Real-Time Pedestrian Detection on a Xilinx Zynq using the HOG Algorithm

Jens Rettkowski, Andrew Boutros, Diana Göhringer
Application-Specific Multi-Core Architectures (MCA) Group
Ruhr-University Bochum
jens.rettkowski@rub.de, andrew.boutros@student.guc.edu.eg, diana.goehringer@rub.de

***Abstract*— Advanced driver assistance systems (ADAS) are the key to enable autonomous cars in the near future. One important task for autonomous cars is to detect pedestrians reliably in real-time. The HOG algorithm is one of the best algorithms for this task; however it is very compute intensive. To fulfill the real-time requirements for high resolution images an efficient parallel implementation is necessary. This paper presents an efficient hardware implementation as well as a parallel software implementation of the HOG algorithm for pedestrian detection on a Xilinx Zynq SoC. The hardware implementation achieves a speedup of 2x compared to the parallel software implementation for high resolution images (1920 x 1080). Against state-of-the-art a speedup of 1.32x is achieved. The hardware implementation has a reliable detection rate of 90.2% using a classifier trained by an AdaBoost algorithm and a minor false positive rate of 4 %.**

Keywords—HOG algorithm; Pedestrian Detection; FPGA; Xilinx Zynq; Image Processing; Real-Ttime

I. Introduction

Traffic accidents, involving pedestrians, have in most cases severe consequences since pedestrians do not have high protection at the time of impact with a vehicle. Therefore, they are exposed to high risk even at low speed impacts. Studies conducted by the National Highway Traffic Safety Administration at the U.S. Department of Transportation showed that 4,735 pedestrians were killed in traffic accidents and crashes in the year 2013. Additionally, an estimate of 66,000 injuries was also reported in the United States [1]. To sum up, a pedestrian was killed every 2 hours and another was injured every 8 minutes in traffic crashes by calculating an average. The reasons behind these accidents are the complexity of the driving scene in urban areas and the continuous motion of all the scene elements. Moreover, the vehicle driver will never be able to have a 360°-view around the car. There are always blind spots that cannot be observed by drivers. Even if the driver was able to detect the location of pedestrians without missing anything in the scene, a slow reaction time of the driver might still result in collisions.

In order to overcome these issues, advanced driver assistance systems (ADAS) are developed. They increase the road traffic safety besides the car safety by warning the driver when critical situations are detected. Furthermore, such systems can also control the vehicle to avoid collisions in addition to warnings. Therefore the need of reliable and real-time pedestrian detection systems has become inevitable.

To enable a reliable pedestrian detection, robust visual object recognition based on a feature set for humans is required. It has been shown by [2] that grids of Histograms of Oriented Gradients (HOG) descriptors outperform other feature sets for human detection. These feature sets from the HOG algorithm can be classified reliable into human or no human. However, the HOG algorithm is very compute intensive. To enable a real-time pedestrian detection this algorithm can benefit from architectures such as Field Programmable Gate Arrays (FPGAs).

The main contribution of this paper is an efficient hardware implementation of a pedestrian detection for FPGAs using the HOG algorithm. A reliable AdaBoost classifier is used to determine a human based on the obtained features. The implementation fulfills the real-time requirements even for high resolution 1080P video streams. This is enabled by optimizations of selected steps from the HOG algorithm for FPGAs. Consequently, the resource utilization is improved in addition to the performance. The resource and performance results are determined for a Xilinx Zynq chip that contains an ARM processor besides an FPGA. Besides the hardware implementation for FPGAs, a parallel software implementation of the HOG algorithm has been developed using OpenCV for the ARM processor. This system has been improved to increase the frame processing time by a factor of 249. Comparing the optimized software to the hardware approach shows a speedup of 2x regarding the FPGA implementation.

The paper is organized as follows: In Section 0 related work is presented. The individual steps of the HOG algorithm and their hardware implementation are explained in Section III. The software implementation is explained in Section IV. In Section V the hardware implementation of the HOG algorithm is compared against the software implementation. Finally, Section VI concludes this paper and gives an outlook to future work.

II. Related Work

Since Dalal and Triggs published the HOG algorithm [2] in 2005, many researchers investigated in efficiently mapping this algorithm on accelerators, such as FPGAs, DSPs and GPUs. In [3] a real-time pedestrian detection technique using the HOG algorithm for embedded driver assistance systems is presented. The proposed techniques are implemented on a digital signal processor (DSP) resulting in a processing time of 8 fps for a 640x480 image size. The work presented in this paper achieves 40 fps for high-resolution images. An example of heterogeneous systems consisting of a multicore CPU, a GPU and an FPGA for pedestrian detection was presented by Bauer et al.[4]. The multi-core CPU managed the FPGA-GPU pipeline, while the FPGA was responsible for the feature detection and the GPU

