HW/SW Co-design and Prototyping Approach for Embedded Smart Camera: ADAS Case Study

B. Senouci[1], H. Rouis[1], Q. Cabanes[1], A.C. Ramdan[2], D.S. Han[3]
[1]*Graduate Engineering School, ECE-Paris, INSEEC-U Research Center, Paris, France*
[2]*University of Versailles Saint-Quentin en Yvelines, LISV Laboratory*
[3]*Kyungpook National University, South Korea*
senouci@ece.fr

***Abstract*— In 1968, Volkswagen integrated an electronic circuit as a new control fuel injection system, called the "Little Black Box", it is considered as the first embedded system in the automotive industry. Currently, automobile constructors integrate several embedded systems into any of their new model vehicles. Behind these automobile's electronics systems, a sophisticated Hardware/Software (HW/SW) architecture, which is based on heterogeneous components, and multiple CPUs is built. At present, they are more oriented toward vision-based systems using tiny embedded smart camera. This vision-based system in real time aspects represents one of the most challenging issues, especially in the domain of automobile's applications. On the design side, one of the optimal solutions adopted by embedded systems designer for system performance, is to associate CPUs and hardware accelerators in the same design, in order to reduce the computational burden on the CPU and to speed-up the data processing. In this paper, we present a hardware platform-based design approach for fast embedded smart Advanced Driver Assistant System (ADAS) design and prototyping, as an alternative for the pure time-consuming simulation technique. Based on a Multi-CPU/FPGA platform, we introduced a new methodology/flow to design the different HW and SW parts of the ADAS system. Then, we shared our experience in designing and prototyping a HW/SW vision based on smart embedded system as an ADAS that helps to increase the safety of car's drivers. We presented a real HW/SW prototype of the vision ADAS based on a Zynq FPGA. The system detects the fatigue/drowsiness state of the driver by monitoring the eyes closure and generates a real time alert. A new HW Skin Segmentation step to locate the eyes/face is proposed. Our new approach migrates the skin segmentation step from processing system (SW) to programmable logic (HW) taking the advantage of High-Level Synthesis (HLS) tool flow to accelerate the implementation, and the prototyping of the Vision based ADAS on a hardware platform.**

***Index Terms*—ADAS; Embedded Architecture; FPGA based design; Hardware Accelerators; High Level Synthesis; HW/SW Co-design; Machine learning; Real Time OS; Smart Cars.**

I. INTRODUCTION

To replace the duel carburetors, Fastback and Square back system that control the fuel injection, in 1968, Volkswagen 1600 integrated an electronic circuit (more than 200 transistors, resistors, diodes and capacitors) as a new control fuel injection system. This system, called the "Little Black Box" [1] is the first embedded system for automotive industry. Presently, every year, automobile constructors integrate new embedded systems into their vehicles. On one hand, the massive usage and availability of these embedded devices on the marketplace bring products to a price consumer can pay for; on the other hand, scaling down of semiconductor technology below 14 nm will surely reach many of these devices, as it improves the diversity and availability of their application in automobile industry. These tiny devices integrated in automobiles collect and exchange information to control, optimize, and monitor many of the functions that just a few years ago were purely mechanical.

During the last decade, this technological advances in electronics enabled the exponential growth of smart objects, which embed more and more intelligence. Their processing and communication abilities provide new solutions to the problems of automobile applications. Smart camera is a typical example of these systems. Basically, it is a video camera coupled to a computer vision system in an embedded package. The smart-camera refers to cameras that are able to acquire and process images in real-time. It captures high-level descriptions of the scene and analyses it. These devices could support a wide variety of applications, including human detection, tracking, motion analysis, and facial identification.

On the other hand, EyeQ2 [2] system is one example of a single chip dedicated to automotive security applications using vision system, that consists of two 64- bit floating-point RISC 34KMIPS processors for scheduling and controlling the concurrent tasks, five vision computing engines and three vector microcode processors.

These Real-time video processing applications require huge computational power. Even though multiplying the processing units in the same design was adopted as the main response for the increasing circuit power computation demand, an alternative solution is being approved to improve this demand and to allow for more sophisticated embedded systems with larger computational capabilities.

Indeed, FPGAs, as parallel architectures, have the promised specialized hardware performance with high computational speed, lower clock frequencies and power consumption. They can implement the logics required by different types of applications by building customized hardware for each function. The main goal of this paper is to build a prototype of vision based ADAS (Advanced Driver Assistant System) as a smart camera capable to detect a fatigue state of the driver and generate alert. This system has to go through a hardware/software partitioning to make it fast and power efficient using multiprocessing and hardware accelerators. A prototype of the system is developed on a Multi-CPU/FPGA Zynq platform.

The rest of the paper is organized as follows: the second section is dedicated to the state of the art for driver drowsiness detection and for embedded systems design techniques. Section 3 details our design flow and methodology. In section 4, an insight of software part of the ADAS system (software application) is presented. In section 5, we detail the HW/SW co-design of the system. Section 6 presents the implementation and experimentation results, while the section 7 concludes the paper.

II. RELATED WORKS AND APPROACHES

A. *ADAS Systems*

Improving safety in roads is one of the biggest concerns of automobile constructors [3] [4] [5]. To address this issue, Advanced Driver Assistant Systems (ADAS) have been introduced to provide the car's drivers with an automatic warning to quickly evaluate a potentially dangerous situation. Many ADAS systems are available on the marketplace and have already been integrated in new cars. Parking-aid with its ultrasonic sensors or embedded cameras in the new cars generation illustrates the existing examples of these systems.

Developing ADAS systems for smart and autonomous cars has been a goal for research departments since many years ago. In the 1980s, the Autonomous Land Vehicle project, funded by the Defense Advanced Research Projects Agency (DARPA), demonstrated an autonomous road-following vehicle with a speed up to 30 km/h, and it used laser radar, computer vision, and autonomous robotic control [6]. In 2005, the winner team of the DARPA Grand Challenge made its car run autonomously for 212 Km in 6 hours and 53 minutes [7]. In recent years, autonomous driving has been attracting more interest due to its great improvements in technologies. Similar to Google, Tesla also tested all its electric car with autopilot capabilities with some certain safety restrictions [8]. In Europe, the new ongoing project VEDECOM-AUTOPILOT also deals with the smart and autonomous vehicles [9] [10].

Over the years, vehicle has been the focus of many safety improvements, including the seatbelts and airbags. Although these passive systems have clearly resulted in reduced damages and victims, the safety benefits of these passive systems have reached their maximum. As such, research community are focusing on active systems to reduce further crashes on the roads, taking advantage of the amazing progress in embedded electronic technologies. Active systems, such as adaptive cruise control, assisted braking, and lane keeping found on many of modern cars, should be complemented with smart embedded systems, such as vision-based technologies and others, to achieve even greater safety improvements.

