J. Parallel Distrib. Comput. 109 (2017) 50–62

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

HW/SW Co-Design of the HOG algorithm on a Xilinx Zynq SoC

Jens Rettkowski [a,*], Andrew Boutros [a], Diana Göhringer [b]

[a] Application-Specific Multi-Core Architectures (MCA) Group, Ruhr-University Bochum, Germany
[b] Chair for Adaptive Dynamic Systems, Technische Universitaet Dresden, Germany

HIGHLIGHTS

- This work presents three different implementations of the HOG algorithm.
- A software implementation achieves a speedup of 249× due to several optimizations.
- The HW/SW Co-Design achieves a speedup of 9× compared to a software implementation.
- The FPGA-based implementation achieves 40 fps.

ARTICLE INFO

Article history:
Received 16 July 2016
Received in revised form 3 May 2017
Accepted 8 May 2017
Available online 24 May 2017

Keywords:
HOG algorithm
FPGA
Xilinx Zynq
Image processing
Real-time
HW/SW Co-Design
Human detection
SDSoC

ABSTRACT

An accurate and fast human detection is a crucial task for a wide variety of applications such as automotive and person identification. The histogram of oriented gradients (HOG) algorithm is one of the most reliable and applied algorithms for this task. However the HOG algorithm is also a compute intensive task. This paper presents three different implementations using the Zynq SoC that consists of an ARM processor and an FPGA. The first uses OpenCV functions and runs on the ARM processor. A speedup of 249×is achieved due to several optimizations that are implemented in this OpenCV-based HOG approach. The second is a HW/SW Co-Design implemented on the ARM processor and the FPGA. The third is completely implemented on the FPGA and optimized for an FPGA implementation to achieve the highest performance for high resolution images (1920 × 1080). This implementation achieves 39.6 fps which is a speedup of 503.9× compared to the OpenCV-based approach and 2× compared to this implementation with optimizations. The HW/SW Co-Design achieves a speedup of approximately 9× compared to an original HOG implementation running on the ARM processor.

1. Introduction

Recently, the detection of humans has been thoroughly investigated in a wide range of applications from robotics to automotive. In a lot of robotic applications, robots have to recognize humans in order to interact with them [15]. Another example is the essential need for a fast and accurate detection of pedestrians in autonomous driving. The National Highway Traffic Safety Administration at the U.S. Department of Transportation [21] shows that 4735 pedestrians were killed and about 66,000 were injured in car accidents in the United States only in 2013. This means that on average a pedestrian was killed and fifteen others were injured in car accidents every two hours. These numbers are extremely frightening as well as demanding for an effective solution to reduce the number of fatalities. Having a vehicle that is capable of detecting pedestrians ahead with a reliable accuracy and speed became an inevitable need. It would surmount the human driver performance in analyzing the complex and continuously dynamic driving scenery in crowded cities and urban areas not only in the direction of vehicle motion but also a total 360° monitoring around the vehicle in all directions. It would also allow for a very fast automated response of the vehicle as it would not just alarm the driver and depend on the relatively slow human reflex action in avoiding an accident. Advanced Driver Assistance Systems (ADAS) try to solve this issue with growing success. However, the high number of accidents caused by pedestrians show that new and innovative approaches are still needed to reduce the number of accidents.

During the past decade, the famous Histogram of Oriented Gradients (HOG) algorithm first introduced in [7] has been the state-of-the-art in human detection. It introduced a sliding-window based feature set that is extracted from a given frame and then can be reliably classified using a pre-trained binary classifier into a positive or negative detection of a human with very high accuracy [7].

* Corresponding author.
E-mail addresses: jens.rettkowski@rub.de (J. Rettkowski), andrew.boutros@student.guc.edu.eg (A. Boutros), diana.goehringer@tu-dresden.de (D. Göhringer).

http://dx.doi.org/10.1016/j.jpdc.2017.05.005
0743-7315/© 2017 Published by Elsevier Inc.

This comes at a cost of very high computing power demand for this algorithm, which contradicts with the strong timing constraints. Thus, it can be concluded that there would always be a trade-off between frame processing time and detection accuracy. Therefore, this algorithm could make great use of different architectures such as embedded processors, Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) or a combination of more than one solution as in Hardware/Software Co-Design or heterogeneous hardware architectures for example.

This article presents three different designs of the HOG algorithm on a Xilinx Zynq SoC. Firstly, highly optimized parallel software implementation enabled by the OpenCV library is introduced using the embedded dual-core ARM processor on a Xilinx Zynq device. This is followed by a discussion that sheds light on both the advantages and disadvantages of such an implementation in terms of frame processing time and programmability. A HW/SW Co-Design of the original HOG algorithm is implemented using Xilinx SDSoC tool. SDSoC uses HLS to synthesize software-coded functions to the FPGA and generates the interfaces for data exchange between the ARM processor and the FPGA. A complete pipelined FPGA-based implementation of the HOG algorithm along with an AdaBoost classifier is implemented. Several optimizations are introduced to the original HOG algorithm in order to make it more suitable for a hardware implementation. This implementation is tested on the programmable fabric of the same Xilinx Zynq device showing an improvement both in computation time and resource utilization when compared to previous FPGA implementations of the HOG algorithm and fulfilling real-time requirements for pedestrian detection in high resolution 1080P video frames. All different realizations are discussed and compared in terms of frame processing time and programmability. This article is an extension of [26]. In [26], the FPGA implementation and the OpenCV-based approach of the HOG algorithm are presented. However, a link between the software- and hardware-based approaches is missing. Thus, this article contributes with a HW/SW Co-Design of the HOG algorithm using SDSoC.

This work is organized as follows: In Section 2, related work of the HOG algorithm and its different applications are presented. Section 3 shows the software implementation using OpenCV functions on the ARM processor. Section 4 analyzes the HOG algorithm and presents a HW/SW Co-Design with SDSoC. The third implementation that uses only the FPGA is presented in Section 5. Section 6 discusses all implementations. Finally, a conclusion is given in Section 7.

2. Related work

The HOG algorithm was originally published by Dalal and Triggs [7] in 2005 as an effective method for human detection. To show the significance of the HOG algorithm, numerous applications for object detection are discussed in the following section:

Remote sensing applications: The use of the HOG algorithm in the detection of landmines and explosives in Ground-Penetrating Radar (GPR) acquired data is introduced in [31] and compared to other existing feature extraction techniques in GPR such as Edge Histogram Descriptor (EHD) showing a slightly better performance of the HOG algorithm compared to its counterpart. Also in space exploration, the HOG features along with boosting and SVM classifiers was proposed in [18] to detect dunes on the surface of Mars in remotely sensed images captured by the Mars Orbiter Camera with a detection accuracy of 95%.

Automotive applications: In [10] a non-text traffic signs detection system was introduced based on extracted HOG features and a cascade of SVM classifiers. The algorithm was implemented in parallel as a pipeline on a 3.3 GHz Intel Core i5 CPU and tested under different illumination and weather conditions. The frame rate achieved in this system was 20 fps with frame resolution 640 × 480 and detection accuracy 89.2% for white signs and 92.1% for color signs. The work of Kim et al. in [12] introduces a solution for two drawbacks in the HOG algorithm by putting into consideration both the Position of the gradients and the Intensity information (PIHOG) to detect on-road vehicles resulting in an improved performance both in terms of detection accuracy and computation time.

Biometric applications: A variation of the HOG descriptor, named Histogram of Oriented Lines (HOL), that is robust against changes in illumination and slight rotations, is proposed in [32] as an effective method for palm-print detection. It was tested on a 2.5 GHz Intel i5-2520 M CPU using MATLAB 2009 showing a recognition rate reaching 99.97% and 100% for two different palm-print databases. Also the combination of both HOG and Scale Invariant Feature Transform (SIFT) were used as a basis for a new descriptor known as Histogram of Invariant Gradients (HIG) described in [6] for fingerprint recognition with an average accuracy of 87.8%.

The previously mentioned examples show that either a variation of the HOG algorithm or a combination between HOG and other features is used to achieve high performance and accurate object detection tasks besides human detection. Also, the use of the HOG algorithm in pedestrian detection is not exclusive for visible spectrum intensity images. Applying the HOG algorithm on Infrared (IR) images to eliminate poor illumination problems has been investigated in the work of Zhang et al. in [17]. This work compares different combinations of two features classes: Edgelets and HOG features along with two classifiers: AdaBoost and SVM concluding that the detection accuracy achieved on IR images were comparable to that on ordinary images. In addition to that, Wu et al. [29] proposed a new feature descriptor, Histogram of Depth Difference (HDD) that describe the depth variance in a region of a depth image and fused it with HOG descriptors to obtain a detection rate reaching to 99.12%.

Since Dalal and Triggs published the HOG algorithm in 2005, many researchers investigated in efficiently accelerating this computationally expensive algorithm on FPGAs and GPUs making use of their high parallelism capabilities. Acceleration of the HOG algorithm is not only done using FPGAs or GPUs, but also makes use of DSPs and heterogeneous systems consisting of different architectures as well. In [5], 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. Also, Chavan et al. [1] used a Texas Instruments' C674x DSP to implement the HOG algorithm with throughput 20 fps. When this implementation is compared to the software version of the OpenCV library, the DSP implementation achieved similar accuracy results and improvement of 130× in terms of computation speed. Both [5] and [1] process 640 × 480 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. [2]. The multi-core CPU managed the FPGA–GPU pipeline, while the FPGA was responsible for the feature detection and the GPU for the SVM classification. In the work by Blair et al. [4] different partitioning schemes of the algorithm among a CPU–GPU–FPGA system are compared showing how each of them affects one or more of the three parameters: power consumption, computation time and detection accuracy. One of the very first implementations of the HOG algorithm using GPUs was the *fastHOG* introduced in [24].

