A low-cost SVM classifier on FPGA for pedestrian detection

Vinh Ngo, Arnau Casadevall, Marc Codina, David Castells-Rufas, and Jordi Carrabina Bordoll

Abstract—**Support Vector Machine (SVM) classifier is an intensive computational part of a pedestrian detection system. A real-time system requires the classifier to be implemented in embedded platforms. In this paper, a hardware accelerator for the SVM classifier, which is part of the pedestrian detection system, has been designed and implemented on FPGA. The accelerator, which targets low latency and on-chip memory use, can be scaled to different input image sizes. The memory usage of the accelerator alone is 77% of the state of the art implementation. The accelerator is demonstrated by being integrated into a pedestrian detection. It increases the system's throughput by 1.9 times.**

Keywords—**Support Vector Machine (SVM), classifier, pedestrian detection.**

I. INTRODUCTION

PEDESTRIAN detection has been one of the key problems to be solved during recent years for self-driving cars. State of the art approaches employ machine learning to classify objects as pedestrians or not. Deep learning method has been used to detect pedestrians even though it requires costly computing platforms with not only many processing cores but also large memory bandwidth and capacity. Therefore SVM, a supervised machine learning method, is more likely used, especially in an embedded system. Unlike the deep learning method which uses the original input image as the training data, SVM learns from the image features such as Histogram of Gradients (HOG) [1], Scale-invariant feature transform (SIFT). After the training phase, a set of support vectors, which represents the model, is identified. Then, the model is used for the inference process that outputs confidence values for every object in the input image. Intuitively, a window, which contains support vectors, slides and calculates convolution at each step over the whole image features to generate confidence values. A threshold value is chosen so that an object can be classified as a pedestrian if its confidence value is greater than the threshold. This sliding window task can be easily implemented in software by nested loops. Its latency, however, takes almost 50 percent of the whole detection system time. The reason for this is that the CPU has to wait for the entire feature of the image to be available and calculate sequentially for thousands of sliding windows. An FPGA is well fitted in this sort of sliding and computing operation as the convolution process and the feature generating process can run in parallel using pipelines. Furthermore, the convolution is executed in parallel for different window positions. In this paper, we propose a hardware implementation of an SVM classifier on FPGA. The design is then integrated with our previous HOG extractor design to realize a real-time pedestrian detection system. The design has helped to double the detection throughput. Besides it only uses 6.3 percent of memory size that needs to store all the confidence values of an image.

The next section will provide some key related works. The detail implementation of the accelerator is presented in section III. Section IV shows the results and comparisons. And finally, a conclusion is given in section V.

II. RELATED WORKS

Sliding window is the key computational part of an SVM classifier. It has been used as a reference application in [2] to compare the performance of multi-core CPU, GPU and FPGA. According to the paper, FPGAs is 11x and 57x faster than GPU and CPU respectively while consuming orders of magnitude less energy. A review on SVM accelerator on FPGA for several applications including pedestrian detection is presented in [3]. Among various implementations, the work in [4] has reported not only a high throughput system but also a detail on hardware resources required by each main part of the pedestrian detection, including the SVM module. The author has created an architecture so that every newly created HOG feature of a specific block is processed at once even though the feature of a block contributes to the final confidence values of up to 105 windows. Therefore, the calculation for up to 105 windows can be done in parallel, which helps to increase the throughput. Besides, the limited on-chip memory does not need to be used for all the HOG feature blocks of a frame. Actually, our SVM architecture is inspired by the one in [4]. However, with a memory optimizing target, our system's on-chip memory usage is even reduced by 23 percent. A recent work, which provides a report on SVM hardware resource, is presented in [5]. The SVM implementation in this work is quite the same as the one in [4]. However, to keep the control circuit simple, the author uses on-chip memory to store all the confidence values for all detection windows. Our architecture only stores 6.3 percent of the total number of detection windows without any hardware resource overhead.

III. SVM BACKGROUND

This section provides a background on SVM that closely relates to the hardware implementation of this research. Basically, SVM is consist of two phases, training and classifying. Owing to the high computational complexity, the training phase is very unlikely used in a real-time embedded system. In fact, the training phase is run offline and it generates a model for the classification phase.