B. *Drowsiness/Fatigue detection using Smart cameras*

A study presented in 2014 by the AAA Foundation (American Automobile Association) [11] prove that drowsy drivers are involved in 21% of fatal crashes, and 16.5% from a previous study done in 2010. Therefore, drowsy driving is one of the rising problems that keeps causing deaths and indicting damages each year [12]. And even if people are aware of this problem, they keep driving when they are drowsy. Further, a study done by the same foundation in the USA in 2015 proved that one of three drivers admitted driving while he/she is tired. This condition needs to be addressed as when you are tired you have a hard time keeping your eyes open and focus on the road. Thus, this has made driver drowsiness detection as a field that has received attraction from researchers. In fact, many methods have been developed for this purpose [13] [14]. Based on the state of the art and as to our knowledge, the research works done in the field of driver drowsiness detection may be regrouped into three categories [15].

The first is physiological field of study that based on brain related activities and biomedical signals. The most used methods in this category are the one based on Electroencephalogram (EEG) signals and Electrocardiography (ECG) [11] [16]. These methods have been reported to be highly authentic for detecting the state of driver drowsiness; however, they require a permanent electrode attached to the body of the driver, which often causes annoyance and discomfort to the driver. This is why they are not getting much attention in the market.

The second category includes methods based on driving behavior, such as the use of steering wheel or the car's pedals. They basically evaluate variations in several signals recorded by CAN bus, which are easy acquired and does not bother the driver. These features made the integration of these methods to the market easier. However, they are subjected to constraints related to the kind of vehicle or driver and the road conditions. They usually require long training periods.

The third category is based on visual assessment. It consists of using computer vision methods for monitoring driver's state from face images. These approaches are effective as sleepiness is reflected through the face and eyes appearance. These methods are considered as the most promising due to their accuracy and non-intrusiveness, and it is our concern in this paper. These techniques are based on studying facial features since fatigue and sleepiness can be detected through the face. There are several indicators used in this category to detect driver's drowsiness. Some based their decision on the frequency of yawning, others rely on the head pose, and eye gaze. In [17], Tayaba et al. developed a method for yawning detection. They used Viola and Jones method for detecting the face. The mouth region is then detected by an improved Fuzzy C-Means clustering technique that uses the spectral and the spatial information to segment the lips accurately. Trivedi et al. [18] used soft histograms of location-specific gradient orientation as the input to a support vector regressed for each degree-of-freedom in their method for head pose estimation.

The method we used in this work relies on an FPGA based HW/SW co-design using the percentage of eyes closure or the PERCLOS indicator to detect drowsiness state. It consists of detecting, for a short period of time, the frames with the driver's eyes being closed and those with the driver's eyes opened and then computing the proportion of time the eyes were closed. There are several steps involved before estimating the accurate PERCLOS value. The first step is the Skin Segmentation followed by the face and eye detection then the eye state classification. A Region of Interest (ROI) is selected on the detected face boundary to locate the eyes and then classify the eye state as open or closed. The method is further explained in the following points [19].

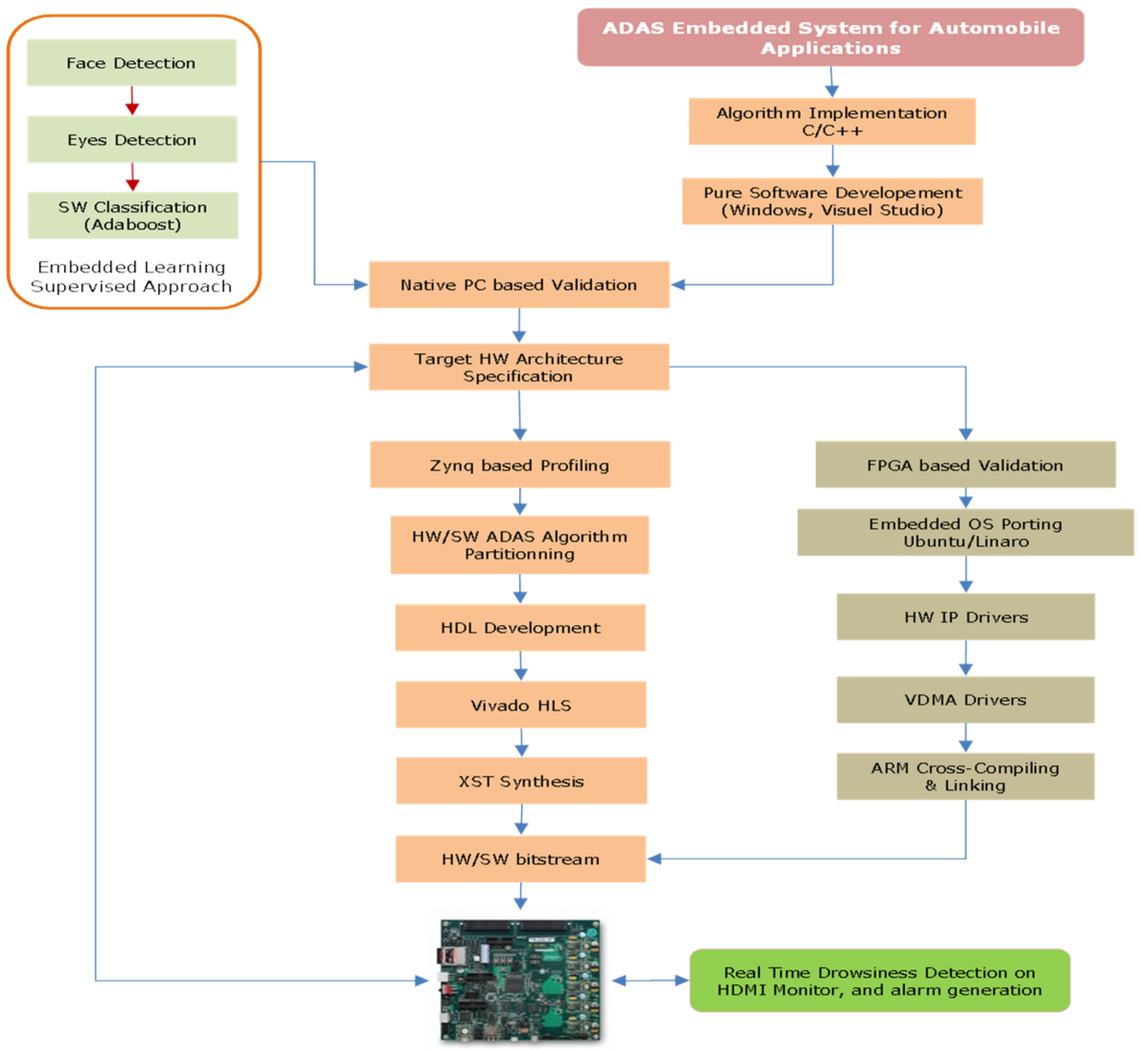
Face Detection
Eyes Detection
SW Classification (Adaboost)
Embedded Learning Supervised Approach
ADAS Embedded System for Automobile Applications
Algorithm Implementation C/C++
Pure Software Developement (Windows, Visuel Studio)
Native PC based Validation
Target HW Architecture Specification
Zynq based Profiling
HW/SW ADAS Algorithm Partitionning
HDL Development
Vivado HLS
XST Synthesis
HW/SW bitstream
FPGA based Validation
Embedded OS Porting Ubuntu/Linaro
HW IP Drivers
VDMA Drivers
ARM Cross-Compiling & Linking
Real Time Drowsiness Detection on HDMI Monitor, and alarm generation

Figure 1: Generic Design and prototyping Flow for Smart ADAS Systems

C. *Smart ADAS design/validation Techniques*

In the related works, the main alternatives design and validation techniques open to ADAS designers are simulation, and hardware platform-based design. Simulation-based design, with virtual prototyping of embedded architecture using Software-based simulation has been widely used at different level of abstraction. It is also widely used even when running on a high-end using simulation environment framework, with a complete system level simulator (correspondingly expensive). It runs six to ten orders of magnitude, slower than the hardware FPGA platforms, which makes the technique is an extremely time consuming and inefficient [20].