In [5], the HOG was implemented on a GeForce GTX 285 GPU and it was found that the detection accuracy of the proposed implementation is similar to the original detector by Dalal and Triggs with a 67× improvement (13 fps for 640 × 480 frames) when compared to the sequential CPU implementation. It was also estimated that using two GeForce GTX 295 GPUs in Quad SLI configuration would achieve a processing rate of 10 fps for 1920×1080 frames. In [30], a highly optimized GPU implementation of the HOG algorithm along with geometric ground plane constraints to enhance the algorithm runtime is introduced. The proposed algorithm is tested on frames of size 680×480 using GeForce GTX 280 GPU achieving same detection accuracy as that of Dalal and Triggs detector and a throughput improvement from 22 fps to 57 fps when applying the ground plane constraints. Bilgic et al. [3] used a GeForce GTX 295 GPU to implement the HOG algorithm along with a cascade-of-rejectors that cuts down the computation time for detection windows that are not possibly containing a human and focus only on windows that are harder to classify. The obtained results showed that the proposed system processes 8 fps for 1280×980 images. Another GPU implementation was described in [35] in which the processing rate of this GPU implementation using GeForce GT 240M GPU was compared to that of a Dual-Core Intel 2.2 GHz CPU. The processing of a single 640×480 frame took 31 ms (33 fps) which is 11× faster than the CPU implementation. All the previous GPU implementations were achieved using the NVIDIA CUDA framework. However, Sun et al. [28] introduced a GPU implementation using OpenCL and it was tested on three different heterogeneous platforms to compare the runtime on GPU and CPU. The first platform consisted of a 3.4 GHz Intel Core-i7 and an AMD Radeon HD6450 GPU, the second platform was a 2.53 GHz Intel Core-i3 CPU along with a GeForce 310M GPU and the third platform consisted of a server PC with an Intel Xeon X5690 and a NVIDIA Quadro 5000 GPU. The third platform was the fastest of the three achieving a throughput of 36 fps on the GPU compared to 8 fps on the CPU for 768×576 frames.

Negi et al. [22] introduced a hardware implementation of the HOG algorithm with an AdaBoost classifier on a Xilinx Virtex-5 LX-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. [33] 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 67.75 MHz was achieved. Both implementations [33] and [22] were for a frame size of 320×240 pixels.

In order to optimize the algorithm for an FPGA implementation, methods to simplify the computation were proposed by Kadota et al. [13] They implemented the feature extraction part only 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 640×480 pixels.

Mizuno et al. [20] 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 1920×1080 frames on an Altera Cyclone IV EP4CE115 FPGA. In [19] Komorkiewicz et al. implemented a pipelined single precision 32-bit floating point HOG feature extraction architecture along with SVM classifier to detect not only pedestrians but different types of objects in a single frame using Xilinx ML605 board with Virtex 6 XC6VLX240T device and Avnet DVI I/O FMC expansion card for video input. The result of this implementation was compared to the original OpenCV software implementation to give similar obtained results. The proposed system analyzed 640×480 images at a frequency 25 MHz and throughput of 60 fps. Yuan et al. [34] proposed a two stage pipeline architecture with SVM classifier embedded as a part of the normalization stage to reduce the hardware resources used. This system was implemented on Spartan-6 FPGA for image resolution of 800×600 operating at a frequency 100 MHz with a throughput 47 fps. In addition to that, a method to avoid multiple detections in neighboring windows was introduced.

In contrast to the presented related work, the FPGA implementation of this work 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 1600×1200 pixels 10.41 fps are achieved which results in approximately 20×10^6 pixels that are processed in 1 s. The FPGA implementation presented in this work is not scale invariant. However, it achieves 4× more pixels that are processed in 1 s. The comparison of FPGA with GPU implementations shows that results in terms of performance using FPGAs are more promising. Therefore, this work focuses on FPGA-based implementations.

Another approach is presented in [14] 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.

An implementation of the HOG algorithm on a 0.13 μm CMOS standard cell library consisting of 1,046,807 gates with a maximum frequency of 200 MHz is developed in [16]. However, this system achieves a lower frame processing time of 33.38 fps for a 640×480 image size in contrast to the work implemented on an FPGA presented in this work.

3. Histogram of oriented gradients

The HOG algorithm can be used to detect humans in images by extracting features that can be classified into humans or no-humans by a classifier such as Support-Vector-Machine (SVM) and AdaBoost. Since classification needs to be robust, the feature extraction describes the image in such a way that humans are detectable under different conditions. The HOG algorithm uses global features to describe a human instead of local features.

Local features are used to represent parts of the human body such as arms and legs. By combining these local features a human can be classified. In contrast to local features, global features represent directly a human. The original algorithm uses a sliding window that groups pixels and moves through the entire image with overlapping. This sliding window creates at every position a feature set that is classified into humans or no-humans by a classifier. The features are calculated with 5 steps in the original work of Dalal and Triggs.

The first step is optional and can be applied for RGB-images. It converts the RGB-image into a gray-scale image. Thus, each pixel is represented by a single byte. The second step computes the gradients for each pixel in x and y-direction. The gradients for a pixel in $X(Y)$ direction is computed by the difference of neighboring pixels in $X(Y)$ direction. Afterwards, the third step computes the magnitude and orientation of these gradients. The gradients are computed as unsigned values. The orientation ranges from 0° to 180°. The fourth step takes the magnitude and orientation values previously computed and sorts them in a histogram. The bins of these histograms are given by dividing the orientation into intervals. The corresponding magnitude is sorted into the respective bin. The magnitude is partly ordered depending on the orientation. For instance, gradients with an orientation of 15° contributes with $3/4$ of its magnitude to the 10° bin and $1/4$ of its magnitude to the 30° bin. Dalal and Triggs divided the pixels into 8×8 pixels that are defined as a cell. For each cell a histogram is calculated. The fifth step normalizes the resulting histograms from the previous step. This step increases the reliability in luminance variation. Instead of normalizing separately each histogram; the cells are combined to blocks. A single block is built by 2×2 cells. These blocks are overlapped by sliding through the image. Each block

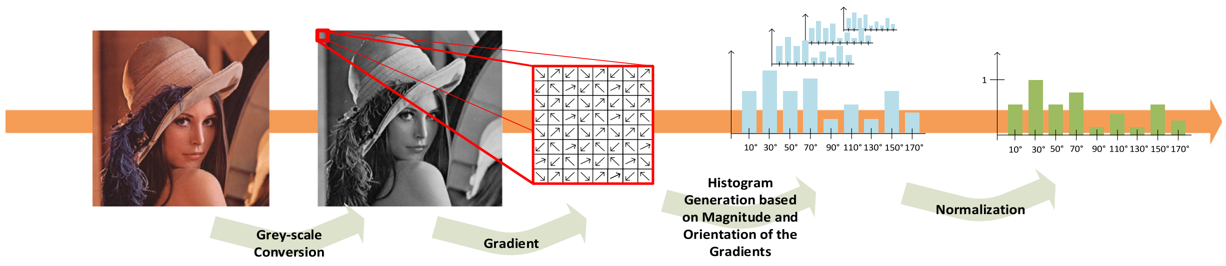

Fig. 1. Complete overview of the HOG algorithm.

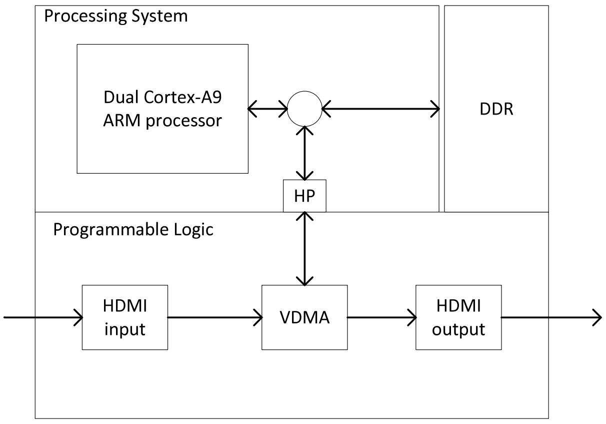

Fig. 2. Overview of the software system.

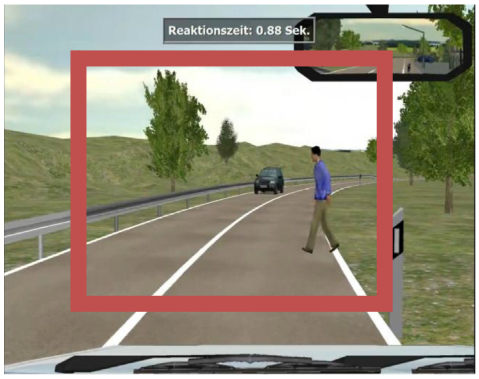

Fig. 3. Driving scene with ROI given by a car simulator [8].

is normalized using all values that are located in 2 × 2 cells. The sliding of the blocks causes that each cell appears multiple times in the feature set normalized with different neighboring cells. A complete overview of the entire algorithm is given by Fig. 1.

After the normalization step, the extracted features of the image can be forwarded to a classifier. The classifier determines if the feature set corresponds to a human or non-human. Dalal and Triggs used an SVM classifier. The SVM classifier distributes the space of the feature set into different classes. Other classifiers such as AdaBoost were also tested and verified as stated in Section 2.

