

[3]Normally, normalization factors are extracted from the contrast insensitive features. In this case, using the L1 norm allows to extract normalization factors from contrast sensitive features as well.

Table 4.1: Memory size of Buffers

Buffer	**Lines**	**Columns**	**Bits per element**	**Size**
Pixels	3	82	8	246B
C.S. Cell Desc.	3	10x18*	14	945B
C.I. Cell Desc.	1	8x9*	14	126B
Norm Factors	2	9	15	270b

*Number of cells times the number of bins.

partitions these buffers, implementing them in such a way that all the data can be accessed in parallel.

4.3.5 Data Flow

The three stages work sequentially, as per Figure 4.2. Due to the size of the input image, sixteen cycles are required for the full output. At each cycle, the FHOG descriptor for a line of eight cells is produced. Internally, the first and second stage are programmed to fill the interface buffers at the first cycle, as illustrated in the timing diagram in Figure 4.3.

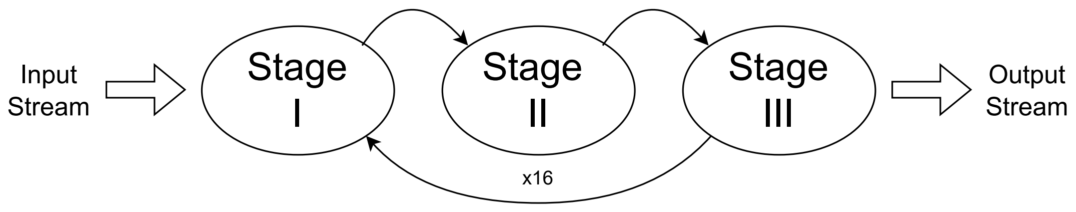


Figure 4.2: State Diagram

4.3.6 Timing Diagram

The behavior of the system is more accurately represented by the timing diagram in Figure 4.3. This is not a traditional timing diagram: there are no clock nor control signals and the width of the signals and data is not proportional to their duration. In order to follow the timing diagram, a few explanations are in order:

Stage 1: the content of the data bubbles in the first stage refer to the data in the stage's output buffer. The numbers represent the corresponding line of cells from the padded image. A number on the data bubble means that the corresponding line of cells is being written at the bottom of the buffer.

Stage 2: Due to how the second stage has multiple outputs, there are two or more numbers in each data bubble. The ones at the left of the slash represent the lines of blocks, while the number on the right represents the lines of cells. The latter are less than the ones from the first stage because they are only the internal cells.

Stage 3: the numbers in the bubbles refer to the internal cells' FHOG descriptor. Since the output of the this stage has been implemented as a stream, the output is available as soon as it's processed.

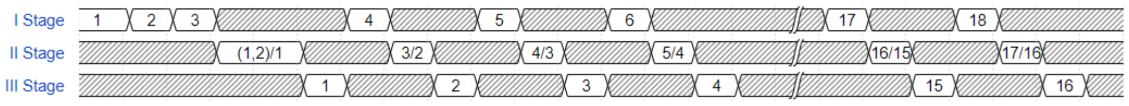


Figure 4.3: Timing Diagram

4.3.7 Control Signals and interface

Vivado HLS provides the architectures with a standard set of control signals, a clock, and the controls required to handle the stream interfaces. Table 4.2, which was adapted from the synthesis report provided by Vivado, represents these signals, their format and their direction (either inputs or outputs). These signals are extremely common in digital architectures and require no further explanation.

4.3.8 Number Format

Aside from the advantages in terms of performance, several of the algorithm optimizations from the previous chapter also bring additional benefits. Calculating magnitudes as subtractions makes it so that the magnitudes are integers; and using the L1 norm makes it possible to avoid averaging when determining cell descriptors. [4]

[4]That is because the L1 norm is proportional to the non normalized vector, whereas L2-Hys and L1 sqrt norms aren't.