TABLE I. COMPARISON OF DIFFERENT FPGA IMPLEMENTATIONS OF THE HOG ALGORITHM

	FPGA	**Frame Size**	**Classifier**	**Max. Frequ. (MHz)**	**Detection Rate**	**False Pos. Rate**	**FPS**	**Pixels per Seconds (PPS)** Frame Size x FPS	**Resources**			
									Slices	***LUTs***	***DSPs***	***BRAMs***
Negi et al.[7]	Xilinx Virtex-5	320x240	AdaBoost	44.85	96.6%	20.7%	112	$8.6x10^6$	2181	17383	-	36
Xie et al.[8]	Xilinx Spartan-3e	320x240	SVM	67.75	98.03%	1%	293	$22.5x10^6$	2041	3379	-	6
Kadota et al.[9]	Altera Stratix II	640x480	-	127.49	-	-	30	$9.2x10^6$	-	3794	12	-
Mizuno et al.[10]	Altera Cyclone IV	1920x1080	SVM	76.2	-	-	30	$62.2x10^6$	-	34403	68	-
Ma et al.[11]	Xilinx Virtex-6	1600x1200	SVM	150	-	-	10.41	$19.9x10^6$	22%	39%	53%	22%
Proposed Approach	Xilinx xc7z020	1920x1080	AdaBoost	82.2	90.2%	4%	40	$82.9x10^6$	5942	21297	4	0

for the SVM classification. An implementation of the HOG algorithm on a 0.13µm CMOS standard cell library consisting of 1,046,807 gates with a maximum frequency of 200 MHz is developed in [5]. However, this system achieves a lower frame processing time of 33.38 fps for a 640x480 image size in contrast to the work presented in this paper. Another interesting approach is presented in [6]. An IP core called IPPro is implemented on a ZedBoard. This IP core achieves a high frame processing rate. However, only the first steps of the HOG algorithm are implemented.

Negi et al. [7] introduced a hardware implementation of the HOG algorithm with an AdaBoost classifier on a Xilinx Virtex-5 VLX-50 FPGA. Empirical evaluation of their system showed a 96.6% detection rate and a 20.7% false positive rate. The maximum frequency of the system was 44.85 MHz.Xie et al. [8] presented a binarization- based implementation of the HOG algorithm along with a linear Support Vector Machine (SVM) classifier implemented on a low-end Xilinx Spartan-3e FPGA. Their system showed a detection accuracy of 98.03% with 1% false positives. A maximum frequency of 67.75MHz was achieved. Both implementations [6] and [8] were for a frame size of 320x240 pixels.

In order to optimize the algorithm for an FPGA implementation, methods to simplify the computation were proposed by Kadota et al. [9]. They implemented the feature extraction part without a classifier on an Altera Stratix II FPGA using Verilog-HDL. The maximum frequency for the implementation of the simplified feature extraction part was 127.49 MHz for a frame size of 640x480 pixels.

Mizuno et al. [10] were able to implement an optimized HOG algorithm with cell-based scanning along with simultaneous SVM classifier with a maximum frequency of 76.2 MHz for HDTV 1920x1080 frames on an Altera Cyclone IV EP4CE115 FPGA.

In contrast to the presented related work, this paper contributes a combination of a block normalization stage that uses only shift operation and a binarization stage. This feature reduces the amount of the required hardware resources.

Moreover, an evaluation of high-throughput fixed-point object detection systems on FPGAs is presented. The HOG algorithm with an SVM classifier is implemented on a Xilinx Virtex-6 LX760 FPGA to detect pedestrians in different scales. For an image size of 1600x1200 pixels 10.41 fps are achieved which results in approximately $20•10^6$ pixels that are processed in 1 second. The work presented in this paper is not scale invariant. However, it achieves 4x more pixels that are processed in 1 second.

Table 1 shows a summary and comparison of the different HOG FPGA implementations. The resource utilization of [7], [8] and [9] is lower due to the smaller frame size. In order to compare the frames per seconds (FPS) between the different frame sizes, pixels per second is used as metric. It gives the number of pixels that are processed in one second. It is calculated by multiplying the number of pixels for a single frame by the FPS rate. The implementation presented in this paper shows the highest number of pixels that are processed in one second. It shows a speedup of 1.32x and a lower resource utilization in comparison to Mizuno et al. [10].

III. HOG ALGORITHM IMPLEMENTED IN HARDWARE

The HOG algorithm is a feature detection algorithm frequently used in computer vision to detect humans. An abstract overview of the HOG algorithm implemented in hardware is given by Fig. 1. The first step of the algorithm calculates the luminance value for each pixel in case of a colored image. In case of grayscale images, this step is excluded. Afterwards, the HOG descriptor is computed by dividing the image in 8x8 pixels that are called *cells*. In each cell, the two gradient components in the x- and y- directions are determined. By means of these gradients components, the gradient magnitude and direction is computed for each of the 64 pixels inside a cell. Subsequently, a histogram of gradient directions is constructed. A normalization of these histograms results in better invariance to changes in illumination. In order to avoid floating point numbers, as they require a lot of FPGA hardware resources, a binarization step is used to optimize the algorithm for an FPGA implementation. A classifier using this descriptor can distinguish between human and non-human.