A. Training phase

The input of a training phase is training data, which, in this case, is the set of HOG feature vector of every training image and their corresponding labels. In fact, every training image includes a fixed number of detection windows, which depends on the size of the image and the detection window. A label, corresponded to a detection window, indicates either a pedestrian or a non-pedestrian. It depends on whether a pedestrian is present in the detection window or not. Mathematically, training is the process of solving the Equation (1) [6], where $\vec{x}_i$ is the HOG feature vector of the window number i^{th} of the input training image and y_i is its corresponding label which could be either *1* or *-1*. By solving equation (1), vector $\vec{w}$ and b is determined. We used open source LIBSVM[7] for the training phase and thus obtaining $\vec{w}$ and b. In this work, each HOG feature vector has 3780 elements because each detection window has 7x15 blocks and each block has 36 elements [8]. And *N* is the number of input training image including both positive and negative samples.

$$y_i\,(\vec{w}\cdot\vec{x}_i + b) \geq 0\,,\; i = 1,\,\dots,\,N \qquad (1)$$

A graphical representation of Equation (1), in which $\vec{x}$ is 2-dimension vector, is shown in Figure 1 where:

$$f(\vec{x}) = \vec{w}\cdot\vec{x} + b.$$

The key idea in solving (1) is to find a hyperplane $f(\vec{x})$ that separates all input positive and negative training data. There are possible multiple solutions for Eq. 1 and SVM selects the one that has the largest margin. The training data that are closest to the hyperplane are support vectors (SV). Those are the circled ones in Figure 1.

The resulting model, generated by the training phase, is an array of weight vector $\vec{w}$, and b values. The weight vector is, in fact, the linear combination of all the SVs.

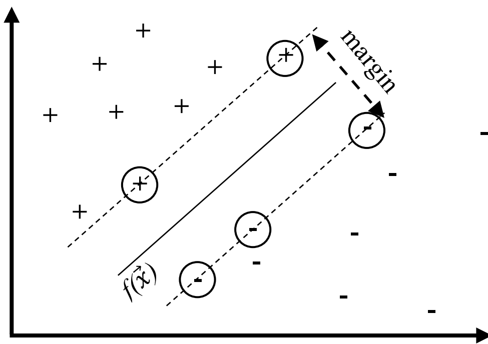

Fig. 1. A 2-D example of an SVM hyperplane

B. Classification phase

The classification phase will then use the model to infer predictions. The confidence value $y(\vec{x})$ of a detection window, which will be compared to a threshold to determine an object as a pedestrian or not, is calculated from Equation (2). Weight vector $\vec{w}$ and the bias *b* are provided by the model while $\vec{x}$ is the HOG feature vector of the window that needs to be detected.

$$y(\vec{x}) = \vec{w}^{T}\cdot\vec{x} + b \qquad (2)$$

IV. HARDWARE IMPLEMENTATION

A. Sliding window

The key factor making an SVM classifier's execution time quite long in software is the sliding window task. Figure 2 illustrates the sliding window implemented in our design. To be more specific, the size of the input image is 640x480. The HOG feature of an image is organized in blocks. Each block is a concatenation of 4 cells, and a cell is formed by 8x8 pixels as illustrated in Figure 3. To improve the detection performance, two consecutive blocks in either horizontal or vertical direction have 2 overlapped cells. Therefore, an image of size 640x480 would have 80x60 cells and 79x59 blocks.