4. SW-based implementation of the HOG algorithm

The first implementation [26] of the HOG algorithm is based on OpenCV 2.4.1. The ARM processor of the ZedBoard is equipped with a Linux operating system (OS). The OpenCV library provides an implementation of the HOG algorithm along with a pre-trained SVM classifier for human detection. The presented OpenCV- based implementation employs a dedicated SIMD engine of the ARM processor which speeds-up the execution. To process streams using the OpenCV-based implementation, an HDMI input stream is used. An HDMI output stream shows the processed frames. A video direct memory access (VDMA) core controls the data transfers from the HDMI streams to and from the memory of the ARM processor via the High Performance (HP) port of the Zynq SoC. The incoming HDMI stream contains a driving scene with pedestrians given by a car simulator [8] in order to verify the functionality in an automotive context. The car simulator is from the company FOERST. The software executing the HOG algorithm analyzes the frames located in the DDR. Afterwards, the result is visualized by sending the processed frames to an HDMI output. An overview of the system in the Zynq SoC is given by Fig. 2. This approach achieves a processing time of 12.7 s for a single high-resolution frame with a size of 1920 × 1080 pixels. This corresponds approximately to 0.08 fps. The OpenCV implementation is configured to multi-scale mode and uses a detection windows size of 64 × 128 pixels. The histogram consists of 9 bins. In comparison to Mizuno et al. [20], they achieved 30 fps which is 375× faster for the same size of frames. In order to improve the processing time of the OpenCV-based approach, three different optimizations are implemented: region of interest (ROI), resize Down/Up and threading building blocks. (TBB) [25].

4.1. *Region of interest*

In a lot of image processing algorithms, the region of interest can be applied to reduce the data that has to be processed. This approach assumes that only a specific region of the frame is relevant. This means that a human does not need to be detected in the entire frame. In case of automotive, a car only needs to detect humans in front of the car. Fig. 3 presents a driving scene from the car simulator [8] containing a frame that defines the ROI. It is not necessary to detect humans around the frame, since only pedestrians that cross the street can cause an accident. A pedestrian that walks in parallel to the car on the pavement does not need to be necessarily detected, since they will not cross the street. Pedestrians that are approaching from the side should be detected as soon as they are within the ROI. Thus, the ROI should have a greater width as the street in a realistic scene. An alternative to this approach would be to start with an initial and fast object detection algorithm that defines a ROI which can be analyzed afterwards by the HOG algorithm. However, it must be considered that the initial object detection algorithm creates an additional latency. To achieve more accuracy in reliability in autonomous driving, redundant systems are required. Accordingly, it would be not sufficient to capture only an ROI in front of the car. Multiple sensors would capture a 360° view which could have different priorities to be analyzed, since a pedestrian behind the car is not as important as a pedestrian in front of the car. Therefore, the 360° view could be split into several ROI with different priorities. Another approach can use such a system to verify the results of a radar system. A radar

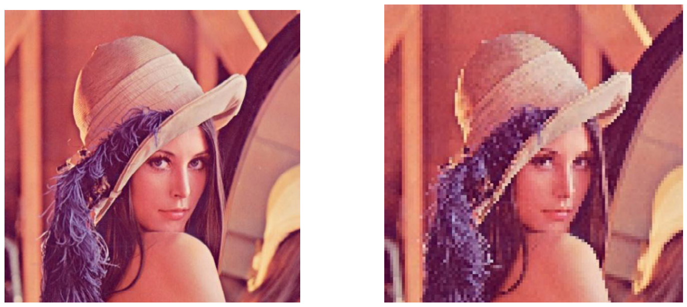

Fig. 4. Resolution decreasing: original image (l.) and image resized down and up (r.).

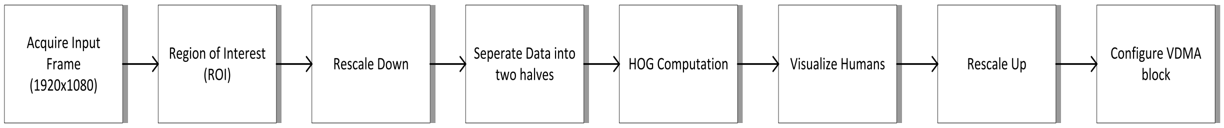

Fig. 5. Sequence of the HOG-based pedestrian detection using ROI, Rescaling and TBB.

system can propose a region that has the potential to contain a pedestrian. This region can be additionally analyzed and verified by the HOG algorithm. In this context, this work describes the analysis of one ROI. The ROI decreases significantly the computation time by reducing the size of the frame that needs to be processed. The ROI specified is of size 700 × 350 pixels in the center of the image.

4.2. Resize down/up

Generally, the silhouette and structure of the human body influences the features extracted by the HOG algorithm. Decreasing the size of the image does not affect essentially the silhouette of the human body. Consequently, resizing down the frame does not imply directly a worse human detection, although the OpenCV implementation used in this work is scale-invariant. The number of pixels that have to be processed decreases by resizing down the image. Fig. 4 shows an example of an image that is resized down and resized up using a nearest-neighbor interpolation. The resolution degrades after resizing the frame. However, the reduction of pixels decreases the processing time of the frames without losing a lot of accuracy until a certain limit. This certain limit is defined by the human size that is used to train the classifier. The main goal of this work is to detect fast and accurately humans. Therefore, degradation in terms of resolution is not necessary, when the accuracy does not decline and the performance increases. 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.

4.3. Threading building blocks

Intel develops a C++ library called Threading Building Blocks that provides the functions to write parallel programs. Programming using TBB can take advantage of the dual-core ARM processor by distributing the processing of frames to both cores. OpenCV provides TBB functionality with its native data types by using the cv::parallel_for_function. Assuming that the processing can be distributed ideally to the dual-core processor of the Zynq chip and the communication time is not taken into account, a maximum speed up of 2×can be achieved.

The frame is split from the middle into two halves that are processed in parallel. The HOG algorithm is applied separately to each half. Thus, we eliminated data dependencies between the two halves of the frame. Applying TBB, each half of the frame is processed on one core of the ARM processor in parallel. A human that is located in the middle between both halves is not detected. However, pedestrians arrive from the left or right side in the driving scene. They are not appearing directly in front of the car without crossing the street. A special case would be that a pedestrian walks or stands in the middle of a straight street. To solve this issue, the frames could be overlapped by the width of the detection window size. Another solution would be to turn the camera in the range of a few degrees. This way the camera scans the environment and no overlapping is required.

4.4. Complete sequence of the HOG-based pedestrian detection

To achieve the highest system performance, the optimizations mentioned previously are combined. After acquiring the input frame, a ROI that shows the most critical part for pedestrians in this driving scene is specified at the beginning of the program. The position of the ROI is manually located in the center of frame. To reduce further the number of pixels that have to be processed, the ROI is resized down. Using TBB, the resized ROI is divided into two halves and processed separately on both cores. If humans are detected, rectangles are drawn around the pedestrians to visualize the detection. After resizing up the frame to its original size, it is forwarded to the output HDMI port. Fig. 5 presents the optimization steps to reduce the HOG computation time.

4.5. Evaluation

As mentioned resizing down increases the system performance due to the reduction of pixels that have to be processed. Fig. 6 presents the detection accuracy in relation to the resize down scale for a 1080p image that contains 28 pedestrian images from the INRIA pedestrian dataset [9].

For certain scales the detection of these 28 pedestrian is checked. Fig. 7 shows the relation between the resize down scale and the frame processing time, which represents the execution time of the HOG computation without the remaining steps shown in Fig. 5. It can be seen that resizing down by more than 0.4 causes a significant degradation in the detection accuracy. This sharp decline can be explained by the classification. The corresponding classifier is not trained at this scale. Thus, the classifier is not able to classify humans in this size anymore. A reconfiguration of the classifier with a classifier trained for this scale can solve this problem of decreasing detection accuracy. According to the

Table 1
Comparison of frame processing time and processed pixels per seconds of the original implementation with and without optimizations.

	Original implementation	Resize down/up	ROI+resize down up	TBB+ROI+resize down/up
Processing time (s)	12.7	3	0.091	0.051
Processed pixels per second	163 275	146 257	569 692	260 970

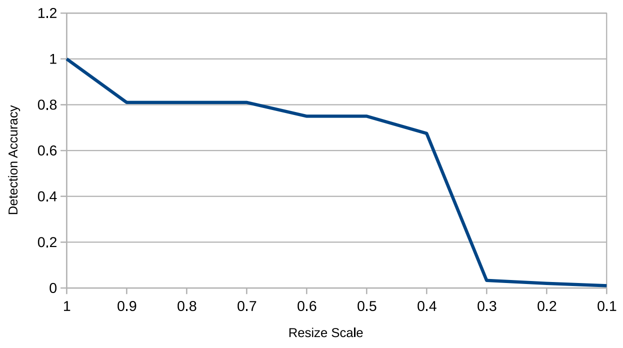

Fig. 6. Resize down scale vs. detection accuracy.

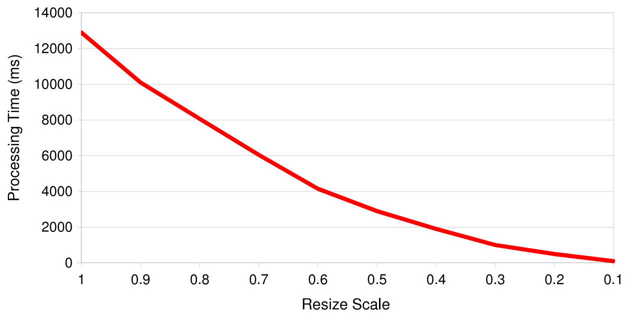

Fig. 7. Resize down scale vs. processing time on ARM processor for a high resolution image (1920 × 1080).