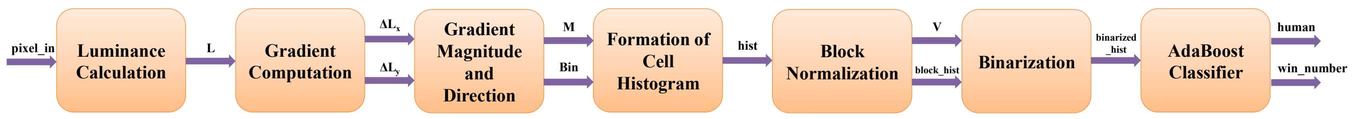

Fig. 1 Overview of the blocks for the HOG algorithm implemented in hardware

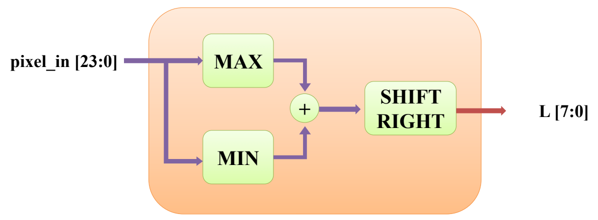

Fig. 2 Block diagram of the luminance calculation module

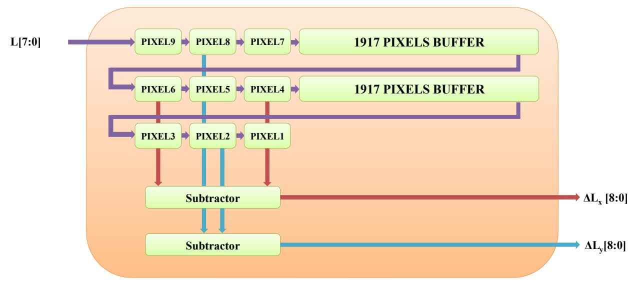

Fig. 3 Gradient computation module consisting of row buffers and subtractors

A. Luminance Calculation Block

The Luminance Calculation block, as shown in Fig. 2, is a module that calculates approximately the luminance value L by the equation:

$$\mathrm{L} = \frac{\max(\mathrm{R,G,B}) + \min(\mathrm{R,G,B})}{2} \tag{1}$$

This module has a 24-bit input signal *pixel_in* that consists of three 8-bit color components R, G and B for each pixel. Since each color component has a value range from 0 to 255, the maximum output luminance value *L* is 255. Accordingly, the output signal *L* has a bit width of 8 bits. The input signal *pixel_in* is processed by a series of comparator blocks to determine the maximum and the minimum components. Afterwards, the result is added up. The division by two is performed by a shift right block to generate the output *L*. This signal is the input for the gradient computation.

B. Gradient Computation

The gradient computation at pixel position (x, y) in the horizontal $\Delta L_x(x, y)$ and vertical directions $\Delta L_y(x, y)$ is conducted by following equations:

$$\Delta L_x(x,y) = L(x+1,y) - L(x-1,y) \tag{2}$$

$$\Delta L_y(x,y) = L(x,y+1) - L(x,y-1) \tag{3}$$

The gradient of the pixel (x, y) is calculated by luminance values *L(x,y)* of pixels that surround pixel (x, y). The luminance values *L(x,y)* are forwarded line-by-line of the image as a stream. Therefore, row buffers are needed in order to buffer two image rows and three pixels of a third row. This results in an architecture as shown in Fig. 3. The output signals $\Delta L_x(x, y)$ and $\Delta L_y(x, y)$ have a range from -255 to 255 and a bit width of 9 bits.

C. Gradient Magnitude and Direction for Bin Assignment

The third step of the algorithm is to calculate the gradient magnitude *M(x, y)* and direction θ(x, y) using the following equations:

$$M(x,y) = \sqrt{\Delta L_x(x,y)^2 + \Delta L_y(x,y)^2} \tag{4}$$

$$\theta(x,y) = \arctan\left(\frac{\Delta L_y(x,y)}{\Delta L_x(x,y)}\right) \tag{5}$$

Equation (4) is implemented using a Look-Up Table as a 2-dimensional vector with the inputs $\Delta L_x(x, y)$ and $\Delta L_y(x, y)$ and the output M(x, y). The values of $\Delta L_x(x, y)$ and $\Delta L_y(x, y)$ range from -255 to 255. Since these values are squared in equation (4), the negative numbers can be neglected. The highest value of the gradient magnitude is $\sqrt{255^2 + 255^2} = 360$.

Equation (5) calculates the gradient direction that is needed to assign the magnitude to histogram bins in the next stage. Due to redundant information, it is sufficient to analyze only from -π/2 until +π/2. This range is divided into 8 bins for the associated gradient magnitude. In order to implement this equation efficiently in hardware, the equation is simplified based on [7]. For example, if a gradient direction θ has to be assigned to bin 7, it should satisfy the following inequality:

$$56.25° < \theta < 78.75° \tag{6}$$

A substitution of θ with equation (5) gives:

$$56.25° < \arctan\left(\frac{\Delta L_y(x,y)}{\Delta L_x(x,y)}\right) < 78.75° \tag{7}$$

Inequation (7) is equivalent to the following inequations with tan(56.25%) ≈1.496 and tan(78.75°)≈5.027 as constant values:

$$\tan(56.25°) < \frac{\Delta L_y(x,y)}{\Delta L_x(x,y)} < \tan(78.75°) \tag{8}$$

$$1.496 < \frac{\Delta L_y(x,y)}{\Delta L_x(x,y)} < 5.027 \tag{9}$$

By multiplying the inequality with 1024 $\Delta L_x(x, y)$:

$$1533\,\Delta L_x(x, y) < 1024\,\Delta L_y(x, y) < 5148\,\Delta L_x(x, y) \quad (10)$$

Subsequently, the inequality contains only integer multiplication and simple comparisons instead of divisions. By deriving the appropriate expressions for all eight bins, conditional expressions are used to assign the magnitudes to bins. The block diagram of this module is shown in Fig. 4. It has two inputs ΔL_x and ΔL_y calculated in the previous stage. It also has two outputs: the gradient magnitude M that has a maximum value of 360 and thus can be represented in 9 bits and the bin assignment *Bin* that has a value from 0 to 7. Thus, it can be represented in 3 bits.