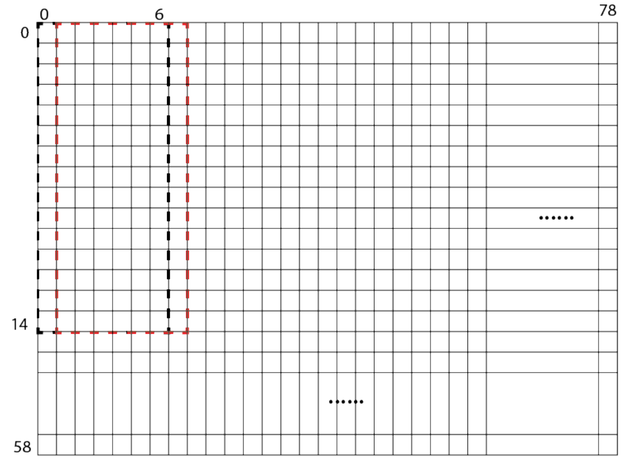

Fig. 2. Sliding a 7x15 window over a 79x59 HOG frame

The SVM classifier works with block unit. In Figure 2, the HOG feature of the input image has 59 rows and each row has 79 blocks. And a detection window has a size of 7x15 [1] blocks. Figure 2 shows two detection windows drawn by dash lines with one block sliding step in the horizontal direction. Therefore, it takes 73 steps to slide horizontally. Similarly, in the vertical direction, there are a total of 45 detection window if the sliding step is one block.

At each step, all 105 blocks containing 3780 fixed-point numbers in the detection window multiplies with the 3780 elements of the *weight vector* stored in a ROM memory. Then the sum of all those 3780 products is added to the *bias* provided by the model to obtain the final confidence value for that specific window. This process is repeated for a total of 73x45 detection windows.

B. SVM classifier hardware architecture

This section provides the hardware implementation of the SVM classifier on DE1-SOC board housing a Cyclone V FPGA device.

From the hardware point of view, there are 2 main problems to be tackled so as to speed up the execution time in compared to software implementation. Firstly, since blocks are generated sequentially, it is more efficient to process every block immediately after its being generated instead of waiting for the whole 105 blocks.

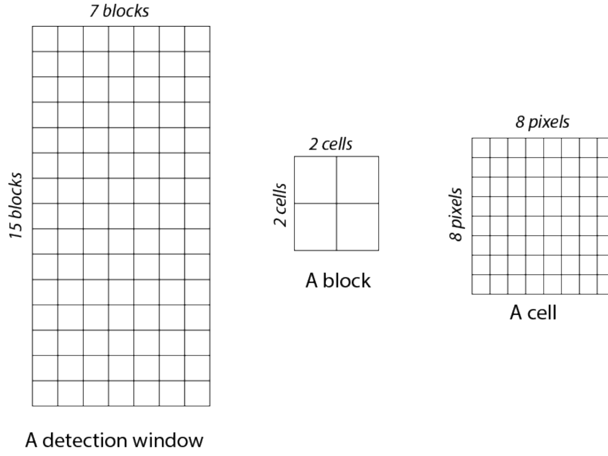

Fig. 3. Size of a detection window, a block, and a cell

It is worth noting that the latency for a full detection window blocks to be available is not just 105 times the latency for one block. This is because blocks are created row-wise and each row contains 79 blocks as in Figure 2. This approach can also save the precious on-chip memory because once the block is processed, it is no longer necessary to be stored.

Besides, the hardware design needs to consider the fact that one block is possibly used for multiple detection windows. For example, the block at position (0,1) in Figure 2 belongs to 2 detection windows. On the other hand, the block (6,14) contributes to 105 different detection windows.

