High Level Synthesis using Vivado HLS for Zynq SoC: Image Processing Case Studies

A. Cortés, I. Vélez and A. Irizar
Department of Electronics & Communications, CEIT and Tecnun (University of Navarra)
Manuel de Lardizábal 15, 20018 San Sebatián, Spain, (+34) 943 212800
Email: acortes@ceit.es, ivelez@ceit.es, airizar@ceit.es

***Abstract*—In this paper, image processing algorithms designed in Zynq SoC using the Vivado HLS tool are presented and compared with hand-coded designs. In Vivado HLS, the designer has the opportunity to employ libraries similar to OpenCV, a library that is well-known and wide used by software designers. The algorithms are compared in terms of area resources in two conditions: using the libraries and not using the libraries. The case studies are Data Binning, a Step Row Filter and a Sobel Filter. These algorithms have been selected because they are very common in the field of image processing and they have high computational complexity. The main benefit of the Vivado HLS tool is the reduction in time-to-market. On the other hand, when a software designer hand-codes the design, the use of image processing libraries similar to OpenCV helps to reduce development time even further because software designers are familiar with them. However, using these kinds of libraries significantly increases the necessary FPGA resources.**

I. Introduction

The complexity of today's hardware designs is growing quickly. Furthermore, this complexity has led to an increase in the lines of code in a Hardware Description Language (HDL) design. [1] indicates that digital designs on the order of one million gates require three hundred thousand lines of Register Transfer Level (RTL) code.

Although complexity is increasing, time-to-market must be reduced in order to improve design productivity. Especially in the case of FPGA platforms, time-to-market is critical since long cycles of chip design and manufacturing are avoided. Therefore, FPGA designers may accept an increase in power or cost in order to cut down design time [2].

The main goal of High Level Synthesis (HLS) tools is to reduce the time-to-market of the design. High Level Synthesis is not a novel concept. In the 1980s and early 1990s, a number of HLS tools were built for researching and prototyping: ADAM [3], HAL [4], MIMOLA [5], Hercules/Hebe [6] and Hyper/Hyper-LP [7]. At that time, the HLS methodology had not achieved wide acceptance. However, in the 1990s with improvements in RTL synthesis tools from the major EDA vendors (Behavioural Compiler from Synopsys [8], Monet from Mentor Graphics [9], Visual Architect from Cadence [10]), the first commercial HLS systems started to appear. Since then, HLS tools have continued to evolve focusing on using widely-adopted languages such as C/C++.

A large number of FPGA designs are developed using HLS tools. Moreover, we can find examples in different fields of application such as 3G/4G wireless systems [11], aerospace applications [12] and image processing [13] in real time environments.

In the image processing field, when there are intensive pixel-level operations, these kinds of FPGA designs are considered suitable for hardware implementation since hardware allows us to parallelize and thus accelerate processing. Different hardware implementations in the image processing field have been proposed in [14], [15], [16], [17], [18], [19], [20].

On the one hand, [14] and [15] present Lane Departure Warning Systems where the received image is preprocessed in FPGA. Both of them present a system based on hardware and software codesign. In [14], only the image is captured in FPGA which greatly reduces the image capture processing time with respect to the software capture. However, in [15] the capture and a preliminary filtering of the image are sent to the FPGA. In this case, the selected platform is Xilinx's Zynq SoC, which has two main parts: the processing system and the Programmable Logic which is a Series 7 FPGA. These two parts can communicate with each other through the standard AXI interface, thus facilitating hardware and software codesign.

On the other hand, [16], [17], [18], [19], [20] propose different implementations of Edge Detection in FPGA in order to achieve real-time systems.

In this paper, we present different implementations of common image processing algorithms that have high computational complexity (Step Row Filter, Data Binning, Sobel Filter) and that should be implemented in hardware in order to parallelize processing and achieve real-time systems. Moreover, these algorithms are being used to get Advanced Driver Assistance Systems in vehicles. The case studies of interest have been implemented over the Zynq XC7Z020 platform.This device is based on the Artix-7 technology, which is the cheapest one among the Series 7 FPGAs from Xilinx. We compare hand-coded implementations with Vivado HLS designs in order to discuss their benefits and drawbacks.