Table 1
Smart Embedded Design Techniques

	Speed	Complexity	Characteristics
Multi-CPU/FPGA Platform Based	Equal to Embedded Systems speed	+++	Very Close to the final Implementation
Simulation Based	Very slow (RTL level)	+++++	Bottleneck to Simulate Heterogeneous Components

In practice, general software validation can be performed on only small portions of the design. What designers need is an alternative that will allow them to get to the market quickly with low risk and at low cost. In this context, the use of Multi-CPU/FPGA platform becomes extremely attractive. They are designed to achieve higher integration levels in low-power low-cost.

Opportunities provided by these hardware platforms, combined with the evolution toward heterogeneous HW/SW, many processors architectures suggest new methods for designing and prototyping embedded systems. Another advantage is that hardware platforms implemented using FPGAs are "reconfigurable". Designers now have the freedom to select a set of hardware and software IPs to create a specialized smart embedded system. Table I presents a comparison between a platform based and simulation techniques for embedded design.

III. Smart Embedded Systems Design Flow

Our approach for smart embedded systems design and validation is based on a platform-based design approach using reconfigurable Multi-CPU/FPGA platform; we propose a design flow which enables seamless validation of multithreaded applications on the top of a hardware platform. This flow can be functional for all HW/SW architectures. Figure 1 presents the design flow. The investigation of the different techniques for smart embedded design and validation shows that we can find basically two configurations: simulation-based and platform-based.

However, in our new approach, we tried to propose a complementary methodology between the two techniques. Then, the first step in the design flow is to develop a pure software application based on the algorithm specification and to validate it on Visual Studio. A profiling step was done afterwards to decide which parts of the application should be turned into hardware accelerators. Once the HW/SW partitioning was done, we used the Vivado HLS tool to generate VHDL code from C++ as an HW-IP (Intellectual Property). The HW-IP's software driver is also generated at the same time. Then, the new HW-IP was integrated into the design using Vivado design suite, and SDK (Software Design Kit) was used to generate the booting files for the ZC702 FPGA board. We give more details about the different steps in the following sections.

IV. Application Development

A. *Drowsiness Detection Method*

In Figure 2, we show the software application task graph that we developed for fatigue detection [19]. For eyes detection, we used the Viola and Jones method for object detection. This method uses Haar Features, which are weak classifier and Adaboost algorithm to build a strong classifier.

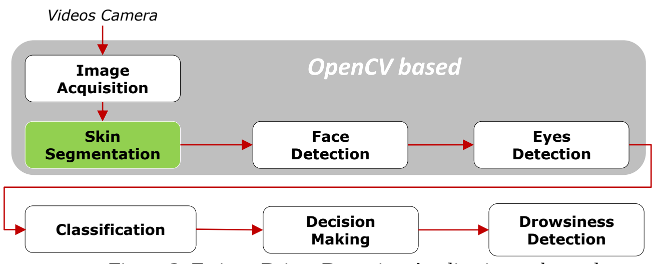

Figure 2: Fatigue Driver Detection Application task graph

However, the execution time of this method increases tremendously with the size of the searching zone. Therefore, we added a skin segmentation step with a contour detection for face extraction. This would allow us to use the eyes detection method only in the ROI (Region of Interest), which is the face.

B. *Skin Segmentation*

In the skin segmentation step, each pixel of the captured frame is determined whether it belongs to the skin or not. In order to accomplish that, we resorted to the **YCrCb** color model for image representation. The **YCrCb** color representation is based on three components: The luminance Y and two chrominance components Crand **Cb**. The transformation to convert from RGB to **YCrCb** color space is shown in equation 1:

This model has proven very efficient for skin segmentation according to many references such as the work done by Gururaj et al. [21]. This is mostly due to the fact that his color representation separates the luminance component from the others making the skin detection less dependent on lightning variations than the other color spaces. In **YCbCr** color space, the two chrominance components Cr, and Cb can be efficiently used to define explicitly the skin region. As shown in the Figure 3, we tested the **Cr** and **Cb** values for each pixel of the frame. It was verified the condition of the pixel is considered to be skin; Otherwise, it is outputted as background.

$$pixel = \begin{cases} skin, & 137 < Cr < 177 \text{ and} \\ & 77 < Cb < 127 \text{ and} \\ & 190 < Cb + 0.6Cr < 215 \\ non-skin, & otherwise \end{cases} \qquad (1)$$

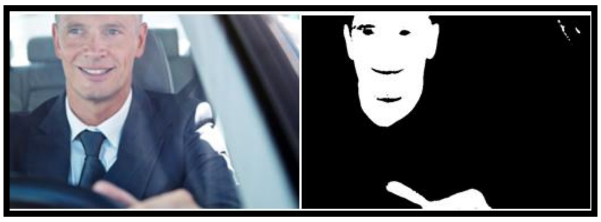

Figure 3: Skin Segmentation

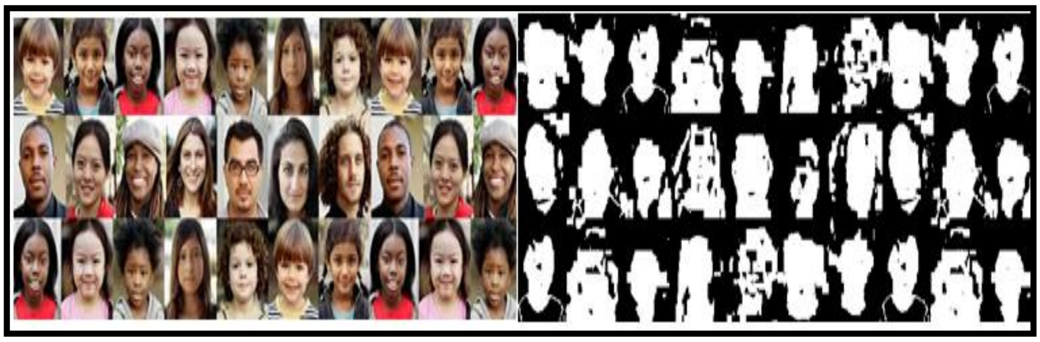

Figure 4: Skin Segmentation for different skin color

C. *Face Detection*

Once the non-skin area is eliminated from the image, the next step is to find the face contour. This is done by grouping all skin areas of the image into contours and then finding the biggest contour. In fact, we assume that the camera is going to be fixed in front of the driver and the biggest skin area is going to contain the face. However, we encountered a problem during this step. When the driver is wearing glasses, the face is going to be divided into two contours, and the biggest contour will not necessarily contain the eyes. Therefore, we resorted to add a morphological transformation for closing the contours. After finding the face contour, we extract the zone from the original image and start using the classifier to locate the driver's eyes in the region as developed in the following section.