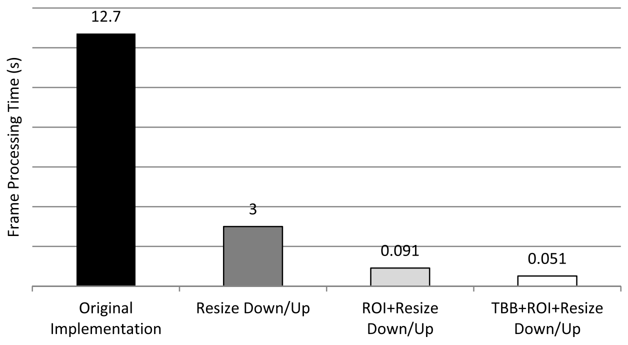

Fig. 8. Frame processing time of the original software implementation with and without optimizations.

detection accuracy, the frame is resized down with a scale of 0.46 using the *resize()* function from OpenCV. The scale of 0.46 is chosen to be not to close at 0.4. This resizing enables a significant speedup of 4.2×. Only 3 s are required to process a single frame instead of 12.7 s. The processing time can be further improved by combining resizing with ROI. The combination decreases the processing time from 3 to 0.091s. Accordingly, a speedup of approximately 30×and 120×compared to the original OpenCV implementation without any optimizations is achieved. The last optimization based on TBB decreases the 0.091–0.051 s. Fig. 8 shows the frame processing time with respect to the optimizations.

Table 1 compares the processing time and processed pixels per seconds of the original implementation with and without optimizations. The resizing down using a factor of 0.46 reduces the amount of pixels by 0.2116. Accordingly, a maximum speedup of 4.73×can be theoretically achieved, The presented speedup of 4.2×is less, since the speedup of 4.73 assumes ideal conditions. Applying the concept of ROI in combination with resizing achieves the processing of 569692 pixels in one second which is greater than the number of processed pixels per second from the original implementation. However, these results show a non-linear behavior in terms of number of processed pixels. An explanation for this behavior can be a non-linear dependency between the allocation of memory and the memory size. In order to prove this, we implemented a simple image processing program that loads images. The execution time of this program for different sizes showed a non-linear correlation. Such a non-linear correlation can explain that the number of processed pixels increases using the ROI.

5. HW/SW Co-Design of the HOG algorithm

The Zynq device consists of an ARM processor and an FPGA on a single chip. This heterogeneous device can take advantage of both processing units (ARM and FPGA) for an application. The previous section presents an OpenCV-based implementation using the ARM processor. This section shows an implementation without OpenCV and combining computation power of the ARM processor and the FPGA.

The HW/SW Co-Design can be very time consuming. Additionally to SW skills, the HW must be implemented in HDL and communication problems such as synchronization must be handled between SW and HW. Xilinx provides the High-Level-Synthesis tool SDSoC [27] to generate a HW/SW Co-Design for an application written in C/C++. Functions of the C/C++ application can be specified to be accelerated in the FPGA and optimized by pragmas. SDSoC uses high-level-synthesis to generate accelerators for the FPGA. The data exchange between the SW and HW will be automatically managed by SDSoC. Furthermore, it supports bare metal projects and a Linux operating system as well as FreeRTOS. In this work, the ZedBoard is equipped with a Linux operating system by SDSoC. The HW/SW Co-Design for the Zynq is developed using SDSoC 2015.4.

The main limitation of SDSoC 2015.4 is the amount of data that can be transferred. This version does not support a data transfer of 6415200 bytes that would be needed for a high resolution image. This issue will be solved in the next versions according to Xilinx. Since the OpenCV-based implementation is optimized by using a ROI and resizing down, the image has only a size of 322×161 pixels. Therefore, the image used in the SDSoC-based implementation has 350 × 175 which is approximated from the reduced size of the optimized OpenCV-based implementation.

5.1. Application profiling on ARM

The OpenCV-based implementation that runs on the ARM processor uses a shared library containing OpenCV functionalities. A subset of OpenCV is supported by SDSoC. However, this subset does not contain the HOG algorithm. Therefore, a C++ implementation of the original HOG algorithm is developed in SDSoC. This C++ implementation has all steps of the HOG algorithm and is compatible to SDSoC. Subsequently, all steps can be synthesized for the FPGA.

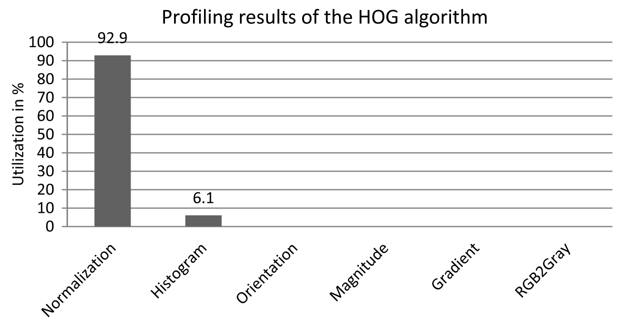

Fig. 9. Profiling results of HOG algorithms on ZedBoard using SDSoC 2015.4.

Pseudo code of the developed program is shown in Listing I. It starts with allocating memory using sds_alloc that is a function provided by Xilinx. This function allocates memory for the device drivers that manage the data exchange between FPGA and ARM processor. It allocates a physically contiguous array of bytes for DMA transfers. This array provides faster access for the DMA cores. Afterwards, all steps of the original HOG algorithm are executed sequentially. The allocated memory has to be released by the function sds_free( ) after the normalization step. All steps are implemented as described in Section 3.

Fig. 9 presents profiling results of the HOG algorithm. SDSoC 2015.4 was used to obtain these results. The normalization step, that is the most expensive function, consumes 92.9% of the execution time of the program.

Listing I. Pseudo code of the complete C++-based HOG Implementation

```
1: sds_alloc (RGB, Gray, Grad_x, Grad_y, Mag, Orient, Hist,
binHist)
2: RGB2Gray(RGB, Gray)
3: Gradient(Gray, Grad_x, Grad_y)
4: Magnitude(Grad_x, Grad_y, Mag)
5: Orientation(Grad_x, Grad_y,  Orient)
6: Histogram(Orient, Mag, Hist)
7: Normalization(Hist, normHist)
8: sds_free(RGB, Gray, Grad_x, Grad_y, Mag, Orient, Hist,
normHist)
```

The histogram step consumes 6.1%. This is due to grouping the pixels to cells and blocks that creates a lot of data by sliding through the frame. The remaining functions consume only a negligible overhead in comparison to the histogram and normalization step. Thus, SDSoC does not display the profiling results of these functions. The remaining functions except the gradient computation process every pixel without including the neighboring pixels. Accordingly, these functions are completed faster.

5.2. HW/SW Co-Design of HOG

Based on the profiling results, the functions *Histogram* and *Normalization* will be accelerated using SDSoC. These functions require neighboring values to compute. The *Histogram* function takes a window of 8×8 pixels (cell) and the *Normalization* function takes a window of 2×2 cells (block). In order to avoid storing the entire image inside the FPGA memory, row buffers are used. Storing the entire image inside the FPGA causes a huge amount of BRAM resources. Since the data is sent in a stream, row buffers can be used to minimize the memory utilization. SDSoC synthesizes functions executing Vivado HLS in the background. Row buffers can be implemented by the class *HLS::stream* in Vivado HLS. Shifting a variable of the type *HLS::stream* represents an incoming value of a stream. However, this class is not supported by SDSoC per default. Thus, stub functions for SDSoC are generated that SDSoC compiles

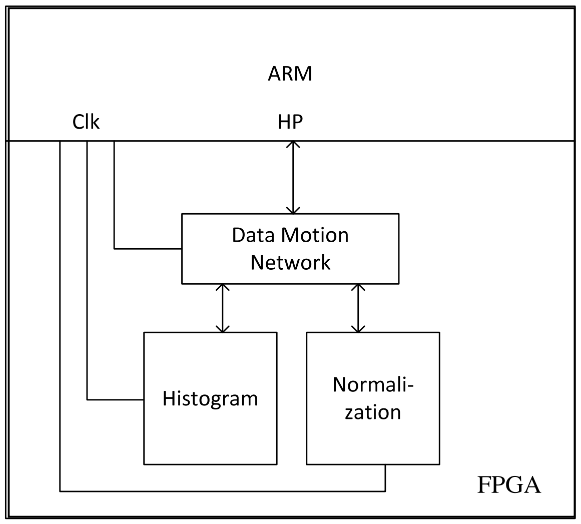

Fig. 10. Overview of design generated by SDSoC.

correctly. The construction of a stub function is similar to a function prototype. The stub function is an empty function that pretends SDSoC correct arguments. Instead of these stub functions, Vivado HLS synthesizes functions for the *histogram* and *normalization* step having arguments from type *HLS::stream*. This way row buffers can be implemented in SDSoC. Pragmas are used to specify the interface for each argument. In this work, the interface is configured to be an AXI DMA in scatter gather modus. The interface configuration is limited to certain configurations. However, the configuration used in this work provides the highest amount of data that can be transferred in SDSoC. The DMA core transfers data from the FPGA to the DDR memory or vice versa after an initial configuration by the ARM processor. Accordingly, the ARM processor can process data while the DMA core controls the data transfers. In addition, SDSoC generates a data motion network that handles the data exchange between the processing elements of a design. Since the data motion network synchronizes also the data exchange, the connected processing elements and the network can run at different frequencies. The data motion network and both functions are configured to run at 142.86 MHz. The ARM processor runs with 667 MHz. The predefined platform of SDSoC has four different configurations for the frequencies. The data motion network and each synthesized hardware block can be separately configured to 100 MHz, 142.86 MHz, 166.67 MHz or 200 MHz. The frequency of 142.86 MHz was the maximum achievable frequency for this design. The computation in software and hardware is realized with fixed-point arithmetic. Fig. 10 gives an overview of the design developed using SDSoC. The frame is preloaded in the DDR of the ARM processor. This allows using the entire FPGA only for the HOG algorithm. In order to optimize the high-level-synthesis of these blocks, loops inside the functions are specified to be unrolled and pipelined with pragmas.