The remainder of this paper is organized as follows. In Section II, the features of the Vivado HLS tool are described. In Section III, several HLS implementations in the image processing field are presented as HLS case studies, and they are then compared with hand-coded designs. In this case, the analyzed case studies are the Step Row Filter and the Data Binning implementations. Moreover, HLS implementations that have used specific image processing libraries are compared

978-1-5090-4565-5/16/$31.00

to those that have not. This analysis is done for the Data Binning and the Sobel Filter implementations. Finally, some conclusions are drawn in Section IV.

II. Vivado HLS

Vivado HLS is Xilinx's HLS engine. Designers use C, C++ or SystemC in Vivado HLS, and the tool generates VHDL, Verilog and SystemC RTL descriptions from the HLS model by taking into account the input constraints. This Xilinx tool is totally integrated with the Xilinx design methodology and the generated description can be packaged as IP blocks that can be imported into the Vivado Design Suite for Xilinx Series 7 device implementation [22].

Vivado HLS has different directives and pragmas for optimizing the hardware of the generated design according to specifications, and it allows features such as the design interfaces, the required parallelization level and the data types to be designed.

The design methodology of Vivado HLS tool starts with the definition of functional specifications and input constraints. From these inputs, designers develop the C/C++ model and use pragmas and/or directives to try to fulfil the timing and area constraints. Then, the C/C++ testbench is designed in order to ensure the model works using different test vectors to obtain the corresponding outputs with C/C++ simulation. Once the C/C++ model is functionally correct, the HLS synthesis generates the RTL description.This RTL code can be verified with C/RTL co-simulation by using the previously defined testbench and test vectors. If the C/C++ simulation or the C/RTL co-simulation does not work as expected, the C/C++ model must be modified. Finally, designers can know whether the RTL design complies with the timing and area constraints by observing the HLS synthesis results. When the users' constraints are not fulfilled, designers should modify the pragmas or directives and repeat all the following steps.

III. Case Studies

In this Section, several typical image processing algorithms are presented as case studies of HLS implementations. Some of them are also hand-coded in order to compare with the HLS results. To complete the HLS comparison, some HLS implementations have been carried out using specific HLS libraries for image processing similar to the well-known OpenCV. Using these kinds of libraries facilitates the design of image processing algorithms and reduces design time even further, especially when the developer is a software expert.

A. Step Row Filter Implementation

The Step Row Filter is a 1D convolution filter that operates on each row of the original image. Its aim is to determine which pixels of the image likely represent lane markings inside a Lane Departure Warning System. The filter function is defined as: $y(r,c) = 2 * x(r,c) - x(r, c - tau(r)) - x(r, c + tau(r)) - abs(x(r, c - tau(r)) - x(r, c + tau(r)))$, where r is the row number, and c is the column number. $x(r,c)$ is the pixel value at (r,c), and $tau(r)$ is the tau value for the given row (more details can be found in [21]). In this case, $r = 0...239$ and $c = 0...319$. The filter generates high values for pixels whose intensity value is higher than its lateral neighbours (at $\pm tau(r)$) and these side values are similar. This provides high values for stripes like lane markings and generates very few false positives for other bright visual patterns that might appear in vehicles or in the road.

The Step Row Filter is pixel-level processing that consumes most of the total computational time of a Lane Departure Warning System algorithm [15]. Therefore, implementing the Step Row Filter in hardware, in this case in a FPGA, is suitable in order to accelerate processing.

Prior to implementing the hand-coded algorithms, a Matlab model was designed in order to ensure that the design's functionality is correct. Figure 1 shows a scheme of the hand-coded Step Row Filter module data–path operations. Each row of the image is stored in the FPGA. The module uses two MAC (multiplier accumulator) units in order to process two pixels in parallel. The selected pixels will be read from the Block RAMs depending on the value of tau, which is an FPGA input vector and they will be multiplied by the coefficients stored in the Look-Up-Table (LUT) called COEFs applying the corresponding mask (coeff_1=11FFFF, coeff_2=11FFFF, coeff_3=000002). The values of these coefficients (-1, -1, 2) are deduced from the filter function defined at the beginning of this Section. Each MAC unit accumulates the results from multiplying three pixels by the three coefficients. To the result of the MAC unit, the result of subtracting $[i - tau]$ from $[i + tau]$ is added. The sign of this sum depends on the sign of the subtraction result. Finally, data are compared against a threshold which is another FPGA input. Thus, the output is limited: if data are higher than the threshold (in this case the threshold was set at 90 by the SW designers), the output will be "FF" and otherwise, it will be zero. All these operations are controlled by a Finite State Machine which is the responsible for doing the sequence of operations in the correct order.