D. *Eyes detection*

In this paper, Viola and Jones method is used to train two classifiers: one for opened eyes and the other for closed eyes. This method involves using the Adaboost algorithm to form a strong classifier out of the weak ones. The weak classifiers chosen in the method are the Haar Features.

E. Haar features

The features used in this work for building the eyes classifier are the Haar Features. These features involved in the disposition of rectangles used to describe a certain object. There are 160,000 features and based on the number of rectangles, they are divided into three categories. For more details about Haar features implementation, refer to [22].

F. Decision making

In this work, we based our decision-making process on the PERCLOS indicator on PERcentage of eyes CLOSure. In fact, PERCLOS is the percentage of duration with closed eyes in a time interval. Ocular measure that has proven very effective is widely used to indicate drowsiness. Then, we used the classifier for opened eyes. In case opened eyes are detected, we incremented the number of frames with opened eyes; Otherwise, we used the classifier for closed eyes (Figure 5). This procedure was repeated for each frame for an interval of three minutes. Then, we used Equation 2 to measure the PERCLOS. If the value of the PERCLOS surpasses a threshold of 25%, it means that the driver is tired, and an alert should be generated.

$$PERCLOS = \frac{Frames\ with\ closed\ eyes}{Frames\ with\ opened\ eyes + Frames\ with\ closed\ eyes} \qquad (2)$$

We chose this procedure to measure the PERCLOS because the driver's eyes are opened most of the time. Thus, the probability that the frame contains opened eyes is higher than that it contains closed eyes. Therefore, it would require less computations and time to start searching for opened eyes, then the closed ones. This is followed by doing both for each frame or to start with closed eyes search.

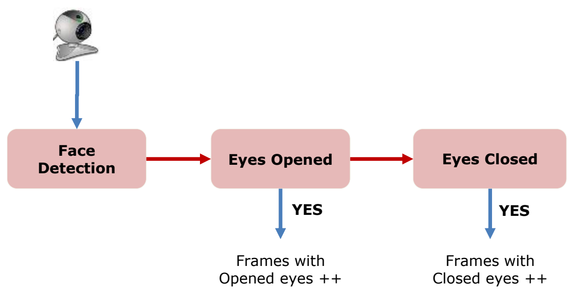

Figure 5: Decision Making

G. Multi-CPU/FPGA Based Profiling:

Once the application for drowsiness detection was validated on Visual Studio, we did a profiling step. The profiling of an application is to measure the execution time for each task in the application. This allows us to detect which parts should be passed to the Programmable Logic (PL) as hardware accelerators. The results of the profiling step are shown in Figure 6. The task that takes the most of the execution time is the eyes detection part that uses Adaboost, which is predictable since machine learning algorithms are known to take significant execution time. However, Adaboost algorithm has been already migrated for a HW implementation and presented in earlier work [20] [23]. Therefore, the next candidate for hardware acceleration was the Skin segmentation part, which represents 22% of the whole execution time of the application.

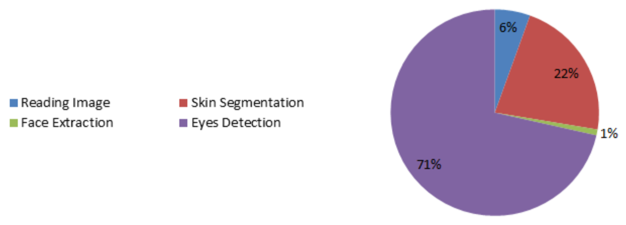

Figure 6: Application Profiling (Execution Time)-Full Software

V. HW/SW Based Co-Design

The main struggle in Hardware design is to balance between speed and resource consumption. It is important to use the available resources efficiently in order to meet the targeted speed requirements, while maintaining a good match between the architecture and the algorithms for implementation [24] [25].

An IP (Intellectual Property) core is a block of preconfigured logic or data used in making a field programmable gate array (FPGA). The performance of the design is strongly influenced by the choice and the configuration of its building blocks (IP cores) and the connections between them. It is also important to design customized IP cores efficiently for image processing. In the current design, the frame capture from the camera is first stored in the external DDR memory, then transferred to the programmable logic through the HP (High Performance) interface. Once the data is available for the AXI VDMA (Video Direct Memory Access) IP core, it is converted into a stream. Then, the data stream is processed using the image processing IP for the skin segmentation and stored again in external memory using the VDMA before being used by the application running on the processor.

A. PL IP Cores

The Processing System 7 IP core is the software interface around the Processing System. It forms the logic connection between the PS and the PL and helps integrate the different IPs with the processing system and link the PS-PL interface signals to their corresponding IPs in the programmable logic. Besides, the GUI-based PS instance allows enabling or disabling the I/O ports and peripherals, configuring PL Clocks, interrupts and PS internal clocks and generating PS configuration registers. The Vivado design tool generates a wrapper that instantiates the processing system section of Zynq Programmable SoC for the programmable logic and external board logic. It includes unaltered connectivity and, for some signals, some logic functions. These configurations are also used to initialize associated PS registers in the "**ps7_init.tcl**" or First Stage Boot Loader (FSBL). Therefore, the FSBL configures the design including the PS and PL. It is configured to use:

- Two HP (HP0 and HP1) and GP0 ports used to transfer data between PS and PL. We tried the same HP port for both the HDMI and the skin segmentation, but there was an interference and the screen monitor started to glitch, so we separated the two channels.
- UART to connect serially to the board.
- Memory controller to directly connect AXI_HP to DDR interface.
- Two Fabric clocks that serve as frequency sources to the PL IPs.

- Two PS reset signals.
- Three asynchronous Fabric interrupts issued by the PL are routed to the PS.

B. Skin Segmentation: HW/SW Co-design

As mentioned before, we used Vivado HLS to create the skin segmentation IP. The image processing IP contains the hardware function to do the skin segmentation function. It takes an RGB image as input and outputs a binary image as a result of the segmentation. The design was done using Vivado HLS (High Level Synthesis), which allowed transforming C++ OpenCV functions into Hardware Description Language (HDL).

Since the data throughput is limited by the data access pattern, multiple memory accesses represent the main bottleneck of image processing. Therefore, stream processing can overcome this bottleneck. In fact, when streaming data, the input is read once from memory then passed directly to be processed by a sequence of operations rather than writing the results back to memory after each operation. Consequently, the benefits of using video streaming in the three IPs could be significant. With the stream-based approach, accessing pixel in an image is not simple and flexible. In fact, the function AXIvideo2Mat consumes the AXI4 stream data and fills it in the image of mat format. An operator FI loads sequentially a pixel from an image and saves it to Scalar<N, T> having the same channels and depth as the image. The operator FI stores sequentially, in an image, a pixel, having the same channels and depth as the image. The function Mat2AXIvideo converts the Mat format of data to AXI4 stream. Another inconvenience for using stream-based approach is that once the data has been read from a stream, it is not in the stream anymore. Therefore, if the data from the stream is required again, it must be cached.