5.3. Evaluation

The resource utilization of the FPGA is shown in Table 2. Vivado 2015.4 was used to generate these results. The C++ implementation of the original sequential HOG algorithm on the ARM processor achieves 0.047 fps. Fig. 11 shows the execution time of each step for a frame that has a size of 350×175 pixels. The C++ implementation accelerated with SDSoC by integrating the histogram and normalization step into the FPGA achieves 0.44 fps.

Table 2
Resource Utilization of the HW/SW Co-Design implemented on XC7Z020-CLG484-1 after Place&Route.

Module	Slices	LUTs	FFs	DSPs	RAMB36
Data motion network	2418	8708	21820	0	28
Histogram	1807	6687	9718	10	8
Normalization	7152	26463	35151	3	8
Total	11377	41858 (79%)	66689(63%)	13 (5%)	44(31%)

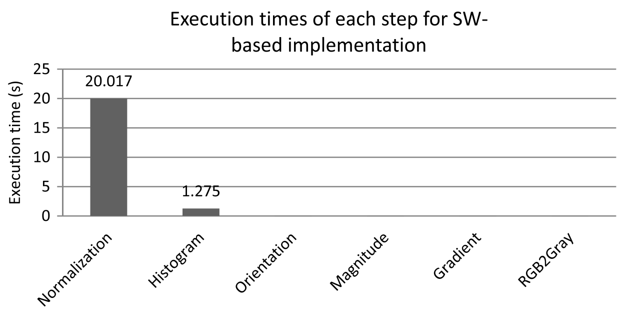

Fig. 11. Execution time of each step for the HOG algorithm on Cortex-A9 ARM processor running with 667 MHz.

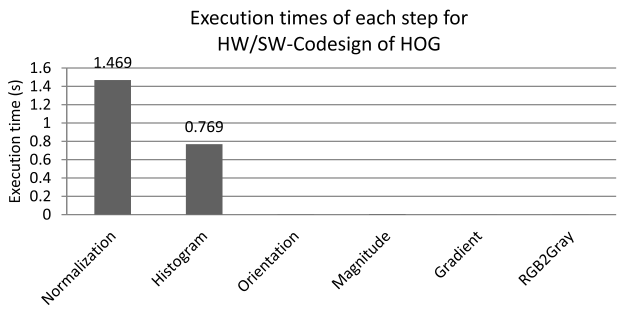

Fig. 12. Execution time of each step for the HOG algorithm on Cortex-A9 ARM processor running at 667 MHz and FPGA running at 140 MHz.

Fig. 12 presents the execution time of each step. Comparing both implementations, the HW/SW Co-Design achieved a speedup of almost 9×. The hardware implementation of the histogram function is 1.66×faster than the corresponding software implementation. It is worth to mention, that especially the synthesis of the normalization function achieved a higher performance than the corresponding software implementation. It has a speedup of 13.62×. These results show the benefit of high-level-synthesis on the Zynq device. However, the OpenCV-based implementation is still approximately 43.9×faster. As mentioned previously, all detailed information about the OpenCV-based approach are not available, but it is assumed that internally the implementation of the OpenCV HoG implementation is optimized. The SDSoC-based approach executes the original implementation of the HOG algorithm. Furthermore, the OpenCV-based approach can benefit from the SIMD engine of the ARM processor. The SIMD engine is very useful especially for image processing due to the parallel processing of pixels. The SDSoC version used in this work does not support the SIMD engine, which is a drawback for vector processing applications.

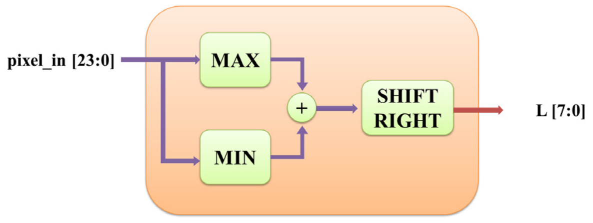

Fig. 14. Block diagram of the luminance calculation module.

6. HW-based HOG algorithm on an FPGA

The HW/SW Co-Design approach has data exchange between ARM processor and FPGA. This data transfer has a timing overhead that slows down the processing time. Thus, an approach using only an FPGA avoids the data exchange between FPGA and ARM processor and accordingly leads to higher system performance. An overview of the FPGA-based HOG algorithm [26] is given by Fig. 13. In order to implement the HOG algorithm efficiently in the FPGA, steps of the original implementation are adapted. Like the original work, the FPGA implementation begins with the luminance calculation converting an RGB-image to grayscale. Afterwards, the gradients are computed. The HOG descriptor is calculated by dividing the frame into cells that consists of 8 × 8 pixels as already explained. The next step calculates the magnitude and directly, the corresponding bin to form the histogram. This is more efficient for FPGAs instead of computing the orientation and afterwards, sort the magnitude based on the orientation. The following step forms the histogram for each cell. The normalization of these histograms improves the invariance to changes in illumination. In contrast to the original implementation a binary step is added to the algorithm. The optimizations applied to the FPGA-based implementation are not used in the HW/SW Co-Design. This step reduces the FPGA resources. An AdaBoost classifier distinguishes between human and non-human features extracted by the HOG algorithm.

6.1. Luminance calculation block

Fig. 14 shows the luminance calculation block. It is a module that calculates approximately the luminance value L by the equation:

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

The input signal *pixel_in* consists of three 8-bit color components R, G and B for each pixel resulting in a bit width of 24. The maximum output luminance value L is 255, because every pixel has a value range from 0 to 255. According to that, the output signal has 8 bits. The three different color values of *pixel_in* are compared to

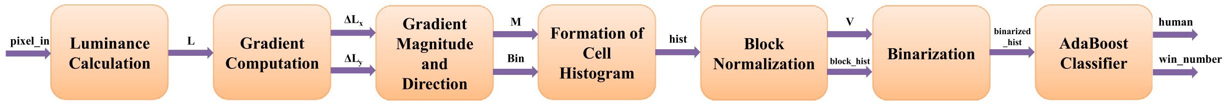

Fig. 13. Overview of the blocks for the HOG algorithm implemented in hardware.

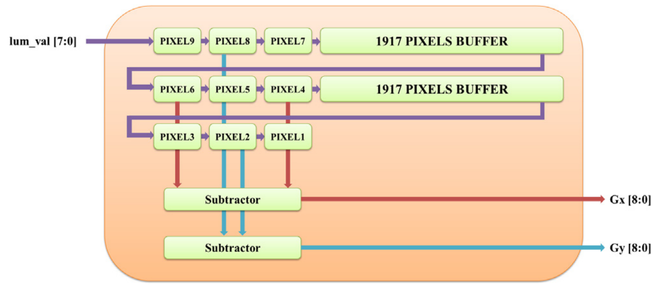

Fig. 15. Gradient computation module consisting of row buffers and subtractors.

determine the maximum and the minimum value. The average of these values builds the output value. This conversion is used, since it can be efficiently implemented in an FPGA. Finding the maximum or minimum of three values can be implemented using comparisons. The division by 2 is realized by shifting the sum of the maximum and minimum value. The outgoing grayscale values are forwarded to compute the gradients.

6.2. Gradient computation

The gradients at pixel position (x, y) in the horizontal $\Delta L_x(x, y)$ and vertical directions $\Delta L_y(x, y)$ are calculated by the 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 pixel (x, y) is the difference of the pixel values $L(x + 1, y)$ and $L(x - 1, y)$ for the horizontal direction. The vertical direction is built by the difference of $L(x, y + 1)$ and $L(x, y - 1)$.

The luminance values $L(x, y)$ are arriving line-by-line in the gradient computation block. Thus, row buffers are used to store two image rows of 1980 pixels and three pixels of the third row. Fig. 15 shows the gradient computation block with its row buffers. Since the output gradient signals $\Delta L_x(x, y)$ and $\Delta L_y(x, y)$ have a range from −255 to 255, they have a bit width of 9 bits.