Table I summarizes the FPGA resources and the processing time of hand-coded and Vivado HLS implementations of a Step Row Filter module for a 320(H)x240(V) image and 8 bits per pixel. Under the IAB project, this hand-coded implementation has been proved in a real–time environment.

The Step Row Filter hand-coded implementation employs two DSP48s to process the filtering of two pixels in parallel. This custom filter is processed per row. Thus, in the FPGA, only N pixels for the input, N being the number of rows in the image, and N pixels for the output need to be stored in Block RAMs. Two RAM18E1s are required: one for the input, and one for the output.

In Vivado HLS, the same model that was designed previously in Matlab was developed in C/C++. The arguments of the C/C++ model are the input and output ports of the RTL generated by the tool. The Step Row Filter has two inputs :

```
unsigned char ptSrc[240][320]
unsigned char tau[240]
```

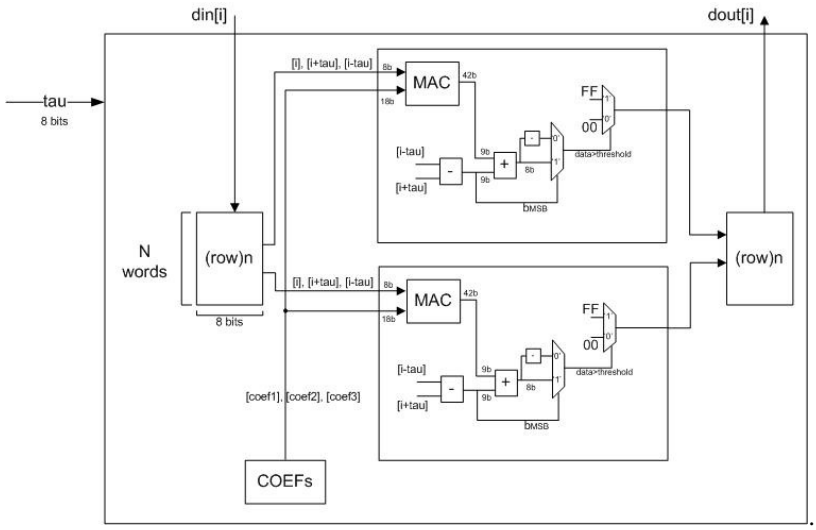

Fig. 1. Scheme of the hand-coded Step Row Filter module data–path operations

And one output:

```
unsigned char ptDst[240][320]
```

In order to optimize HDL generation, pragmas and directives were used. The following pragma was included in the C/C++ code. This was used for several variables that take part of arithmetic operations such as products and sums. Thus, DSP48 resources in the FPGA are employed to operate with those variables instead of using LUTs.

```
#pragma HLS RESOURCE variable=var_name
core=AddSub_DSP
```

As Table I shows, by using this pragma, the number of LUTs are reduced and two DSP48 are instantiated in the generated RTL code instead of using LUTs for the arithmetic operations.

In this case, no interface pragmas have been used. Therefore, the input (ptSrc, tau) and output (ptDst) interfaces have a memory interface by default.

The time needed to develop the HLS design is reduced by approximately six days relative to the hand-coded design.

Furthermore, as shown in Table I, unlike hand-coded design, HLS design does not use Block RAMs. Therefore, HLS design needs more registers and LUTs than the hand-coded implementation. The processing time of the hand-coded design working at 200 MHz is 0.95 ms. However, the minimum processing time of the HLS design working at the same clock frequency is 4.5 ms. In spite of this fact, it can be said that the generated design is faster than a SW solution. In [15], the SW solution proposed for the Step Row Filter requires 10.530 ms.

Figure 2 shows the 320(H)x240(V) input image from the Step Row Filter module and the 320(H)x240(V) output image from the filter in the different stages in HLS: C model simulation and C/RTL co-simulation.