D. *Formation of Cell Histogram*

The next step of the algorithm is the generation of a histogram for groups of 8x8 pixels. A group of 8x8 pixels is called a cell. The block diagram of this module is shown in Fig. 5. Partial histograms are calculated for each row of a cell by the partial histogram generator. Subsequently, the eight resulting partial histograms are summed up to form the final histogram of the cell. This requires seven row buffers of partial histograms. The maximum value of a histogram bin occurs when all 8x8 pixels have a maximum magnitude of 360 and are assigned to the same bin. Accordingly, the maximum value of a histogram is 8 x 8 x 360 = 23,040 which can be represented in 15 bits. Therefore, the resulting histogram output *hist* of each cell is represented in 8 bins x 15 bits = 120 bits.

E. *Normalization*

In this step of the algorithm, the resulting cell histogram from the previous step is normalized with respect to neighboring cells that form a larger building block called block. Each block consists of 2x2 cells. This normalization step is performed using equation (11):

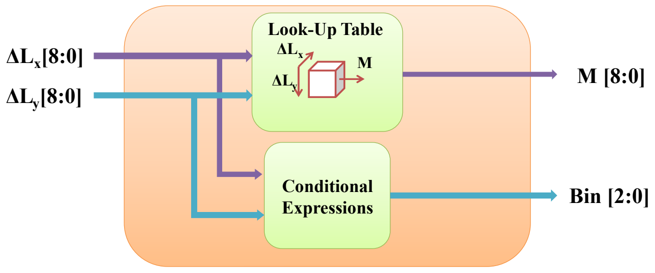

Fig. 4 Gradient magnitude and bin assignment module block diagram

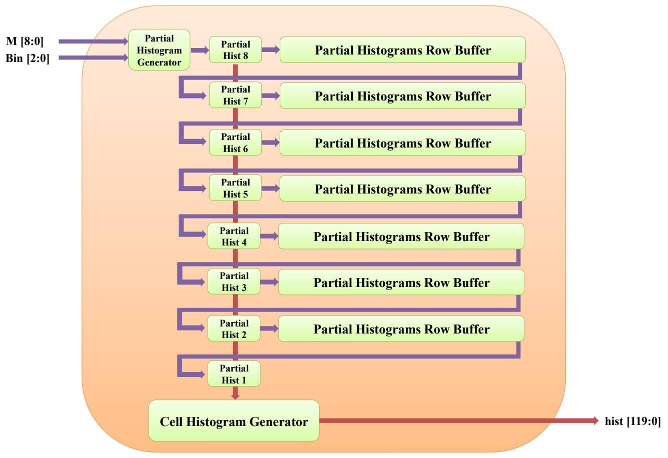

Fig. 5 Histogram generation module block diagram

$$v = \frac{v_k}{\sqrt{\|v_k\|^2 + \varepsilon}} \qquad (11)$$

Given that ν is the normalized histogram, ν_k is the vector corresponding to the combined histograms of 4 cells in a block, $\|\nu_k\|$ is the summation of all elements of ν_k and ε is a constant to avoid a zero enumerator. This equation consumes a lot of resources in hardware. Negi et al. [7] simplifies the normalization step into different shift operations for sub-intervals. This paper presents another approach using only one shift operation without sub-intervals.

Following this, equation (11) can be simplified since ε is very small in comparison to the summation of all histogram elements in a block $\|v_k\|$. Therefore, the equation is reduced to:

$$v = \frac{v_k}{\|v_k\|} \qquad (12)$$

This is the division of each element of the block combined histogram by the summation of all histogram elements of the block to result in a normalized histogram. This expression contains a division. Subsequently, it will always result in decimals and floating point numbers which are complicated and consume a lot of FPGA resources. In order to solve this, another step called Binarizaton is added to the original HOG algorithm. In this step, a certain threshold is specified. If the value of each element of ν is greater than this threshold, it is considered as logic '1', else it is considered as logic '0' as it is presented in equation (13). The histogram binarizaton step is shown in Fig. 6.

$$v = \begin{cases} 0, & \dfrac{v_k}{\|v_k\|} < threshold \\ 1, & \dfrac{v_k}{\|v_k\|} \geq threshold \end{cases} \qquad (13)$$

In the work of Negi et al.[7], a binarization threshold of 0.08 is tested. However, this will also lead to floating point arithmetic and a resource consuming division. Therefore, this threshold is taken as 8/128 instead of 8/100 since the first can be seen as a logic shift right four times. This step is considered to be one of the most important optimizations of the original HOG algorithm to be implemented in hardware. It has the advantage to change each 14-bit value of v_k to a 1-bit binary value which reduces the size of the block histogram to 4 cells x 8 bits = 32 bits. It also helps getting rid of the division operation of the normalization step and replacing it with a logic shift operation and a comparator.

F. Classifier

The AdaBoost algorithm was first introduced in 1995 by Freund and Schapire [12]. It is based on the idea of creating a strong and accurate classifier by combining together several weak and inaccurate classifiers. For example, if it is required to classify fruits into two groups: apples and other fruit. The weak classifiers for this operation can be that apples are circular, apples are red or green or yellow and apples have a stem at the top. If only one of those rules is used, the resulting classification is inaccurate. However, combining all weak classifiers leads to an accurate classifier. AdaBoost classifiers are most suitable when dealing with dense features sets which is the case in the HOG algorithm. Additionally, it can be implemented using block descriptor buffers to form the detection window, a set of comparators and adders.