6.3. Gradient magnitude and direction for bin assignment

Based on the gradients, the corresponding magnitude and orientation is needed in the original HOG implementation. It can be calculated using the following equations:

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

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

The magnitude computation is implemented by a 2-dimensional Look-Up Table containing approximated values for the magnitude depending on the inputs $\Delta L_x(x, y)$ and $\Delta L_y(x, y)$. The Look-Up Table has the output $M(x, y)$. Since the incoming values are squared, negative input values can be neglected. The highest value of the gradients is 255. Accordingly, the maximum magnitude is $\sqrt{255^2 + 255^2} = 360$. Thus, it can be represented in 9 bits. Therefore, the total memory required for this LUT is $256 \times 256 \times 9$ bits =72 KB which can be placed into a Block RAM (BRAM) from the FPGA or into slices optimized for memory. The assignment of the magnitude to histogram bins in the next stage is based on the orientation that ranges from $-\pi/2$ until $+\pi/2$. This range is divided into 8 bins for the magnitude. To implement this equation efficiently in terms of resources, the histogram generation is optimized for FPGAs based on [22] from the original HOG algorithm. For instance, a gradient direction θ that satisfies the following inequality, will be assigned to bin 7:

$$56.25^\circ < \theta < 78.75^\circ. \tag{6}$$

Eqs. (5) and (6) result to:

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

The inverse function of arctan in equation (7) applied to (7) with tan(56.25%) ≈1.496 and tan(78.75°) ≈ 5.027 gives the following equations:

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

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

In order to execute only integer multiplication which is more efficient for FPGAs, the inequality is multiplied with 1024 $\Delta L_x(x, y)$:

$$1533\Delta L_x(x, y) < 1024\Delta L_y(x, y) < 5148\Delta L_x(x, y). \tag{10}$$

Subsequently, only integer multiplication and simple comparisons instead of divisions are implemented. Such inequalities are built for each bin and implemented using conditional expressions. Consequently, these expressions give the corresponding bin to the gradients. Fig. 16 presents an overview of this step. The inputs are ΔL_x and ΔL_y that are generated in the previous step. Besides the magnitude output, the corresponding bin is also forwarded to the next block. The bin signal is represented by 3 bits.

6.4. Formation of cell histogram

The next step forms the histogram based on the bin with its corresponding magnitude out of a cell. A cell is built by a window of 8×8 pixels that slides through the frame.

Fig. 17 presents the block diagram that is used to build the histograms. Partial histograms are calculated for each row of a cell by the partial histogram generator. The row buffers allow to sum up the histograms of a cell.

The maximum value of a bin is $8 \times 8 \times 360 = 23.040$, when all 8×8 pixels have a maximum magnitude of 360 and are assigned to the same bin. Thus, a bin can be represented with 15 bits. As the histogram is constructed by 8 bins, the resulting histogram output *hist* has 8 x 15 bits = 120 bits.

6.5. Normalization

To be more robust against luminance changes, the histograms are normalized with respect to neighboring cells that form a block with 2×2 cells. Eq. (11) shows the normalization step:

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

The result v is the normalized histogram. The input v_k is the vector of 4 histograms from a block. $\|v_k\|$ is the summation of all elements of v_k and ε is a constant to avoid a zero enumerator. The division and square root consumes a lot of resources, when it is implemented in an FPGA. Negi et al. [22] reduces the resource utilization by simplifying the normalization step into different shift operations for sub-intervals. This work uses only shift operations without sub-intervals that also highly reduce the resource utilization.

Eq. (11) is simplified by removing the constant ε, since it is very small in comparison to the summation of all histogram elements

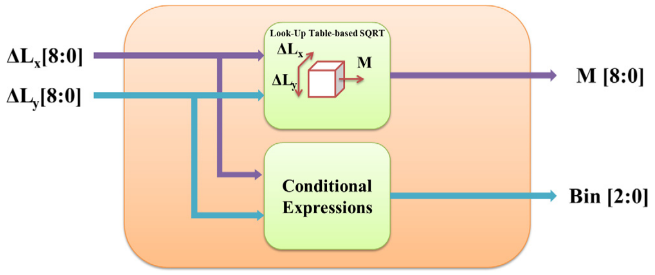

Fig. 16. Gradient magnitude and bin assignment module block diagram.

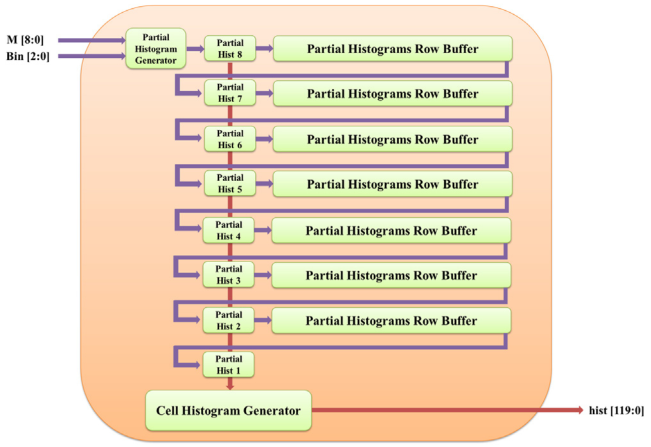
M [8:0]
Bin [2:0]
Partial Histogram Generator
Partial Hist 8
Partial Histograms Row Buffer
Partial Hist 7
Partial Histograms Row Buffer
Partial Hist 6
Partial Histograms Row Buffer
Partial Hist 5
Partial Histograms Row Buffer
Partial Hist 4
Partial Histograms Row Buffer
Partial Hist 3
Partial Histograms Row Buffer
Partial Hist 2
Partial Histograms Row Buffer
Partial Hist 1
Cell Histogram Generator
hist [119:0]

Fig. 17. Histogram generation module block diagram.

in a block $\|v_k\|$. Thus, Eq. (11) is formed to:

$$v = \frac{v_k}{(v_k)}. \tag{12}$$

The normalized histogram is calculated by a division in Eq. (12). The next step called binarization is additionally in comparison to the original HOG algorithm that enables an efficient implementation of Eq. (12). However, it allows to implement it efficiently for FPGAs. The binarization uses a threshold to binarize the histograms as shown in Eq. (13). If the value of each element of v is greater than this threshold, it is considered as logic '1', else it is considered as logic '0'. Fig. 18 illustrates the binarization step.

$$v = \begin{cases} 0, & \frac{v_k}{(v_k)} < threshold \\ 1, & \frac{v_k}{(v_k)} \geq threshold. \end{cases} \tag{13}$$

Negi et al. [22] implemented a binarization threshold of 0.08. However, this can result to floating point arithmetics. In this work, the threshold is set to 8/128 which is similar to 8/100. This threshold allows to be implemented as a logic shift right four times of $\|v_k\|$ that can be compared afterwards with v_k to binarize the histogram. The binarization step reduces every 14-bit value of v_k to a 1-bit binary value which reduces the size of the block histogram to 4 cells $\times$8 bits = 32 bits. Therefore, this step is one of the most important optimizations for an efficient FPGA implementation.

6.6. Classifier

The classifier is implemented to determine if a feature set represents a human or no human. Freund and Schapire [9] introduced an AdaBoost classifier in 1995. It creates a strong and accurate classifier by combining several weak and inaccurate classifiers. For instance, an AdaBoost classifier of fruits divides them into two groups: apples and other fruit. Weak classifiers could classify based on the following rules: circular objects are apple, apples are red or green or yellow and apples have a stem at the top. Using these classifiers separately results in inaccurate classifications. However, combining them improves the classification. Negi et al. [22] showed that an AdaBoost classifier gives a high accuracy for FPGA implementations with several optimizations. Furthermore, AdaBoost classifiers are most suitable when dealing with dense features sets which is the case in the HOG algorithm. It can be implemented using block descriptor buffers to form the detection window, a set of comparators and adders in an FPGA. In order to train the AdaBoost and verify the functionality of the FPGA-based approach, a complete MATLAB implementation of this approach was developed. The human detection is evaluated again with frames from the car simulator. 136 $\times$ 280 pixels are used for the detection window for human. The size is chosen because of the typical size for humans of the driving simulator. The detection window is composed of 16 $\times$ 34 blocks. Therefore, the descriptor of each detection window is a vector of 16 $\times$ 34 $\times$ 32 = 17,408 dimensions.

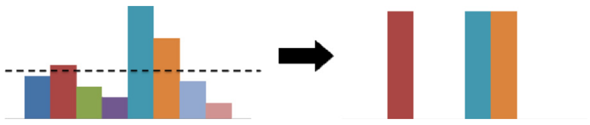

Fig. 18. Histogram binarization.

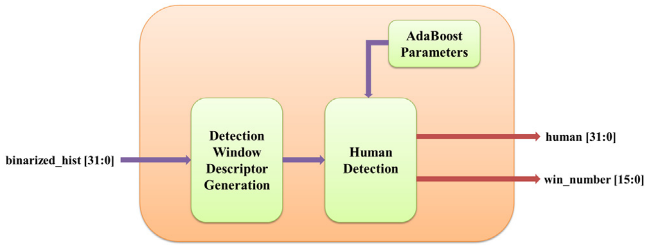

Fig. 19. AdaBoost classifier block diagram.

The training set for the AdaBoost classifier is based on two hundred positive samples of humans and three hundred negative samples of non-humans. The set is obtained from INRIA pedestrian database [11] and NICTA pedestrian database [23]. The classifier is trained with 500 iterations. As a result, 41 weak classifiers are computed. Each of the weak classifiers compares a specific dimension of the binarized descriptor of a complete detection window with its computed values. Afterwards, it sums a specific positive or negative weight depending on the binarized value and the classifier. The AdaBoost classifier is implemented in the FPGA. It is constructed in a pipeline to increase the performance of the system. The classifier is implemented that it compares each element of the vector with certain values given by the training. The detection window is classified as a human, when the overall sum is positive. In case of a negative number, the detection window contains no human. Fig. 19 shows a block diagram of the classifier. The output signal *human* indicates the result of the classifier. In order to locate a human in the frame, the 16-bit output signal *win_number* specifies the corresponding detection window. The *Detection Window Descriptor Generation* block receives the binarized histogram. Row buffers are used to generate the detection window of 16 $\times$ 34 blocks.