TABLE I
STEP ROW FILTER MODULE IMPLEMENTED IN XC7Z020

	RTL Designer (hand-coded)	SW Designer (Vivado HLS)
Dev. Time (man-days)	10	4
Proc. Time (ms)	0.95	4.5
Clock Frequency (MHz)	200	200
Number of Slice Registers	145	370
Number of Slice LUTs	120	414
Number of Block RAMs (RAMB18E1s)	2	-
DSP48E1s	2	2

(a) Input of the Step Row Filter module

(b) Output of the Step Row Filter module

Fig. 2. Results of Step Row Filter simulations

B. Data Binning Implementation

The Data Binning function is defined as: $y(r,c) = 1/4 * x(2*r-1, 2*c-1) + 1/2 * x(2*r-1, 2*c) + 1/2 * x(2*r, 2*c-1) + 1/4 * x(2*r, 2*c)$, where r stands for row and c for column. $x(r,c)$ is the pixel value at (r,c).

The Data Binning module is connected to an image sensor in order to receive the measured image in RAW format ([15]). Once the image sensor is calibrated, it starts the normal operation shown in Figure 3. The timing diagram of Figure 3 presents the acquisition of a 640(H)x480(V) image. A pulse in the VSYNC signal indicates that a new frame starts. HREF is a logic one during 640 cycles of PCLK. A rising edge

in the PCLK signal marks that a pixel is valid. Thus, 640 horizontal pixels are obtained. To complete the image capture, 480 periods of HREF are needed. Implementing the Data Binning module in hardware, in this case in an FPGA, is suitable in order to accelerate processing [14].

The Data Binning module receives the camera data and processes the data binning in order to achieve the desired 320(H)×240(V) image (in this case $r = 0...239$, $c = 0...319$) from the received 640(H)x480(V) image. When two rows of the image have been received, the 320 pixels obtained are stored in Block RAMs of the FPGA.

The Data Binning module has been also hand-coded. Under the IAB project, this design has been implemented in a real–time environment.

Figure 4 shows a scheme of the Data Binning module operations. As can be deduced from the algorithm described at the beginning of this section, the design has two constant ($1/4$ and $1/6$) products (values selected by the high–level SW team). We have different control blocks implemented in the design. Control State block determines the sequence of the required operations: pixels acquisition, processing per row and storage of the processed image. The stages blocks are used to carry out the different processing according to the row because the image pattern is different for even or odd rows.

This design has a FIFO of size N, where $N = 2^n$ and 2^n is equal to or greater than c. In this case, the maximum value of c is 320, and thus $n = 9$ in order to store all the elements in a row. The output pixels are grouped in fours so the output size will be 32 bits in order to send the data through a standard communication interface such as the AXI interface.

This module receives the camera data and processes the data binning in order to achieve the desired 320(H)×240(V) image from the received 640(H)x480(V) image. In Vivado HLS, similar functionality has been implemented in order to achieve the same purpose.

Table II summarizes the FPGA resources and the processing time of the hand-coded and Vivado HLS implementations of the Data Binning module for a 640(H)x480(V) image and 8 bits per pixel.

In the hand-coded Data Binning module, the pixels to be processed are stored using Slice Registers. Slice LUTs are used to implement logic and also memory as shift registers in order to store the results of the binning operations.

HLS offers libraries for image processing, similar to OpenCV, which facilitates design by reducing development time even further. In this case, the data binning operation was implemented by using the Resize function of the hls_video_imgproc library. This function uses bilinear interpolation to reduce the size of the input image to the size of the output image.

In this case, the input and output images of the C/C++ model pass through the AXI streaming interface in order to make it easier to employ the hls_video_imgproc library. Therefore, the following pragmas are included:

```
#pragma HLS INTERFACE axis port=ptSrc
#pragma HLS INTERFACE axis port=ptDst
```

The AXI streaming image needs to be converted into Mat format, which the Resize function requires as input, via the AxiVideo2Mat function. The output image is in Mat format, and it is converted back to an AXI streaming image via the Mat2AxiVideo function.