C. Design Optimization:

Depending on the programming technique with Vivado HLS, the implementation resulted in the derivation of a set of characteristics, principally in terms of latency and used resources. For instance, assuming that we want to calculate the average of an input array containing 10 numbers. We would have nine addition operations followed by a multiplication by 0.1 to calculate the average. There are three different ways to implement these operations in hardware and each implementation has its characteristics:

- The first implementation focuses on the hardware resources and uses only one adder and one multiplier, and has a latency of 11 clock cycles. This is because a new operation cannot start until the last one has finished
- Based on the target technology, the HLS process determines that three addition operations can be scheduled per clock cycle, while meeting the timing constraints.
- This means that we will be using more hardware resources but we reduce latency and increase the throughput.
- The final implementation possibility focuses on the latency rather than the resources. In fact, if DSP48x slices are used in place of fabric resources, all operations can take place within one clock cycle. This corresponds to the costliest implementation in terms of resources, requiring nine DSP48x slices in total (one each for the first eight additions, with the last addition and multiplication combined into a single DSP48x slice); however, it results in a latency of only one clock cycle. We used this possibility in order to meet the timing constraints; otherwise, Vivado HLS may insert pipelining registers to meet the timing.

By default, the HLS process focuses on the hardware resources, and therefore it uses the first possibility. This would result in long latency and low throughput, which may not meet the requirements of the application. However, when designing with vivado HLS, there are ways to constrain and direct the HLS processes of scheduling and binding, and thus optimizing in a different way. In fact, there are two ways to control the High-Level Synthesis process and influence the results:

Constraints: The designer may impose certain constrains on the synthesis process, such as specifying the minimum clock period. This makes it easy to ensure that the implementation meets the requirements of the system into which it will be integrated. On the other hand, he can also place constraints on the hardware utilization.

Directives: There are various types of available directive, which map to certain features of the code, enabling the designer to dictate, for example, how the HLS engine treats loops or arrays identified in the C code, or the latency of particular operations. Therefore, with the knowledge of the available directives, the designer can optimize according to application requirements.

Data flow directive: In a Data flow region, memory channels are inserted between tasks, and each task is executed as soon as its input data is available. They could be implemented using Ping-Pong or FIFO buffers.

Function inline: This directive removes all function hierarchy. It is used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.

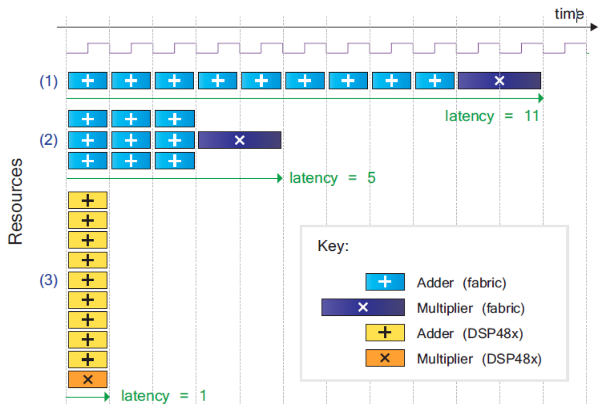

Figure 7: Three possible outcomes from HLS for an example function

Pipelining: Pipelining is a method of increasing the concurrency of the hardware produced, and thus improving the throughput. It refers to the segmentation of logical processing stages. These stages can each process different data simultaneously. This means that data dependencies that group operations are broken up, which would allow parallel execution of the stages. In hardware terms, the method used to achieve this separation of stages is to insert registers between the new, smaller stages, which allows data samples to be held in memory. The insertion of pipeline registers may also cause an increase in the maximum supported clock frequency. Figure 7 shows an example of three possible outcomes from HLS for an example function.

VI. IMPLEMENTATION AND EXPERIMENTATIONS RESULTS

A. Hardware Architecture:

The hardware architecture is illustrated in Figure 8, it is a Vivado based and shows the different hardware blocks around the Processing System (two processors ARM) [26].

The Processing System IP core is the software interface around the Processing System (PS). It forms the logic connection between the PS and the PL and helps integrate the different IPs with the processing system and link the PS-PL interface signals to their corresponding IPs in the programmable logic. Besides, the GUI-based PS instance allows enabling or disabling the I/O ports and peripherals, configuring PL Clocks, interrupts and PS internal clocks and generating PS configuration registers.

The Vivado design tool generates a wrapper that instantiates the processing system section of Zynq Programmable SoC for the programmable logic and external board logic. It includes unaltered connectivity and, for some signals, some logic functions. These configurations are also used to initialize associated PS registers in the *ps7_init.tcl* or First Stage Boot Loader. Therefore, the FSBL configures the design, including the PS and PL. It is configured to use:

- Two HP (HP0 and HP1) and GP0 ports used to transfer data between PS and PL. We tried the same HP port for both the HDMI, the skin segmentation but there was an interference, and the screen monitor started to glitch, so we separated the two channels.
- UART to connect serially to the board.
- Memory controller to directly connect *AXI_HP* to DDR interface.
- Two Fabric clocks that serve as frequency sources to the PL IPs.
- Two PS reset signals.

Three asynchronous Fabric interrupts issued by the PL are routed to the PS.

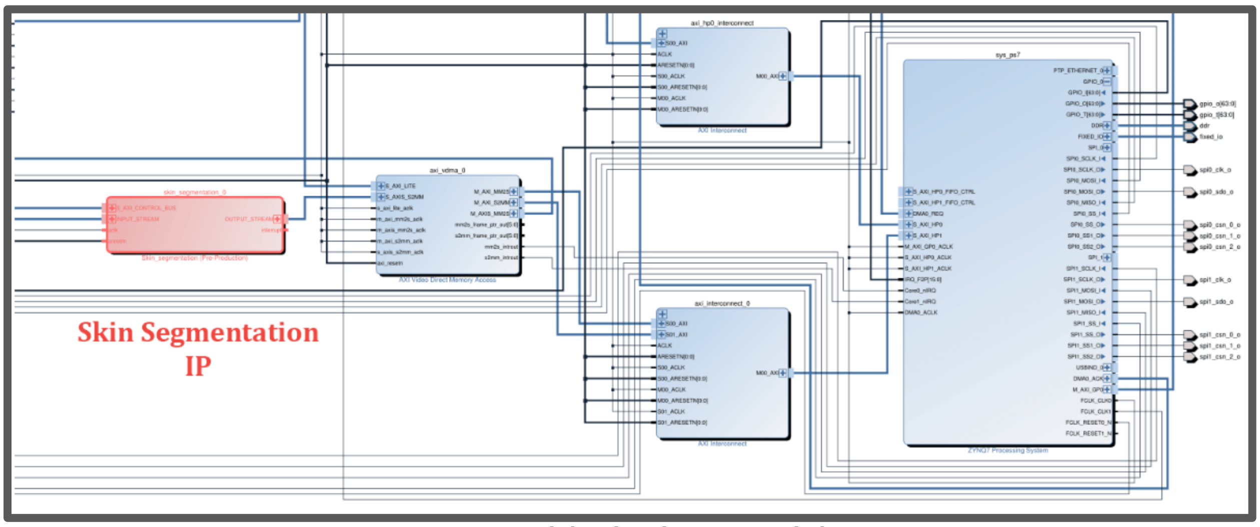

Figure 8: Vivado based Hardware Design Blocks

B. System Integration

The goal of this work is to have an application for the driver drowsiness detection running on top of an OS and communicating with the added hardware accelerator (Figure 9).