Table 4.2: Vivado Interface Report

RTL ports	**Direction**	**Bits**
Clock	in	1
Reset	in	1
Start	in	1
Done	out	1
Idle	out	1
ready	out	1
Pixel Stream	in	8
Pixel Stream Empty	in	1
Pixel Stream Read	out	1
Output Stream	out	20
Output Stream Full	in	1
Output Stream Write	out	1

Understanding the dynamic of the variables allows to represent them with the proper number of bits and the right data type, to improve performance and memory usage. While in C++ integer types are implemented with 32 bits, in this algorithm we can very easily work with less bits. In this section, the data type and bits number is assigned to the main variables, exploiting the arbitrary precision libraries provided by Vivado HLS. In Table 4.3 the main variables' ranges and chosen data types are listed.

Table 4.3: Data Types for the main FHOG variables

Variable Name	**Min Value/Max value**	**Number Format**
Pixels	0/255	8 bits unsigned integer
Gradients	-255/255	9 bits signed integer
Magnitudes	0/510	9 bits unsigned integer
Cell Descriptors*	$0/\approx 2^{14}-1$	14 bits unsigned integer
Norm. Factors	0/1	21 bits fixed point **
Output FHOG	0/1	21 bits fixed point **

*Refers to the content of each bin. ** 20 bits for the fractional part, unsigned.

The ranges for each of the variable in Table 4.3 have been determined as follows:

- **Pixels:** Pixels are always represented with eight bits, with values ranging from 0 (black) to 255 (white).
- **Gradients:** Gradients are determined as subtractions between pixel values, therefore they range from -255 $(0 - 255)$ to 255 $(255 - 0)$.
- **Magnitudes:** Due to the choice of calculating them with the absolute value of the subtraction, magnitudes range from 0 $(|0-0|)$ to 510 $(|255-(-255)|)$.
- **Cell Descriptor:** Assuming that all magnitudes in a cell not only have the highest possible value, but also contribute to the same bin [5], and considering that due to the column skip 32 pixels per cell are processed, the content of each bin ranges from 0 to $510 \cdot 32$ (which is slightly lower than 2^{14}).
- **Norm. Factors and Output FHOG:** The format was decided after thorough experimentation, by comparing obtained results with expected results. Initially, 14 bits were used, but it was not enough. With 20 bits the results are very close to those displayed by MATLAB. The maximum for the normalization factor has been set to one, and it is used when the cell descriptor only has zeros. Since one over zero diverges to infinity, it was decided to use one instead as maximum value in this unlikely occurrence.

4.4 Performance results

Let's now analyze the results obtained with the RTL synthesizer, and compare results with previous HOG [6] implementations. These results were obtained by selecting a Virtex7 FPGA.

The values in table 4.5 are expressed in minimum to maximum to take in account that stages one and two perform more steps in the first iteration. The total latency may be misleading though. The maximum is calculated

[5]This is never going to happen, but the assumption is made anyway just to be safe. A more extensive study on the subject could maybe lead to use one or two less bits

[6]Due to the absence of FHOG hardware implementations.

Table 4.4: Clock Frequency

Clock	**Target**	**Estimated**	**Uncertainty**
-	10 ns	8.497ns	1.25ns

Table 4.5: Latency

Module	**Latency (Cycles)**	**Latency (Absolute)**
STAGE I	3075-9386	30.750-93.860 μs
STAGE II	299-950	2.990-9.500 μs
STAGE III	321-321	3.210-3.210 μs
Total**	60048-171440	0.600-1.714 ms

In each field are indicated the minimum and maximum values. **The total includes the buffer loading and the 16 iterations of the three stages.

as the sum of the maximums of all three stages times sixteen. A more accurate estimate would evaluate the total latency by summing the maximum latency for the three stage with the sum of the minimums times fifteen. That would lead to a total latency of around 66000 cycles, or 0.66ms (assuming a 10ns clock). With that in mind, operating frequency (100MHz) and latency (0.7ms), are enough to process more than a thousand 64x128 images per second, which should be enough in most applications.

As for the resource usage (Tab 4.8), it is much lower than anticipated. In theory, it should be possible to load as many as twenty five FHOG modules in the the same FPGA. In any case, there is more than enough space left on the FPGA to hold whatever other modules before and after the FHOG. The resource usage also reveals that modifying the module to be able to process much larger images should indeed be possible as well.

4.4.1 Comparison with previous Architectures

There are several FPGA HOG implementations in literature, which will be briefly presented in this section and compared to the implementation developed for this work.