As Table II shows, the FPGA resources using the hls_video_imgproc library increase significantly with respect to the hand-coded solution. On the other hand, the development time of the HLS design is reduced by around ten days with respect to the hand-coded design. In Table II, the total resources including the AxiVideo2Mat, Resize and Mat2AxiVideo functions, are presented. However, the resources for the AxiVideo2Mat and Mat2AxiVideo functions are only 387 LUTs and 318 FFs. Therefore, the majority of the resources corresponds to the Resize function.

The necessary processing time for our system is around 2 ms. The HLS design, working at a clock frequency of 160 MHz, achieves the target processing time. Taking this specification into account, the FPGA resources are considerably higher than the results for the hand-coded design.

TABLE II
DATA BINNING MODULE IMPLEMENTED IN XC7Z020

	RTL Designer (hand-coded)	SW Designer (Vivado HLS-OpenCV)
Dev. Time (man-days)	12	2
Proc. Time (ms)	1.89	2.05
Clock Frequency (MHz)	200	164
Number of Slice Registers	121	20379
Number of Slice LUTs	196	18136
Number of Block RAMs (RAMB18E1s)	-	6
DSP48E1s	-	36

C. Sobel Filter Implementation

The Sobel Filter is frequently used in image processing systems. This filter has great computational complexity. Therefore, it is important to optimize its implementation in order to fulfill the timing specifications of Real-Time systems. Implementations of Sobel filters have been proposed for FPGA platforms in [16], [18], and [17].

The Sobel operator is a discrete differentiation operator which computes an approximation of the gradient of an image intensity function. This is achieved by convolving each pixel value with an odd size kernel matrix in both the horizontal and vertical directions. Finally, an approximation of the gradient is calculated at each point of the image by combining both convolution results.

A Sobel Filter with a kernel size of 3 was implemented using Vivado HLS for different image lengths, namely 320(H)x240(V) and 1920(H)x1080(V), and with and without using specific HLS libraries similar to OpenCV.

Table III summarizes the FPGA resources and the processing time of Vivado HLS implementations with and without the hls_video_imgproc library from the Sobel Filter module for a 320(H)x240(V) image.

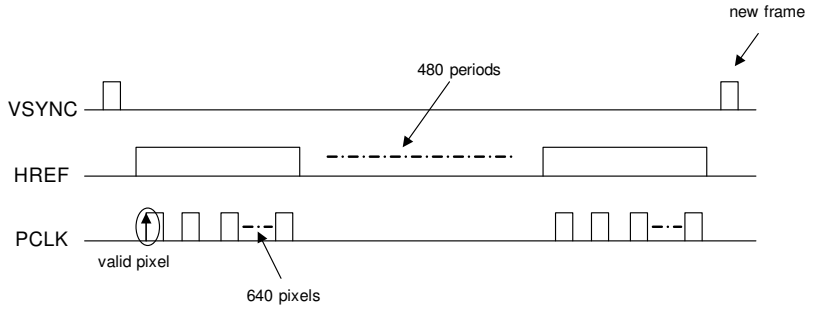

Fig. 3. Timing diagram of the CMOS image sensor

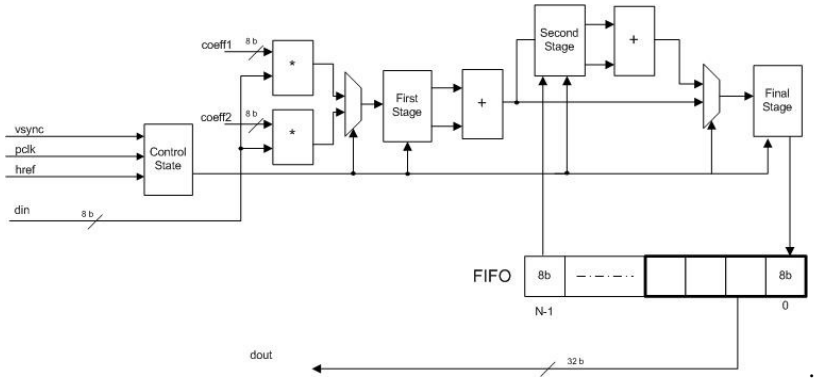

Fig. 4. Scheme of the hand–coded Data Binning module operations

A custom design (without specific libraries) Sobel Filter was developed in HLS by applying pragmas to improve the processing time:

```
#pragma AP loop_flatten off
#pragma AP dependence variable=&buff_A
false
#pragma AP PIPELINE II=1
```

The loop_flatten off pragma avoids flattening nested loops. The PIPELINE pragma permits the concurrent execution of operations inside the loop and the dependence pragma is needed in order to eliminate dependence with respect to the buff_A variable, and thus, it allows the pipeline.