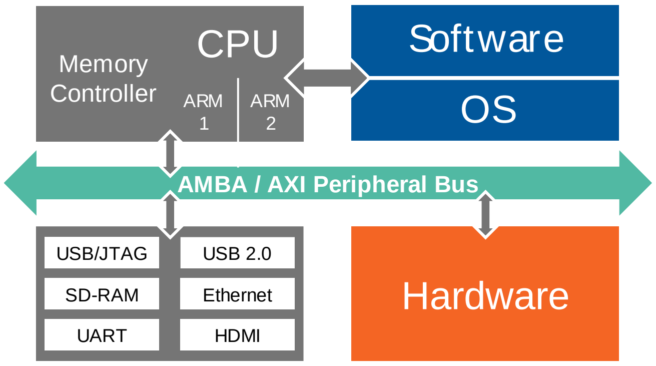

Figure 9: HW/SW Architecture

However, before building the application on the OS, a validation step was conducted for the added IP bloc of skin segmentation. This validation is done in the Bare-Metal mode [11].

Figure 9 represents the platform architecture consisting of two main parts: The Software/OS and the Hardware. The Software/OS part represents the application driver running on a specific OS in order to control the hardware accelerator. The Hardware part represents the FPGA in which the hardware accelerator is designed. Those two parts are communicating through the platform system bus to exchange data.

C. Bare Metal Application:

A Bare-Metal is a software system without an operating system. This kind of applications is developed using SDK and is written in C using drivers for each IP. The drivers are provided when exporting the design from Vivado design suite to SDK. In fact, Vivado design suite creates:

- A hardware description XML file presenting the processors, peripherals, memory map information
- A bitstream file containing the data used to program the PL with custom logic.
- These files are then exported to created SDK
- PS configuration data used by the First Stage Bootloader (FSBL).
- A collection of libraries and drivers forming the bare-metal Board Support Package (BSP). It represents the lowest layer of the application as shown in Figure 1.

The bare metal application is developed, debugged, and deployed using the hardware platform data and the BSP in a single-threaded runtime environment. It consists of a series of instruction to read from and write to the PL registers:

- To write in a register: *XilfiOut32* (Register_Adress, Value);
- To read from a register: *Xil_In32* (Register_Adress);

D. Linux Boot process:

The processor first solution is Zynq. The first step is to boot the CPU in the PS so that it will be in charge of configuring both the PS and PL. Zynq supports four master boot methods [27], which are:

- QSPI (16MB, 50MB/Sec)
- NOR (64MB, 20MB/Sec)
- NAND (tested up to 1 GB, 10MB/Sec)
- SD (Up to 32GB)

and one slave boot method, namely the JTAG for debug and development used for standalone applications. Since we are using an OS, the boot method used was SD boot. Linux is an open-source operating system. Linux consumes a small amount of memory throughput. However, its overhead could be negligible since the processing speed of the Cortex A9 is important. Linux runs on both Arm cores (Symmetric Multi-Processing). We used a Linux operating system because it includes many features such as symmetric multiprocessing, which allows communication between the two CPUs. Besides, Multitasking allows us to run many concurrent processes and threads on the system. The Zynq boot process comprises three required stages before executing Linux kernel. The Zynq boot process can be divided into three high level stages:

Stage 0: BootROM

On power up, the CPU0 begins the stage boot process and starts executing code from its boot ROM. Boot ROM initializes Cortex A9 CPU 0. CPU0 checks the CRC on ROM code and reads the boot mode pins to determine the stage 1 boot mode. If it is in SD mode, the SD interface must be connected to pre-defined MIO pins. The FSBL (First Stage Boot Loader) is typically stored in this external non-volatile memory. The CPU0 copies this file to OCM (192KB max).

Stage 1: First Stage Boot Loader

In the second stage, the FSBL uses the PS7 Init data to initialize the processing system blocks (PLL, external memory controller and MIO). It optionally configures the PL with Bitstream. Besides, the FSBL loads the second stage boot loader to the DDR memory and transfers execution to it. FSBL can be generated automatically using the template provided by SDK.

Stage 2: Second Stage Boot Loader (U-Boot)

The third stage is characterized by loading the Linux kernel. When the processor is powered on, the operating system is on the SD card and not in the memory. The U-Boot is an open source GPL cross-platform bootloader that locates the OS and brings it into memory. Then, it starts the Kernel image from memory and passes the device tree to Linux. Besides, U-Boot provides network access, reads and writes arbitrary memory locations and configures and accesses hardware peripheral devices. Toolchain is an essential part in any build system. It is the main programs that creates the arm binaries from the source codes. These programs are built to do cross compiling, which means to generate a code for a processor architecture other than the one in which the tool is running. The cross Toolchain used for Linux is from the GNU compiler collection.

The U-Boot source code and build scripts are fed to the cross Toolchain that outputs the arm binaries. The U-Boot is downloaded using Git and built as described in the wiki [12].

E. Boot Image

The Zynq boot image is created using SDK. It contains the FSBL, the hardware bitstream and the U-Boot. The Bootgen is used for constructing boot image. It merges the .BIT and .ELF files into a boot image using a .BIF file.

F. The device tree

The device tree is a file that describes the hardware to the Linux kernel so that the drivers get the information required to operate properly. The information is used by Linux during the kernel boot process to map device parameters, such as device type, memory location and interrupt signals. The kernel initializes the driver for each device during the boot process. In the device tree, the physical address of IP cores is mentioned as well as the priority of the interrupts used by a device driver. Besides, the device tree contains the boot arguments that configure the kernel at boot time. It includes the information about the location of the file system to load. A sample of the device tree that contains information about IP block is as shown in the following Figure 10:

```
skin_segmentation_0 : skin_segmentation@43c00000 {
        compatible =  "xlnx, skin—segmentation—1.0" ;
        reg = <0x43c00000 0x10000>;
        xlnx , s—axi—control—bus--add--width = <0x5>;
        xlnx , s—axi—control—bus—data—width = <0x20 >;};
```

Figure 10: Device Tree for the Skin Segmentation IP

The Device tree DTS file is compiled with the DTC device tree compiler to create a Device Tree Binary, which is a machine-readable binary.