6.7. Verification with MATLAB

In order to make sure that the designed hardware architecture was functional correct, a VHDL behavioral simulation was performed using 8 rows of randomly generated numbers ranging from 0 to 255, each of which contains $1920 \times 3 = 5760$ values. Only 8 rows of number were evaluated to improve the simulation time of the VHDL design. These numbers represent the first 8 rows of a 1080P RGB image. A similar parallel implementation of the modified HOG algorithm was implemented completely in MATLAB. Then the VHDL test-bench results of each of the HOG algorithm stages were compared to their counterparts of the MATLAB implementation to make sure that the implemented hardware architecture has correct functionality. The MATLAB implementation was much easier and shorter in terms of development time due to the advantage and simplicity of dealing with multi-dimensional matrices.

Notice that the MATLAB implementation assumes that the borders of the frame have zero gradients in the *x*-direction and *y*-direction as it deals with the frame as an array instead of a stream. The VHDL implementation is a pipeline that deals with the image as a continuous stream of pixel values, thus having different values at the borders. The first steps of the algorithm create the same results in the MATLAB implementation. The histogram bins do not show identical results in the VHDL simulation and MATLAB implementation. However, it is found to be a result of a small difference between the MATLAB model and the VHDL model due to the different values at the borders of the frames. The hardware implementation is a pipeline that considers the image as a continuous stream of pixel values. This problem could be adapted in the MATLAB implementation, but it is not necessary since the error can be count back.

When calculating the histogram bin values for the first cell, the values of the last row in the VHDL model must be also taken into consideration. When these values are added to those from the MATLAB implementation, identical results are obtained. For the block normalization step, another 8 rows of randomly generated numbers were added to the inputs in order to form a 2×2 block which is 16×16 pixels. The VHDL simulation and MATLAB implementation results for this step are identical by considering the differences in gradient values mentioned previously. The binarization step is a shift and comparison operation. Therefore, it is proved to function correctly if the previous steps of histogram generation and block normalization are proved. As result, the implemented hardware architecture for the HOG algorithm is proved to be functioning correctly by performing a simulation for the VHDL code and comparing its results to that of the MATLAB implementation.

6.8. Evaluation

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 simulation enables a detailed observation of the processing time. 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 [8]. 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 [11]) were used to train the classifier for a detection window size of 136×280 pixels. The detection window size represents the size of the image that can contain a detectable human. Humans that cannot be displayed in such a detection window cannot be detected by the classifier. 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 39.6 fps.

The resource utilization after place&route of the implemented HOG algorithm is shown in Table 3. The results are generated for the ZedBoard using Xilinx Vivado Design Suite 2014.4.

7. Discussion

The HW/SW Co-Design achieves a speedup of ca. $9\times$in comparison to the original HOG implementation of SDSoC without synthesis of functions for the FPGA. This is a significant speedup with a practicable overhead in terms of programming using SDSoC. The main drawback of SDSoC is currently the limited size of data that can be transferred. A high resolution frame of size 1980×1020 requires 6415200 bytes. It is not possible to allocate the memory correctly for larger data transfers. However, future releases will solve this issue. Another limitation of SDSoC 2015.4 is that multiple functions, which are implemented in hardware and are executed sequentially, require a data transfer from the DDR memory to the FPGA and from the FPGA to the DDR memory after the processing is finished. This creates a high overhead in terms of performance, since for every core, that accelerates a function, data must be accessed in the DDR memory. The overhead increases especially for high resolution images.

In comparison to the OpenCV and FPGA implementation, the HW/SW Co-Design is the slowest solution. The exact HOG implementation of the OpenCV library is not known. However, it is probably optimized for the ARM processor by the OpenCV developers. Such optimizations can also increase the performance of the SDSoC-based approach. Furthermore, the NEON core that is a SIMD engine and attached to the ARM processor is enabled by compiling the OpenCV-based program for the ARM processor, which leads also to a speedup. Image processing algorithms can benefit a lot from such SIMD engines due to the parallel execution of the algorithm. SDSoC 2015.4 does not support the NEON core. Thus, it only uses the ARM processor and the FPGA core. Another reason for the slower processing time is the data transfer that has to be executed in between the FPGA cores. The processing time would be improved by streaming the data directly from the histogram step to the normalization step without copying the data to the DDR memory. SDSoC copies the data to the DDR memory and back to the FPGA, since it simplifies the automatic scheduling of data transfers by SDSoC. In future releases, it will be possible to directly send the data between cores according to Xilinx.

The FPGA approach is optimized to be efficiently implemented in an FPGA. It is also possible to integrate each FPGA core to the SDSoC approach. Within SDSoC, a function call can be mapped to the cores to transfer data. This way all optimizations can be also applied to the HW/SW Co-Design. However, this requires the knowledge about HDL to develop these cores. The High-Level-Synthesis simplifies the development of FPGA cores, but also loses flexibility of the design. The communication for data transfers is limited to different solution. Application-specific approaches cannot be used without major modification. As mentioned previously, SDSoC has a limited size of the data transfer length. The resource utilization of the HW/SW Co-Design is much higher than the FPGA implementation. This is due to the optimizations that

Table 3
Resource Utilization of the HOG Algorithm implemented on XC7Z020-CLG484-1 after Place&Route.

Module	LUTs	Slices	FFs	DSPs	RAMB36
Luminance calculation	59	17	7	0	0
Gradient computation	2985	797	90	0	0
Gradient magnitude and direction	8432	2397	13	4	0
Formation of cell histogram	6140	1676	1116	0	0
Block normalization	1278	421	500	0	0
Binarization	93	26	46	0	0
AdaBoost classifier	2310	608	203	0	0
Total	21297 (40%)	5942 (45%)	1975 (19%)	4 (2%)	0(0%)

Table 4
Frame Processing Time for the three presented implementations.

	Optimizations	Processing time (s)	Frame size
SW-based implementation (OpenCV)	–	12.7	1980 × 1020
	Resize down/up	3	910 × 469
	Resize down/up +ROI	0.091	322 × 161
	Resize down/up +ROI +TBB	0.051	322 × 161
HW/SW Co-Design (SDSoC)	–	21.3	350 × 175
	FPGA accelerators	2.27	350 × 175
HW-based approach (VHDL)	Binarization, optimizations for FPGA implementations	0.0252	1980 × 1020

are applied for the FPGA approach and the limited flexibility of the SDSoC design. Furthermore, a lot of resources are consumed by the communication network that is implemented by SDSoC. SDSoC uses a platform that is suitable for a general design that is not optimized for a specific implementation. As a result, the communication network contains also cores that are not necessary for this specific implementation. This creates an overhead in terms of resource utilization. The FPGA approach shows only the resources of the implemented cores that are directly connected without a network. Furthermore, an FPGA implementation has only physical constraints to create different clock domains. SDSoC can only create clock domains on function level. The frequency of the FPGA-based approach is 82.2 MHz which is lower than the frequency of the SDSoC-based approach. The SDSoC approach uses more dedicated hardware such as DSP and BRAM blocks. In general these blocks provide higher performance than using LUTs, since the dedicated blocks are optimized for their functionalities.

In addition, the FPGA implementation does not have the data exchange between ARM and FPGA. Consequently, no communication overhead between the individual cores is created. Therefore, a complete FPGA implementation that has an input stream connected directly to the FPGA promises higher performance. SDSoC uses data transfers between the DDR memory and the programmable logic between function calls which slows down the execution time. One way to solve this issue is defining only one function call for the whole processing within the FPGA. The FPGA implementation was evaluated with simulation, since it provides a detailed analysis of the timing. The OpenCV-based approach and the HW/SW Co-Design were evaluated on the ZedBoard. The best programmability is given by the OpenCV implementation taking advantage of the already existing OpenCV functions. SDSoC requires the implementation of the steps in C/C++. However, the HW/SW Co-Design using SDSoC is a good tradeoff between programmability and performance. Table 4 summarizes the obtained results from each implementation. Several optimizations are applied for the different approaches. The ROI and resizing down/up optimizations from the OpenCV-based approach reduce the amount of pixels that are processed. This concept can be also applied to the HW/SW Co-Design and the HW-based approach. However, the HW/SW Co-Design processes an image containing 350 × 175 pixels due to the limited amount of data that can be transferred. Further reduction of pixels using a resizing down and ROI approach will decrease the detection accuracy as shown in Section 5. The HW-based approach requires additional cores to split the HDMI input stream to the ROI and resizing down the frame. The HW-based approach would utilize fewer resources, since the buffers can be implemented for fewer pixels, and the processing time would be decreased. This is not only because of fewer pixels that need to be processed. A smaller design will probably also run at higher frequency. In this context, the acquisition of images must be fast enough to take advantage of these optimizations. The distribution of the algorithm on both ARM cores, can be applied to the HW/SW Co-Design. The accelerators cannot be shared in SDSoC by two processes. Therefore, distributing the HOG computation on both cores require a duplication of the accelerators connected to another HP port.

8. Conclusion

This work presents three implementations of the HOG algorithm on the Xilinx Zynq SoC introduced by Dalal and Triggs. The first implementation uses only the ARM processor of the Zynq SoC to process frames with a resolution of 1920 × 1080 pixels. Several optimizations are applied to improve the system performance of this implementation. The second implementation is a HW/SW Co-Design using the ARM processor in combination with the FPGA. The design was developed using SDSoC. The third implementation is an FPGA-based and pipelined approach of the HOG algorithm. The functions accelerated in the HW/SW Co-Design achieved a speedup of up to 13.62x in comparison to the original HOG algorithm. The complete HOG algorithm was accelerated by a factor of ca. 9×in the HW/SW Co-Design. By exploiting data parallelism and a reasonable reduction of pixels that has to be processed, the OpenCV implementation achieves a speedup of 249×compared to the OpenCV implementation without optimizations. However, the FPGA implementation achieves 39.6 fps which is twice as much as the optimized OpenCV implementation and 90.9×faster than the HW/SW Co-Design.