The Sobel Filter was also implemented by applying the Sobel function from the hls_video_imgproc library in directions x and y over the same initial image. Before applying the Sobel function, a duplicate function is needed to copy the input image to two output images for the point of divergence for two data paths. After calculating Sobel operations in both directions, the addweighted function was used to compute the weighted per-element sum of the two images that resulted from the Sobel functions.

In both Sobel C/C++ models, the input and output images employ the AXI streaming interface. At a clock frequency of 200 MHz, the FPGA resources that use the specific HLS library increase considerably with respect to the HLS custom design as shown in Table III. On the other hand, the development time of the HLS design which uses specific image processing libraries is reduced by around eleven days with respect to the HLS custom design.

TABLE III
SOBEL FILTER MODULE IMPLEMENTED IN XC7Z020 FOR 320(H)X240(V)

	RTL Designer (Vivado HLS)	SW Designer (Vivado HLS-OpenCV)
Dev. Time (man-days)	12	1
Proc. Time (ms)	0.398	0.498
Clock Frequency (MHz)	200	200
Number of Slice Registers	748	25439
Number of Slice LUTs	1111	21946
Number of Block RAMs (RAMB18E1s)	3	18
DSP48E1s	2	138

The FPGA resources used during the Sobel Filter implementation to process a 320(H)x240(V) or a 1920(H)x1080(V) image are the same. However, the processing time is affected. Working at 200 MHz, the HLS custom design needs 0.398 ms to process a 320(H)x240(V) image and 10.44 ms to process a 1920(H)x1080(V) image.

Figure 5 shows the 320(H)x240(V) input image for the Sobel module and the 320(H)x240(V) output image of the filter in the different stages in HLS: C model simulation and C/RTL co-simulation.

IV. CONCLUSION

In this paper, we implemented common image processing algorithms designed in Zynq SoC using Vivado HLS in order

(a) Input of the Sobel Filter module

(b) Output of the Sobel Filter module

Fig. 5. Results of Sobel Filter simulations

to reduce the time-to-market of the design process. Vivado HLS offers libraries similar to OpenCV, a well-known library that is widely used by software designers. The presented algorithms were implemented with and without these libraries and the area, processing time results and design time were compared and discussed. The case studies presented were the Data Binning, Step Row Filter and Sobel Filter modules. These algorithms were selected because they are very common in the field of image processing and have high computational complexity. As can be seen, the main advantage of this tool is the reduction of time-to-market. When a software designer is responsible for the design, the use of image processing libraries similar to OpenCV helps to reduce the development time even further because software designers are familiar with them. However, using these kinds of libraries significantly increases the FPGA resources needed.

Acknowledgments

This work has been partially supported by the Basque Government ETORGAI 2014-2016 program, under the IAB–El autobús asistido project and by the Ministry of Science and Innovation RETOS DE INVESTIGACION 2014-2016 program, under the REDAS: Sistemas Reconfigurables Embebidos de Asistencia a la Conducción project. This work has been possible thanks to Vicomtech's support in designing the algorithms and to the Datik-Irizar Group for their support during the project's installation, integration and testing stages.

REFERENCES

[1] K. Wakabayashi, "C-based behavioural synthesis and verification on industrial design examples", in Proc. ASPDAC'04 pp. 344–348, 2004.

[2] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 30, Issue 4, pp. 473 – 491, DOI: 10.1109/TCAD.2011.2110592, 2011.

[3] J. Granacki, D. Knapp, and A. Parker. "The ADAM advanced design automation system: Overview, planner and natural language interface", in Proc. DAC, PP. 727–730, 1985.

[4] P.G. Paulin, J.P. Knight, and E.F. Girczyc, "HAL: A multi-paradigm approach to automatic data path synthesis", in Proc. DAC, pp. 263–270, 1986.