G. Linux Application

A Linux application is developed to configure and control the different drivers, to run the skin segmentation functions and to use the resulting image in our main application. First, the Linux application initializes the VDMA and the *Skin_Segmentation* IPs. Then, it receives the image either from a webcam stream or from a stored image. After that, it uses the VDMA unit to load the image from memory and turn it into a stream of data that will then be passed to the skin segmentation unit. In fact, the application writes the input for the AXI VDMA frame buffer. Therefore, the hardware will automatically begin processing. The result is then written in VDMA frame buffer. As soon as the VDMA finishes writing the resulting frame in memory, the software starts processing it.

In order to accomplish these tasks, drivers for both the VDMA and our image processing IP are used. Two kinds of drivers are used: Kernel drivers and User-space drivers. Kernel driver are compiled when creating the kernel image and the User space drivers are included in the Linux application. User-space drivers are used more because they are simpler and easily modified. In fact, since Kernel drivers are added to the kernel image, any modification of the driver requires the recompilation of the kernel image which takes a long time. Hence, we opted for the user-space drivers.

H. *Drivers*

The drivers provide access to physical hardware resources Figure 11. Many drivers are used to configure, run and control the different node in the design and to manage memory accesses.

```
int devmem = open ( "/dev/mem" , O_RDWR | O_SYNC) ;

uint32_t * vdma0 = ( uint32_t *)mmap(NULL, MAP_LENGTH, PROT_READ | PROT_WRITE,
MAP_SHARED, devmem, ( of f_t )VDMA_0_BASEADDR) ;

// Read out VDMA 0
            p r i n t f ( "============ VDMA 0 ============\n" ) ;
            for ( o f f s e t=0x00 , i =0; o f f s e t <=0xF4 ; o f f s e t+=4, ++i )
            {
             value = in32 (vdma0 , o f f s e t ) ;
                           p r i n t f ( "0x%02x ⍰   %s = 0x%08x\n" , o f f s e t ,
            vdmaRegisterNames [ i ] , value ) ;
             }
             munmap(vdma0 , MAP_LENGTH) ;
            c l o s e (devmem) ;
            return 0 ;
```

Figure 11: Device Tree for the Skin Segmentation IP

VII. Results

In this section, we share our results regarding this research work. We used two classifiers. One for detecting opened eyes and the other for closed eyes. We used the classifier provided by the library OpenCV for open eyes since it showed satisfying results, and we built our own classifier for the closed eyes. We used the "Closed eyes in the wild" database as samples for closed eyes, which contains 2423 subjects from different origins, and we gathered our own database for background containing 3063 images that do not contain closed eyes. The method was on a series of videos and in real time with a camera, and it managed to detect drowsiness states.

A. *Booting files*

We used a pre-built Ubuntu-based OS by Linaro. The Ubuntu image is re-compiled using ARM-cross-compiler. The Linux kernel runs on the dual core Arm-9 Cortex processor chip. In addition, a fully featured desktop from Ubuntu/Linaro contained in the root file system, allows the ZC702 to work as a personal computer using a USB Keyboard and mouse, along with an HDMI monitor. In Table 2, we presented the code size of the different software files, Ubuntu image, the boot file created using SDK design tool, and the device-tree with the different drivers for the Zynq-based architecture.

Table 2
Software Code Size

Embedded OS	Kernel	Boot	Device Tree
Linux-Linaro	3.09 M-Bytes	4.16 M-Bytes	10 K-Bytes

B. *IP acceleration*

We have tested the acceleration accomplished by this hardware implementation on a series of 750x450 images from the internet and we recorded an average of 11.57 times improvement of execution time of the skin segmentation step. The values of the hardware and software average execution times are shown in Figure 12 and both have been measured on the board's 666 MHz ARM processors.

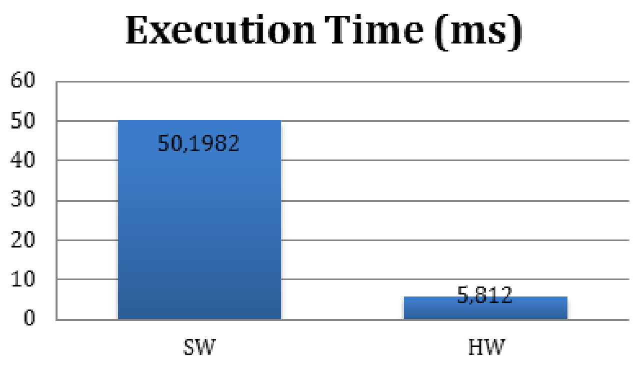

Figure 12: Hardware Acceleration

This acceleration in execution time is due to the inherent parallelism in the algorithm. Inherent parallelism implies the property of a system that allows one to decompose a software task into elements that can run concurrently. This is a characteristic of FPGA. However, not all algorithms can be accelerated with FPGA. In fact, it is easier to parallelize a streaming input application (e.g. image processing) than a finite state machine. For some algorithms, radical acceleration can be achieved by using wholly different algorithms, which are better aligned to the massive parallelism of an FPGA compared to an inherently sequential processor methodology.

C. *HW/SW results comparison*

The results on a 750x450 image are shown in the Figure 13, where we have the original image before segmentation as shown in Figure 13-(c). The results of the skin segmentation is produced by our hardware block as shown in Figure 13-(a) and with OpenCV functions Software as shown in Figure 13-(b) both on the zc702 board and the comparison between the two results (Figure 13-(d)).

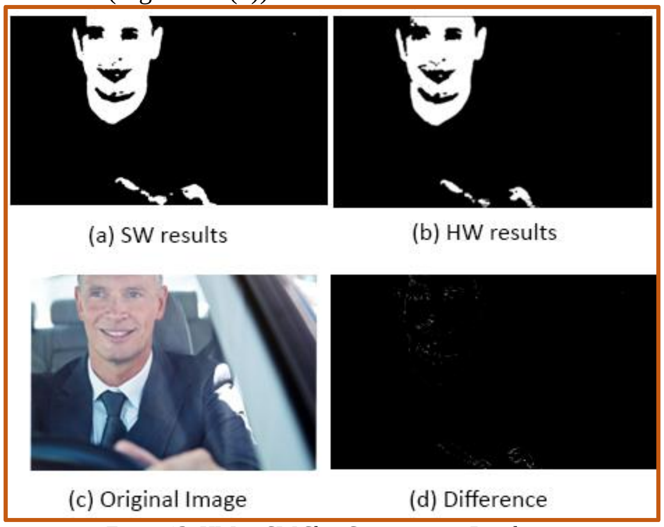

Figure 13: HW vs SW Skin Segmentation Results

Figure 13-(d) shows that our hardware acceleration for the skin segmentation task did not affect the result.

D. *Resources consumption*

One of the most important things that we need to pay attention in HW/SW partitioning is the hardware consumption because hardware is expensive. In fact, you pay a large price in transistors or other real-world costs for every twist and turn in the design. Our IP block (yellow parts highlighted in Figure 6) consumed a tiny portion of the hardware resources provided by the ZC702. It used 3% of the Look-Up Tables (LUT) around 1% of the Flip Flops (FF) and nearly 3% of the Digital Signal Processing (DSP), as shown in the Table 2. The parts highlighted in blue are the parts consumed by other blocks, such as HDMI block for the screen monitor or the SPDIF block. These blocks are just used to validate the design and will not be needed for the main application.