In order to train the AdaBoost, a Matlab implementation of the presented hardware is developed that extracts the features. The pedestrian detection is evaluated with frames captured from a car simulator. The car simulator is from the company FOERST [13]. According to that, the size of the detection window was chosen to be 136x280 pixels which is the typical size for pedestrians in these frames. The detection window is composed of 16x34 blocks. Therefore, the descriptor of each detection window is a vector of 16 x 34 x 32 = 17,408 dimensions.

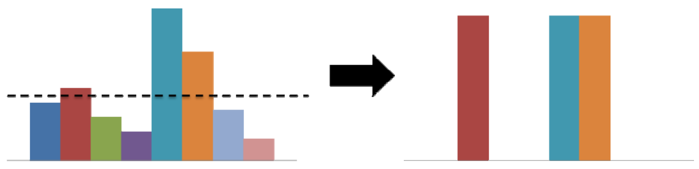

Fig. 6 Histogram binarization

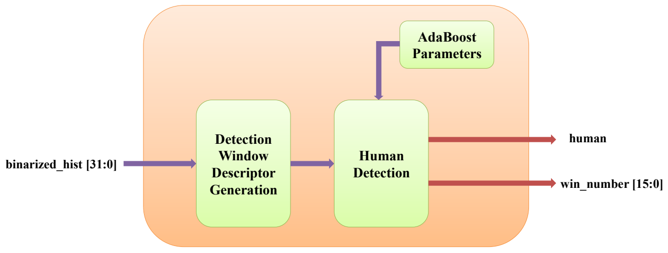

Fig. 7 AdaBoost classifier block diagram

Two hundred positive samples of humans and three hundred negative samples of non-humans obtained from both INRIA pedestrian database [14] and NICTA pedestrian database [15] were used to train the classifier with 500 iterations. 41 weak classifiers are obtained. Each of the weak classifiers checks a specific dimension of the binarized descriptor of a complete detection window. Subsequently, it adds a specific either positive or negative weight depending on the binarized value and the classifier. If the summation of all these positive or negative weights is a positive number, then the detection window represents a human. If it is a negative number, then the detection window does not represent a human. In Fig. 8, an overview of the classifier is given. The signal *human* specifies the detection of a human. The appropriate detection window is given by the 16-bit signal *win_number*. The *Detection Window Descriptor Generation* block receives the binarized histogram. This is forwarded through buffers to generate the detection window of 16x34 blocks.

IV. HOG Algorithm Implemented in Software

In addition to the hardware implementation of the HOG algorithm, a software approach using OpenCV 2.4.1 has been implemented. This approach is built on the ARM processor of the Zedboard that has installed Linaro 12.09 as operating system. The OpenCV library offers an implementation of the Dalal and Triggs HOG algorithm along with a pre-trained SVM classifier for human detection. This function is used to detect pedestrians in 1920x1080 input frames.

A VDMA core stores an HDMI input stream via the High Performance port (HP) into the DDR memory. The HDMI stream that shows a driving scene with pedestrians is given by the car simulator [13]. The ARM processor analyzes this stream from the DDR memory to detect pedestrians. Afterwards, the frames that are processed by the ARM processor are forwarded to an HDMI output to visualize detected pedestrians. An overview of the system is given by Fig. 8. An AXI performance monitor measures the performance of the system.

The total processing time of the HOG algorithm for a 1920x1080 frame is approximately 12.7 seconds which is equivalent to 0.08 fps. In comparison to related work, Mizuno et al. [10] achieves 30 fps which is a speedup of 375x for the same size of frames. This section presents 3 optimizations for the software implementation to improve the performance as much as possible to fulfill the real-time requirements.

A. Resize Down/Up

The HOG descriptor depends in human detection on the silhouette and structure of the human body. Resizing down the image to a smaller size does not affect significantly the silhouette of the human body until a certain limit of resizing. This implies that it does not downgrade the detection of the pedestrians, but it decreases the number of pixels that have to be processed.

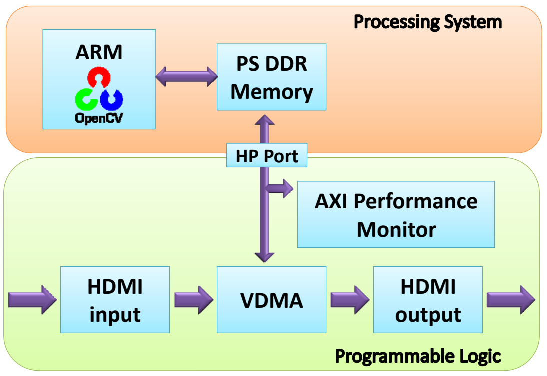

Fig. 8 Overview of the software system

Subsequently, it reduces the processing time of the HOG algorithm. However, if the human size in an image becomes much smaller than the image size that was used to train the classifier, the human might not be detected correctly. The decrease of processing time due to resizing comes at the cost of degradation of resolution after resizing the image down and up as shown in Fig. 9. A nearest-neighbor interpolation is used in this figure. Since the main target of the application is the detection of pedestrians in real-time, the resolution of the output image can be sacrificed for the sake of decreasing the computation time of the HOG algorithm. According to that, a frame is resized down before it is processed and resized up after it is processed using the OpenCV *resize( )* function with nearest neighbor interpolation.