[5] P. Marwedel, "The MIMOLA design system: Tools for the design of digital processors", in Proc. DAC, pp. 587–593, 1984.

[6] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus synthesis system", IEEE Des. Test Comput., vol. 7, no. 5, pp. 37–53, Oct. 1990.

[7] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen, "HYPER-LP: A system for power minimization using architectural transformations", in Proc. ICCAD, pp. 300–303, 1992.

[8] D.W. Knapp, "Behavioural Synthesis: Digital System Design Using the Synopsys Behavioral Compiler. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[9] J.P. Elliott, "Understanding Behavioral Synthesis: A Practical Guide to High-Level Design", Heidelberg, Germany: Springer, 1999.

[10] A. Hemani, B. Karlsson, M. Fredriksson, K. Nordqvist, and B. Fjellborg, "Application of high-level synthesis in an industrial project", in Proc. VLSI Des., pp. 5–10, 1994.

[11] Y. Guo, D. McCain, J.R. Cavallaro, and A. Takach, "Rapid industrial prototyping and SoC design of 3G/4G wireless systems using an HLS methodology", EURASIP J. Embedded Syst., vol. 2006, no. 1, pp. 1–25, Jan. 2006.

[12] P.J. Pringee, L.J. Scharenbroich, T.A. Werne, and C.M. Hartzell, "Implementing legacy-C algorithms in FPGA co-processors for performance accelerated smart payloads", in Proc. IEEE Aerospace Conf., pp. 1–8, Mar. 2008.

[13] K. Denolf, S. Neuendorffer, and K. Vissers, "Using C-to-gates to program streaming image processing kernels efficiently on FPGAs", in Proc. FPL, pp. 626–630, 2009.

[14] Ming-Jer Jeng, Chung-Yen Guo, Bo-Cheng Shiau, Liann-Be Chang, and Pei-Yung Hsiao, "Lane detection system based on software and hardware codesign", 4th International Conference on Autonomous Robots and Agents, 319 – 323, DOI: 10.1109/ICARA.2000.4803972, 2009.

[15] G. Vélez, A. Cortés, M. Nieto, I. Vélez, and O. Otaegui, "A reconfigurable embedded vision system for advanced driver assistance", Journal of Real-Time Image Processing, 10(4), 725-739, 2015.

[16] I. Yasri, N.H. Hamid, and V.V. Yap, "Performance analysis of FPGA based Sobel edge detection operator", International Conference on Electronic Design ICED 2008, 1 – 4, DOI: 10.1109/ICED.2008.4786751, 2008.

[17] S. Singh, C. Shekhar, and A. Vohra, "Area optimized FPGA implementation of color edge detection", 2013 International Conference on Advanced Electronic Systems (ICAES), 189 – 191, DOI: 10.1109/ICAES.2013.6659389, 2013.

[18] Bin Mohamed Shukor M.N., Lo Hai Hiung, and Sebastian P., "Implementation of real–time simple edge detection on FPGA", ICIAS 2007, International Conference on Intelligent and Advanced Systems, 1404 – 1406, DOI: 10.1109/ICIAS.2007.4658616, 2007.

[19] G. Chaple, and R.D. Daruwala, "Design of Sobel operator based image edge detection algorithm on FPGA", 2014 International Conference on Communications and Signal Processing (ICCSP), 788 – 792, DOI: 10.1109/ICCSP.2014.6949951, 2014.

[20] Luo Hai-bo, Jiao An-bo, Xu Ling-yun, and Shao Chun-yan, "Edge detection using matched filter", 27th Chinese Control and Decision Conference (CCDC), 1132 – 1136, DOI: 10.1109/CCDC.2015.7162087, 2015.

[21] M. Nieto, A. Cortés, O. Otaegui, J. Arróspide, and L. Salgado, "Real-time lane tracking using rao–blackwellized particle filter. Journal of Real-Time Image Processing", vol. 1, no. 1, pp. 179-191, DOI 10.1007/s11554–012–0315–0, 2016.

[22] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx Vivado High Level Synthesis: Case studies", Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET Year: 2014 Pages: 352–356, DOI: 10.1049/cp.2014.0713