Comparing these architectures with the one developed in this work is not entirely fair, for a number of reasons. First, they are implementations of

Table 4.6: Resource Usage

Module	**BRAM 18K**	**DSP48E**	**FF**	**LUT**
STAGE I	4	0	891	2105
STAGE II	0	0	3399	9656
STAGE III	0	6	516	552
Total	4	6	4806	12313
Available	2060	2800	607200	303600

Table 4.7: Performance and Resources of HOG architectures (I)

	Platform	**Resolution**	**fps**	**LUTs**	**Registers**
[19]	Stratix II	640 x 480	30	37940	66990
[20]	Virtex-5	320 x 240	62	17383	2181
[21]	Cyclone IV	800 x 600	72	34403	23247
[7]	Virtex-5	1920 x 1080	64	5188	5176

Table 4.8: Performance and Resources of HOG architectures (II)

	DSPs	**Memory (kBit)**	**MHz**	**Windows per sec**
[19]	120	n/d	127	56466
[20]	n/d	1327	44	95480
[21]	68	348	76	401760
[7]	49	1188	270	1789440

a different algorithm, secondly they process larger images, and thirdly they include pre processing and classification. Having these data can be however useful as a benchmark for future, more complete, object detectors or trackers that employ one or more FHOG modules.

Chapter 5

Conclusion and future Improvements

The objective of this work is to serve as future reference, or as a starting point, to develop full FHOG based detection systems. For this reason, this chapter provides a few suggestions on how to improve the existing implementation.

5.1 Performance and resources improvements

As it is evident from Table 4.5, the first stage, where pixels are processed up to the cell descriptors, is the longest task by far. Should anyone want to speed computation up, it would be best to start there. For example, in the current implementation, the main loop in the first stage was pipelined but not unrolled. That is because there is a data dependency which prevents from using the unroll pragma: when calculating the gradient in the x direction, adjacent pixels require the same surrounding pixels to be accessed. There are several ways to circumvent this problem, such as splitting the pixel buffer in two (or more) buffers. Each buffer would then be used to process half the pixels, effectively halving the latency of the first stage. Of course, this would come at a cost in terms of resources.

In the case of unrolling, another data dependency may rise, regarding the LUTs used for the binning process. This can be avoided by either replicating the LUTs, or using the approach of [7], where the tangent value is stored as two integer values.

Stage 2, as evident from Table 4.8, uses a very large amount of flip flops and LUTs. A thorough analysis of the stage would surely lead to a reduction.

The saved space could then be used to load more modules into the FPGA, or to further unroll the first stage. Stage 2 can also be rearranged: currently, the normalization factors are calculated from contrast sensitive features[1]. This choice was done to allow the contrast insensitive cell descriptors calculation to work in parallel to the normalization factors block. The operations required by the normalization factors could be halved by restoring the proper order of the algorithm, with a possible cost in terms of latency.

5.2 Precision and Testing

The FHOG implementation uses several optimizations that greatly reduce the precision of the algorithm. While it was proven that such precision loss is not an issue for the fDSST algorithm, it may be the case for other object detection algorithms. Should anyone use this work for other algorithms, it would be recommended to test the optimizations more extensively. Object detection algorithms are commonly tested on the INRIA datasets, which were used for both HOG and FHOG. The MATLAB provided with this work can serve once again as a tool for testing.

5.3 Conclusion

In conclusion, the developed implementation of the FHOG algorithm was successful both in providing a functioning FPGA implementation, and in giving the tools necessary to adapt it for future, application specific, implementations.

[1]Normally, they are calculated from the smaller contrast insensitive features instead, but thanks to the use of L1 norm it was possible to use contrast sensitive features.

Bibliography

[1] Pedro F. Felzenszwalb, *Object Detection with Discriminatively Trained Part-Based Models*, IEEE Transactions on pattern Analysis and Machine Intelligence, Vol. 32, No. 9, Sep 2010.

[2] N. Dalal and B. Triggs, *Histograms of Oriented Gradients for Human Detection*, Proc. IEEE Conf. Computer Vision and Pattern Recognition, 2005.

[3] Navneet Dalal, *Finding People in Images and Videos*, Human-Computer Interaction [cs.HC]. Institut National Polytechnique de Grenoble - INPG, 2006. English.