B. *Region of Interest*

Another method to reduce the runtime of the HOG algorithm is the idea of not processing the whole frame but a specific Region of Interest (ROI). In this region it is expected that humans appear and they must be detected. For example, Fig. 10 shows a driving scene given by the car simulator [13]. There is a large area of the image where humans are not expected to appear and it is not valuable to detect humans in this area. Therefore, processing only a ROI (bordered with a frame shown in Fig. 10) would decrease the computation time significantly as the size of the image to be processed is decreased. The ROI specified was of size 700x350 pixels at the center of the image.

C. *Threading Building Blocks*

Threading Building Blocks (TBB) is a C++ library developed by Intel that allows writing parallel programs.

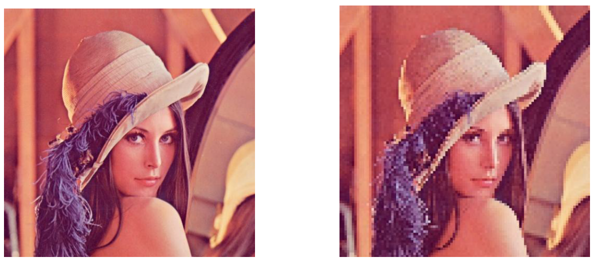

Fig. 9 Resolution decreasing: original image (l.) and image resized down and up (r.)

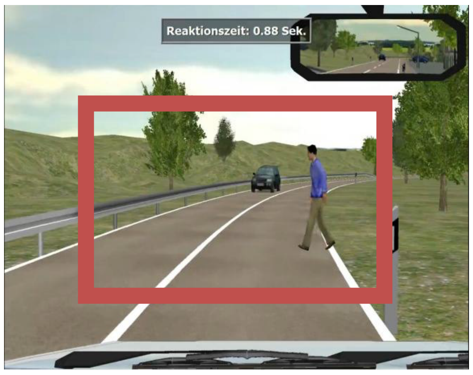

Fig. 10 Driving scene with ROI given by a car simulator [13]

It can take the advantages of multicore architectures. In this paper, the ZedBoard is used that has a dual-core ARM Cortex-A9 processor. OpenCV allows the use of TBB functionality with its native data types by providing the cv::parallel_for_ function. This function executes independent for-loops in parallel. Since the processing system (PS) of the ZedBoard is a dual-core Cortex-A9 processor, it is possible to use TBB along with OpenCV. In order to reduce the processing time theoretically to the half, the image is divided into two halves that are then processed in parallel. By dividing the image equally into two halves that are processed independently by two cores, a human that is located in the middle of the image is probably not detected. However, pedestrians are expected to arrive from the left or right side in the driving scene. Thus, pedestrians will be detected before they arrive in the middle of the image.

D. *Complete Sequence of the HOG-based Pedestrian Detection*

The optimization techniques mentioned before are combined to increase the system performance. First, the input frame is acquired. A ROI that shows the most critical part for pedestrians in this driving scene is specified manually at the beginning of the program. By resizing down this ROI, the number of pixels that have to be processed is reduced. Since the software approach runs on a dual-core processor, the frame is divided into two halves to be processed simultaneously by the cores. After the HOG computation has finished, rectangles are drawn around pedestrians detected in this scene. Before the stream is forwarded as an output HDMI stream, the frame is resized up to the original frame size of 1920x1080. Listing I presents the pseudo code for the optimization steps to reduce the HOG computation time.

LISTING I. PSEUDO CODE OF THE COMPLETE SEQUENCE OF THE HOG-BASED PEDESTRIAN DETECTION

```
1: Get input 1920x1080 image
2: Specify a 700x350 region of interest
3: Resize down by scaling each dimension of the image
4: Divide into two halves to be processed in parallel using TBB
5: HOG computation
6: Get the output image with rectangles drawn around humans
7:  Resize up to original 1920x1080 size
8: Copy to the address of the output buffer of the VDMA block
```