VIII. CONCLUSION

In this paper, we shared our experience presenting a Multi-CPU/FPGA platform-based design approach for fast HW/SW embedded smart Advanced Driver Assistant System (ADAS) design and prototyping, as an alternative for the pure time-consuming simulation technique. Based on a Multi-CPU/FPGA platform, we introduced a new methodology and a flow to design the different HW and SW parts of the ADAS system for smart vehicle applications. Then, we have implemented a vision based ADAS for driver fatigue detection. During the design process, we built a HW/SW architectural decision to achieve the optimal balance between performance, area, and power consumption. The implemented solution is used to show the efficiency of the proposed Multi-CPU/FPGA based HW/SW co-design approach, and to evaluate the cost and benefits of each one. We first started by implementing the image processing pipeline using Vivado IDE. Then, the bare metal application is developed to configure and control the different implemented IPs. After that, the custom contrast enhancement IPs are developed added to the image processing pipeline and the design is deployed on the ZC702 board. Finally, the hardware and software design are implemented and tested on the board and the results are analyzed. Future work should focus on the automation of the design steps of ADAS systems.

REFERENCES

[1] "How VW Fuel Injector Works: A Mini-Computer Aids Economy, Cuts Pollution" Chicago Tribune, Sunday, February 25, 1968

[2] A. Eskandarian and A. Mortazavi, "Evaluation of smart algorithm for commercial vehicle driver drowsiness detection," Proceedings of the 2007 IEEE Intelligent Vehicles Symposium, pp. 553–559, 2007.

[3] http://asirt.org/initiatives/informing-road-users/road-safety-facts/road-crash-statistics

[4] Pedro U. Lima, Aamir Ahmad, André Dias, André G.S. Conceição, António Paulo Moreira, Eduardo Silva, Luis Almeida, Luis Oliveira, and Tiago P. Nascimento. 2015. "Formation control driven by cooperative object tracking. *Robot" AutonSyst.* 63, P1 (January 2015), 68-79. DOI: http://dx.doi.org/10.1016/j.robot.2014.08.018

[5] https://international.fhwa.dot.gov/ipsafety/ipsafety.pdf

[6] http://en.wikipedia.org/wiki/Autonomous car, "Autonomous car."

[7] S. Russel, "DARPA Grand Challenge Winner: Stanley the Robot!" Popular Mechanics, Jan. 2006. [Online]. Available: http://www.popularmechanics.com/technology/robots/a393/2169012/

[8] T. C. Frankel, "What it feels like to drive a Tesla on autopilot," The Washington Post, Feb. 2016. [Online]. Available: https://www.washingtonpost.com/news/theswitch/wp/2016/02/01/what-it-feels-like-to-drive-a-tesla-on-autopilot/

[9] http://www.vedecom.fr/domaines-de-recherche/?lang=en/#PROJETS

[10] Nurvitadhi, E., Subhaschandra, S., Boudoukh, G., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., Liew, Y.T., Srivatsan, K., Moss.D. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?", Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17. Presented at the t2017 ACM/SIGDA International Symposium, ACM Press, Monterey, California, USA, pp. 5–14. https://doi.org/10.1145/3020078.3021740-

[11] Chin-Teng, L. Ruei-Cheng, W. Sheng-Fu, L.Wen-Hung, C. Yu-Jie, C. Tzyy-Ping, J. "EEG-Based Drowsiness Estimation for Safety Driving Using Independent Component Analysis" IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, VOL. 52, NO. 12, DECEMBER 2005

[12] https://www.cdc.gov/features/dsdrowsydriving/

[13] R. Sayed and A. Eskandarian, "Unobtrusive drowsiness detection by neural network learning of driver steering," Proceedings of the Institution of Mechanical Engineers. Part D, Journal of Automobile Engineering, vol. 215, pp. 969–975, 2001.

[14] Y.Lin, H.Leng, and e. a. G.Yang, "An intelligent noninvasive sensor for driver pulse wave measurement," IEEE Sensor Journal, vol. 7, pp. 790–799, 2007.

[15] Duy Tran et al "A Driver Assistance Framework based on Driver Drowsiness Detection" The 6th Annual IEEE International Conference on Cyber Technology in Automation, Control and Intelligent Systems June 19-22, 2016, Chengdu, China

[16] R. Wang, Y. Wang, and C. Luo, "Eeg-based real-time drowsiness detection using hilbert-huang transform," in Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2015 7th International Conference on, vol. 1. IEEE, 2015, pp. 195–198.

[17] A. Tayyaba, M. Arfan Ja_ar, M. Ramzan, and M. Anwar Mirza, Automatic Fatigue Detection of Drivers through Yawning Analysis_ 2009.

[18] E. Murphy-Chutorian, A. Doshi, and M. Trivedi,_Head Pose Estimation for
Driver Assistance Systems: A Robust Algorithm and Experimental Evaluation 2007.

[19] B.Senouci, H.Rouis, D.S.Han and E.Bourennane "A Hardware Skin-Segmentation IP for Vision Based Smart ADAS Through an FPGA Prototyping" 9th IEEE International Conference on Ubiquitous and Future Networks ICUFN, The 5th International Workshop on Intelligent Vehicles, Milan, Italy, July 2017

[20] B. Senouci, I. Charfi, B. Heyrman, J.Dubois, J.Miteran, "Fast prototyping of a SoC-based smartcamera: a real-time fall detection case study" Journal of Real-Time Image Processing, pp. 1861_8200, 2015

[21] G. P. Stein, E.Rushinek, G. Hayun, and A. Shashua, "A computer vision system on a chip: a case study from the automotive domain," IEEE Conference on Computer Vision and Pattern Recognition (CVPRW'05), p. 130, June 2005.

[22] Gururaj P et al "An Analysis of Skin Pixel Detection using Different Skin Color Extraction Techniques" International Journal of Computer Applications (0975 - 8887) Volume 54 - No. 17, September 2012

[23] C.Claus, W. Stechele, and A.Herkersdorf, "Autovision– a run-time reconfigurable mpsoc architecture for future driver assistance systems," Information Technology, vol. 49, no. 3, pp. 181–187, 2007

[24] Shi, W., Alawieh, M.B., Li, X., Yu, H., 2017. Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey. Integration 59, 148–156. https://doi.org/10.1016/j.vlsi.2017.07.007

[25] Falsafi, B., Dally, B., Singh, D., Chiou, D., Yi, J.J., Sendag, R., 2017. FPGAs versus GPUs in Data centers. IEEE Micro 37, 60–72. https://doi.org/10.1109/MM.2017.19

[26] www.xilinx.com\zynq

[27] http://fpga.org/2013/05/24/yet-another-guide-to-running-linaro-ubuntu-desktop-on-xilinx-zynq-on-the-zedboard/.