References

[1] Chavan Abhi, Senthil Kumar Yogamani, Real-time dsp implementation of pedestrian detection algorithm using HOG features, in: ITS Telecommunications, ITST, 2012, 12th International Conference on. IEEE, 2012.

[2] S. Bauer, S. Kohler, K. Doll, U. Brunsmann, FPGA-GPU architecture for kernel svm pedestrian detection, in: Computer Vision and Pattern Recognition Workshops, CVPRW, 2010, IEEE Computer Society Conference on, 2010 pp. 61,68, 13-18 June 2010.

[3] Bilgic Berkin, Berthold K.P. Horn, Ichiro Masaki, Fast human detection with cascaded ensembles on the GPU, in: Intelligent Vehicles Symposium (IV), 2010 IEEE, IEEE, 2010.

[4] Blair Charlotte, Neil M. Robertson, Danny Hume, Characterizing a heterogeneous system for person detection in video using histograms of oriented gradients: power versus speed versus accuracy, IEEE J. Emerg. Select. Topics Circuits Syst. 3 (2) (2013) 236–247.
[5] Chiang C., Y. Chen, K. Ke, K. Yuan, Real-time pedestrian detection technique for embedded driver assistance systems, in: Consumer Electronics, ICCE, 2015 IEEE International Conference on, 2015, pp. 206–207 , 9-12 January 2015.
[6] Gottschlich Carsten, et al., Fingerprint liveness detection based on histograms of invariant gradients, Biometrics, IJCB, in: 2014 IEEE International Joint Conference on, IEEE, 2014.
[7] N. Dalal, B. Triggs, Histograms of oriented gradients for human detection, in: Computer Vision and Pattern Recognition, 2005, CVPR 2005, IEEE Computer Society Conference on, vol. 1, 2005, pp. 886,893.
[8] Foerst GmbH, Programming Tool Reference, 2012 Available at: www.driving-simulators.eu.
[9] Y. Freund, R.E. Schapire, A decision-theoreticgeneralization of on-line learning and an application toboosting, in: Proceedings of the 2nd European Conference on Computational Learning Theory, Springer Verlag, London, UK, 1995, pp. 23–27.
[10] Greenhalgh Jack, Majid Mirmehdi, Real-time detection and recognition of road traffic signs, IEEE Trans. Intell. Transport. Syst. 13 (4) (2012) 1498–1506.
[11] INRIA Person Dataset. Available at: http://pascal.inrialpes.fr/data/human/.
[12] Kim Jisu, Jeonghyun Baek, Euntai Kim, A novel on-road vehicle detection method using HOG, IEEE Trans. Intell. Transport. Syst. 16 (6) (2015) 3414–3429.
[13] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, Y. Nakamura, Hardware architecture for HOG feature extraction, in: Intelligent Information Hiding and Multimedia Signal Processing, 2009 IIH-MSP '09, Fifth International Conference on, 2009 pp.1330,1333, 12-14 September 2009.
[14] C. Kelly, F.M. Siddiqui, B. Bardak, R. Woods, Histogram of oriented gradients front end processing: an fpga based processor approach, in: Signal Processing Systems (SiPS), 2014 IEEE Workshop on, 2014 pp. 1,6, 20-22 October 2014.
[15] G. Keramidas, C. Antonopoulos, N.S. Voros, Schwiegelshohn F., Wehner P., Rettkowski J., Goehringer D., Huebner M., Konstantopoulos S., Giannakopoulos T., Karkaletsis V., V. Mariatos, Computation and communication challenges to deploy robots in assisted living environments, in: Proc. of the Design, Automation and Test in Europe Conference, DATE, Dresden, Germany, 2016.
[16] S. Lee, H. Son, J.C. Choi, K. Min, Hog feature extractor circuit for real-time human and vehicle detection, in: TENCON 2012 - 2012 IEEE Region 10 Conference, 2012, pp. 1–5 19-22 November 2012.
[17] Zhang Li, Bo Wu, Ram Nevatia, Pedestrian detection in infrared images based on local shape features, in: Computer Vision and Pattern Recognition, 2007 CVPR'07, IEEE Conference on, IEEE, 2007,.
[18] Bandeira Lourenço, et al., Automated detection of martian dune fields, Geosci. Remote Sens. Lett. 8 (4) (2011) 626–630.
[19] Komorkiewicz Mateusz, Maciej Kluczewski, Marek Gorgon, Floating point HOG implementation for real-time multiple object.
[20] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, M. Yoshimoto, Architectural study of hog feature extraction processor for real-time object detection, in: Signal Processing Systems, SiPS, 2012 IEEE Workshop on, 2012 pp. 197,202, 17-19 October 2012.
[21] National Center for Statistics and Analysis, Pedestrians: 2013 data, (Traffic Safety Facts. Report No. DOT HS 812 124) Washington, DC: National Highway Traffic Safety Administration. 2015, February.
[22] K. Negi, K. Dohi, Y. Shibata, K. Oguri, Deep pipelined one-chip fpga implementation of a real-time image-based human detection algorithm, in: Field-Programmable Technology, FPT, 2011 International Conference on, 2011 pp. 1,8, 12-14 December 2011.
[23] G. Overett, L. Petersson, N. Brewer, L. Andersson, N. Pettersson, A new pedestrian dataset for supervised learning, in: IEEE Intelligent Vehicles Symposium, 2008.
[24] V. Prisacariu, I. Reid, fastHOG - a real-time GPU implementation of HOG, in: Department of Engineering Science, Oxford University, Tech. Rep. 2310/09, 2009.
[25] J. Reinders, Intel Threading Building Blocks, O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2007.
[26] J. Rettkowski, A. Boutros, D. Göhringer, Real-time pedestrian detection on a Xilinx Zynq fpga using the hog algorithm, in: Proc. of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), December, Cancun, Mexico, 2015.
[27] SDSoC Environment – User Guide UG1027, v20152, July 20, 2015. Available at:https://www.xilinx.com.
[28] Sun Rong, Xuzhi Wang, Xuannan Ye, Real-time pedestrian detectionl using opencl, in: Audio, Language and Image Processing, ICALIP, 2014 International Conference on, IEEE, 2014.
[29] Wu Shengyin, Shiqi Yu, Wensheng Chen, An attempt to pedestrian detection in depth images, in: Intelligent Visual Surveillance (IVS), 2011 Third Chinese Conference on. IEEE, 2011.
[30] P. Sudowe, B. Leibe, Efficient use of geometric constraints for sliding-window object detection in video, in: Int. Conf. on Computer Vision Systems, ICVS'11, 2011.
[31] Peter A. Torrione, et al., Histograms of oriented gradients for landmine detection in ground-penetrating radar data, IEEE Trans. Geosci. Remote Sens. 52 (3) (2014) 1539–1550.
[32] Jia Wei, et al., Histogram of oriented lines for palmprint recognition, IEEE Trans. Syst. Man Cybern. 44 (3) (2014) 385–395.
[33] Shuai Xie, Yibin Li, Zhiping Jia, Lei Ju, Binarization based implementation for real-time human detection, in: Field Programmable Logic and Applications, FPL, 2013 23rd International Conference on, 2013 pp. 1,4, 2-4 September 2013.
[34] Yuan Xu, et al., A two-stage hog feature extraction processor embedded with svm for pedestrian detection, in: Image Processing, ICIP, 2015 IEEE International Conference on, IEEE, 2015.
[35] C. Yan-ping, L. Shao-zi, L. Xian-ming, Fast hog feature computation based on cuda, in: Computer Science and Automation Engineering, CSAE, IEEE Int. Conf. on, vol. 4, 2011, pp. 748–751.

Jens Rettkowski received the B.Sc. degree in microtechnology from Westphalian University of applied Science in 2011 and the M.Sc. degree in Electrical Engineering and Information Technology from the Ruhr-University Bochum (RUB) Germany in 2014. He is a Ph.D. candidate at the MCA (application-specific Multi-Core Architectures) research group from the Ruhr-University Bochum (RUB), Germany. His research interests are self-adaptive networks-on-chip, Multiprocessor Systems-on-Chip (MPSoCs) and Reconfigurable Computing.

Andrew Boutros received his B.Sc. Engineering degree in Electronics and Electrical Engineering from the German University in Cairo, Egypt in July 2016. During the period from March to July 2015, he worked on his bachelor thesis at the application-specific Multi-Core Architectures (MCA) research group at Ruhr-University Bochum, Germany. He was appointed as a Research Assistant Intern at the German University in Cairo and Cairo University from July to August 2016. From September 2016, he is pursuing a M.Sc. degree at University of Toronto, Canada. His current research interests are FPGA architectures, embedded computer vision and hardware security.

Diana Göhringer is an assistant professor and head of the MCA (application-specific Multi-Core Architectures) research group at the Ruhr-University Bochum (RUB), Germany. She received her Ph.D. and her master degree in Electrical Engineering and Information Technology from the Karlsruhe Institute of Technology (KIT), Germany in 2011 and 2006, respectively. She is author and co-author of over 60 publications in international journals, conferences and workshops. Additionally, she serves as technical program committee member in several international conferences and workshops. She is reviewer and guest editor of several international journals. Her research interests include Reconfigurable Computing, Multiprocessor Systems-on-Chip (MPSoCs), Networks-on-Chip, Hardware–Software-Codesign and Parallel Programming Models.