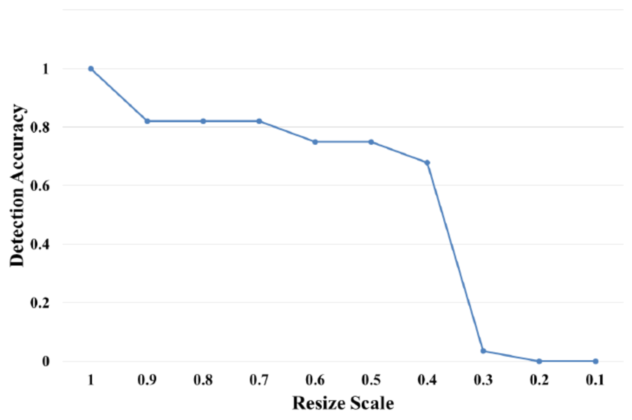

Fig. 11 Resize down scale vs. detection accuracy

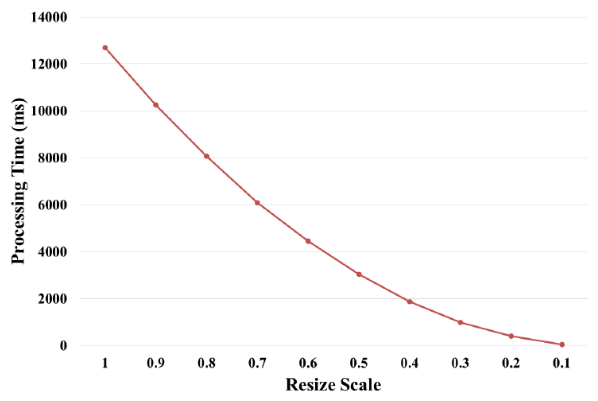

Fig. 12 Resize down scale vs. processing time

V. Evaluation

A. HOG Algorithm Implemented in Software

Resizing down reduces the resolution of the frames. By reducing the resolution the accuracy also decreases. Fig. 11 shows the relation between the resize down scale and the detection accuracy using a 1080p image that contains 28 pedestrian images from the INRIA pedestrian dataset [12]. In addition, the relation between the resize down scale and the frame processing time is shown in Fig. 12.

According to that, instead of processing the full resolution 1920x1080 frame, it is resized by scaling each dimensions of the frame to 0.5 using the OpenCV resize( ) function. Resizing down by more than 0.5 leads to large degradation in the detection accuracy. This dropdown of the detection accuracy can be explained with a missing classifier trained for this scale. The classifier is not able to classify humans in this size anymore. Another classifier trained for this size of detection could solve this problem. The processing time of a single frame is reduced from 12.7 seconds to 3 seconds. This is a significant speedup of 4.2x.

Combining resizing with ROI leads to a further reduction of the computation time from 3 seconds to 0.091s.

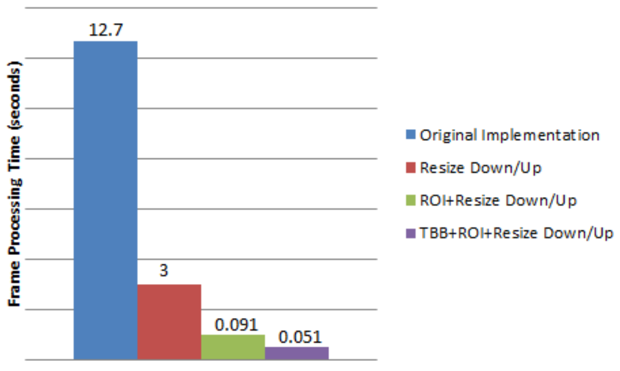

Fig. 13 Frame processing time of the original software implementation with and without optimizations

This means that it runs 30x faster than when using the resize method only and 120x faster than the original implementation without any optimizations. In addition, the system performance is improved by exploiting parallelism of the application using TBB. The frame processing time decreases from 0.091 seconds to 0.051 seconds. A summary of the different performance results is presented by Fig. 13.

B. Comparison between Hardware and Software

The hardware implementation is synthesized for the ZedBoard using Xilinx Vivado Design Suite 2014.4. Simulation using ISim 14.7 is used to verify the functionality of each step of the HOG algorithm. The results of the simulation are compared to a Matlab implementation. Each step of the hardware was implemented in the same way in Matlab. Both implementations are evaluated with frames from the car simulator [13]. In conclusion, the Matlab results showed the same output in comparison to the hardware implementation. This proves that the implemented hardware architecture executes each step of the HOG algorithm properly. Five hundred images (200 positive samples and 300 negative samples from [14]) were used to train the classifier for a detection window size of 136x280 pixels. Matlab supports an AdaBoost learning algorithm which has been used in this work. This classifier was evaluated using 1104 images. It gives an accuracy of 90.2% and a false positive rate of 4%. The results can be further improved by using a larger set of images to train the classifier.

The synthesis and implementation of the complete system gives an operating frequency of 82.2 MHz. A full post-synthesis and post-routing VHDL test-bench simulation is performed to measure the frame processing time. A complete HDTV 1080p frame needs 25.207 ms to be analyzed by the implemented HOG algorithm. Consequently, it achieves a final throughput of 40 fps. The original software implementation is improved by a factor of 250x in terms performance due to several optimizations. In comparison to this optimized software implementation, the hardware implementation still achieves a speedup of 2x without any optimizations, such as ROI or Resized Down/Up, as shown in Fig. 14.

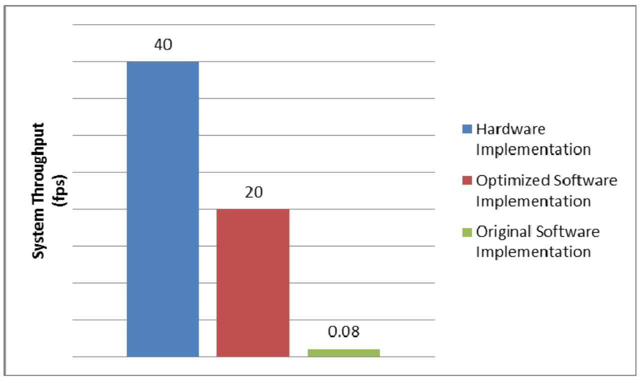

Fig. 14 Comparison between hardware and software implementations in terms of system throughput

TABLE II. RESOURCE UTILIZATION OF THE HOG ALGORITHM IMPLEMENTED ON XC7Z020-CLG484-1 AFTER PLACE&ROUTE

MODULE	**LUTS**	**SLICES**	**DSPS**
LUMINANCE CALCULATION	59	17	0
GRADIENT COMPUTATION	2985	797	0
GRADIENT MAGNITUDE AND DIRECTION	8432	2397	4
FORMATION OF CELL HISTOGRAM	6140	1676	0
BLOCK NORMALIZATION	1278	421	0
BINARIZATION	93	26	0
ADABOOST CLASSIFIER	2310	608	0
TOTAL	**21297 (40%)**	**5942 (45%)**	**4 (2%)**

The resource utilization after place&route of the implemented HOG algorithm is shown in TABLE II. The results are generated for the ZedBoard using Xilinx Vivado Design Suite 2014.4.