[4] W. Walid, M. Awais, A.Ahmed, M. Martina, G. Masera, *Real-time implementation of fast discriminative scale space tracking algorithm*, Journal of Real-Time Image Processing (2021).

[5] K. Nikolaos, P. Nikolaos *Real time HOG implementation*, University of Thessaly Department of Electrical and Computer Engineering Volos, Greece. (2018)

[6] Pdollar's FHOG:
`https://github.com/pdollar/toolbox/blob/master/channels/fhog.m`

[7] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, K. Doll *FPGA-based Real-Time Pedestrian Detection on High-Resolution Images* University of Applied Sciences Aschaffenburg, Germany, Otto-von-Guericke University Magdeburg, Germany.

[8] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer *Parallel Programming for FPGAs* (2018).

[9] fDSST github repository:
`https://github.com/C2H5OHlife/fDSST/tree/master`

[10] Martin Danelljan, Gustav Hager, Fahad Shahbaz Khan and Michael Felsberg *Discriminative Scale Space Tracking* (2017).

[11] Vivado HLS user guide:

https://docs.amd.com/v/u/2019.2-English/ug902-vivado-high-level-synthesis

[12] Piotr Dollár, Ron Appel, and Wolf Kienzle *Crosstalk Cascades for Frame-Rate Pedestrian Detection* Microsoft Research Redmond, California Institute of Technology (2012).

[13] Piotr Dollár, Zhuowen Tu, Pietro Perona, Serge Belongie *Integral Channel Features* (2009).

[14] Piotr Dollár, Pietro Perona, Serge Belongie *The Fastest Pedestrian Detector in the West* (2010).

[15] T. Sutikno *An Optimized Square Root Algorithm for Implementation in FPGA Hardware* (2010).

[16] Sebastian Bauer, Ulrich Brunsmann, Stefan Schlotterbeck-Macht Faculty of Engineering Aschaffenburg University of Applied Sciences, Aschaffenburg, Germany *FPGA Implementation of a HOG-based Pedestrian Recognition System* (2009).

[17] Simple explanation of HOG:
https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/

[18] Andrzej Maćkiewicz, Waldemar Ratajczak *Principal components analysis (PCA)* Computers & Geosciences Volume 19, Issue 3, March 1993, Pages 303-342.

[19] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. *Hardware Architecture for HOG Feature Extraction.* Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, pages 1330–1333, Spetember 2009.

[20] K. Negi, K. Dohi, Y. Shibata, and K. Oguri.*Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm.* 2011 International Conference on Field-Programmable Technology, pages 1–8, December 2011.

[21] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto.*Architectural Study of HOG Feature Extraction Processor for Real-Time Object Detection.* 2012 IEEE Workshop on Signal Processing Systems, pages 197– 202, October 2012.

Acknowledgements

Gli anni passati al politecnico sono stati decisamente faticosi, ma con pazienza e testardaggine, ho infine terminato questo percorso. In questi anni non sono mai stato solo, e questa laurea la devo non solo ai miei sforzi, ma anche al supporto incondizionato delle bellissime persone che mi sono state vicine in questi anni, a cui dedico questo spazio.

Lo dedico innanzitutto agli amici di Roma, insostituibili amici da una vita. Tra sessioni di giochi da tavolo, avventure tra le montagne o semplicemente una chiacchierata davanti a una birra, siete sempre stati lì per me, condividendo preziosi momenti di leggerezza o di confidenza.

Agli amici di Torino, senza i quali il mio soggiorno e le mie trasferte a Torino sarebbero state senz'altro più grigie e monotone. Mi avete fatto sentire a casa mentre ero lontano da casa.

Al professor Martina, che con competenza, prontezza e cortesia ha risposto ai miei dubbi e mi ha indicato la strada da seguire durante il lavoro di tesi; e a Walid, che ha seguito il mio lavoro passo passo nonostante la distanza geografica che ci separava.

Infine, alla mia famiglia: ai miei nonni, ai miei zii, ai miei cugini, a mio fratello e a mia sorella, e a mamma e papà. Per l'affetto che ci ha sempre legato, e per la fiducia che avete sempre avuto in me.