VI. CONCLUSION & OUTLOOK

This paper presents a real-time implementation in hardware and software for pedestrian detection using the HOG algorithm presented by Dalal and Triggs [1] on the ZedBoard. These systems process high resolution images with 1920x1080 pixels. The HOG algorithm implemented in an FPGA uses an AdaBoost classifier. It has an additional binarization step that allows in combination with a modified normalization step an efficient FPGA implementation. A verification of the system with Matlab shows a reliable detection rate of 90.2 % and a minor false positive rate of 4%. Besides the hardware approach, a software implementation using OpenCV has been implemented on the ARM processor of the ZedBoard and improved by a speedup of 250x. This is enabled by exploiting the data parallelism and a reasonable reduction of pixels that have to be processed. However, the FPGA implementation achieves 40 fps which is twice as much as the optimized software implementation.

Future work is to improve further the frame processing time of the software implementation by exploiting NEON cores from the ARM processor. Adding multi-scale detection to the hardware implementation is an improvement in order to be able to detect pedestrians of different sizes. This could be achieved by training several AdaBoost classifiers. These classifiers can be reconfigured partially to reduce the resource utilization.

REFERENCES

[1] National Center for Statistics and Analysis. (2015, February). Pedestrians: 2013 data. (Traffic Safety Facts. Report No. DOT HS 812 124). Washington, DC: National Highway Traffic Safety Administration.

[2] Dalal, N.; Triggs, B., "Histograms of oriented gradients for human detection," Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, vol.1, no., pp.886,893 vol. 1, 25-25 June 2005.

[3] Chiang, C.; Chen, Y.; Ke, K.; Yuan, K., "Real-time pedestrian detection technique for embedded driver assistance systems," in Consumer Electronics (ICCE), 2015 IEEE International Conference on , vol., no., pp.206-207, 9-12 Jan. 2015.

[4] Bauer, S.; Kohler, S.; Doll, K.; Brunsmann, U., "FPGA-GPU architecture for kernel SVM pedestrian detection," Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on , vol., no., pp.61,68, 13-18 June 2010.

[5] Lee, S.; Son, H.; Choi, J. C.; Min, K., "HOG feature extractor circuit for real-time human and vehicle detection," in TENCON 2012 - 2012 IEEE Region 10 Conference , vol., no., pp.1-5, 19-22 Nov. 2012.

[6] Kelly, C.; Siddiqui, F.M.; Bardak, B.; Woods, R., "Histogram of oriented gradients front end processing: An FPGA based processor approach," Signal Processing Systems (SiPS), 2014 IEEE Workshop on , vol., no., pp.1,6, 20-22 Oct. 2014.

[7] Negi, K.; Dohi, K.; Shibata, Y.; Oguri, K., "Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm," Field-Programmable Technology (FPT), 2011 International Conference on , vol., no., pp.1,8, 12-14 Dec. 2011.

[8] Shuai Xie; Yibin Li; Zhiping Jia; Lei Ju, "Binarization based implementation for real-time human detection," Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on , vol., no., pp.1,4, 2-4 Sept. 2013.

[9] Kadota, R.; Sugano, H.; Hiromoto, M.; Ochi, H.; Miyamoto, R.; Nakamura, Y., "Hardware Architecture for HOG Feature Extraction," Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP '09. Fifth International Conference on , vol., no., pp.1330,1333, 12-14 Sept. 2009.

[10] Mizuno, K.; Terachi, Y.; Takagi, K.; Izumi, S.; Kawaguchi, H.; Yoshimoto, M., "Architectural Study of HOG Feature Extraction Processor for Real-Time Object Detection," Signal Processing Systems (SiPS), 2012 IEEE Workshop on , vol., no., pp.197,202, 17-19 Oct. 2012.

[11] Ma X.; Najjar, W.A.; Roy-Chowdhury, A.K., "Evaluation and Acceleration of High-Throughput Fixed-Point Object Detection on FPGAs," in Circuits and Systems for Video Technology, IEEE Transactions on , vol.25, no.6, pp.1051-1062, June 2015.

[12] Y. Freund; R. E. Schapire; "A decision-theoretic generalization of on-line learning and an application to boosting," in Proceedings of the 2nd European Conference on Computational Learning Theory. London, UK: Springer Verlag, 1995, pp. 23-27.

[13] Foerst GmbH, "Programming Tool Reference", 2012. Available at: www.driving-simulators.eu

[14] INRIA Person Dataset. Available at: http://pascal.inrialpes.fr/data/human/

[15] G. Overett, L. Petersson, N. Brewer, L. Andersson and N. Pettersson, A New Pedestrian Dataset for Supervised Learning, In IEEE Intelligent Vehicles Symposium, 2008.