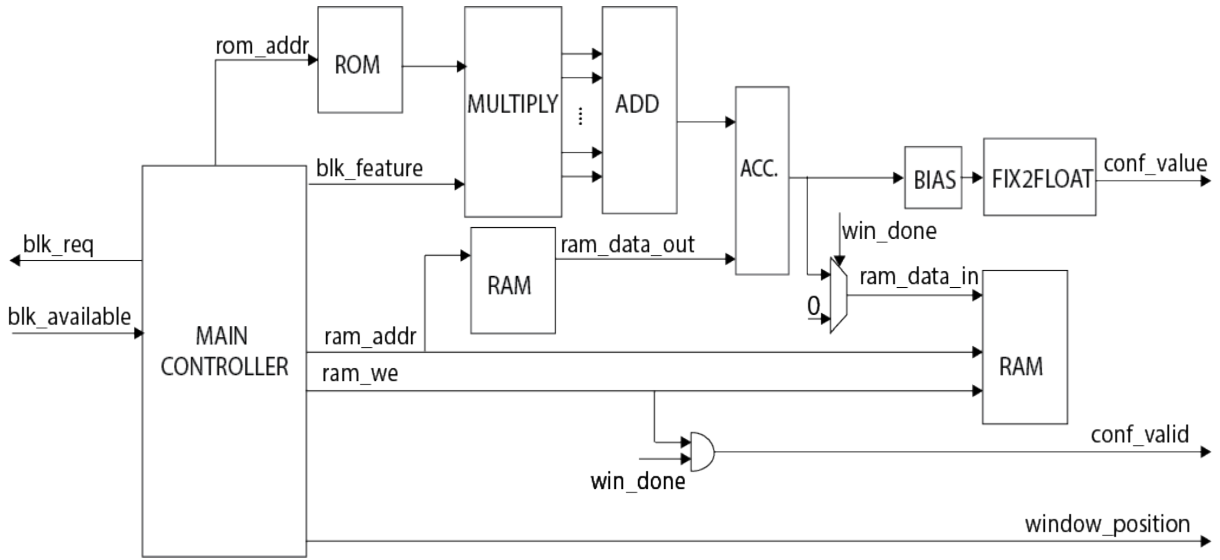

Fig. 4. SVM classifier hardware block diagram

The design, shown as block diagram in Figure 4, is structured to solve the problems mentioned. Although the hardware design is pipelined for high throughput, the pipeline registers are not shown in the figure for the sake of clarity. Each hardware components will be briefly described in the following paragraphs.

MAIN CONTROLLER: it is a finite state machine (FSM) that orchestra the whole design. Once a block is finished, it will check if a new block is available before fetching it to the pipeline. Knowing the block position, the FSM can infer how many detection windows that the block belongs to. Besides, the FSM generates appropriate addresses to access ROM and RAM memory.

ROM: this memory stores all the elements of the weight vector. If the model has any change, this ROM must be reloaded with the new weight vector. The size of this ROM is 3780 x10 bits. It means that each element of the weight vector is represented by a 10-bit fixed-point, in which 8 bits are fractional bits.

RAM: there are 2 RAM instances in Figure 4 to distinguish between the reading and the writing process. Physically, there is one unique RAM module in the design. The memory, which has a size of 30x73x19 bits, stores temporary sums for final confidence values. Each word is 19-bit width including a 12-bit partial sum and a 7-bit counter. Each resulting confidence value of a detection window is a sum of 105 partial sums. Therefore, the counter is used to signify that the detection window's confidence value is valid. The *win_done* signal is active when 105 partial sums of a detection window are fully accumulated. Furthermore, to optimize the on-chip memory usage, the memory location storing that window's value will be reused for other detection windows. Therefore, the design only uses 30x73 RAM locations to effectively store 45x73 detection windows' temporary sums.

MULTIPLY: this module takes a hog block and multiplies it with appropriate elements of the weight vector stored in the ROM memory. One-cycle multiplication will generate 36 products because a block contains 36 elements. Depends on the position of the block, it might belong to multiple detection windows. It would take 105 cycles to finish processing a specific block if that block belongs to 105 detection windows.

ADD: This module simply sums up 36 products from the MULTIPLY module.

ACC.: Since a detection window's confidence value is the sum of 105 partial values. This module accumulates the temporary value stored in the RAM memory with the new partial sum.

BIAS: This module adds the *bias* value to generate the final confidence value in fixed-point representation.

FIX2FLOAT: Fixed-point confidence values are converted to 32-bit floating-point numbers by this block.

From the right side of Figure 4, we can see that each confidence value is accompanied with a valid signal and an address indicating the position of that detection window in the image. This coordination is used by the HPS software to draw the rectangular if the confidence value is higher than the threshold.

C. Number representation

In this work, we used fixed-point numbers for all the calculations to achieve high accuracy in the detection system. The inputs to the MULTIPLY block are two fixed-point numbers, which both have 8 fractional bits. Although, the MULTIPLY generates 16-bit fractional fixed-point numbers, only 8 fractional bits are kept and provided to the ADD module. This is reasonable since keeping 16-bit fractional number increases the system resources without adding any significant difference in the system accuracy.

At the end of the pipeline, final confidence values are converted from fixed-point to floating point to avoid that task being implemented in HPS software.

D. Pedestrian detection system

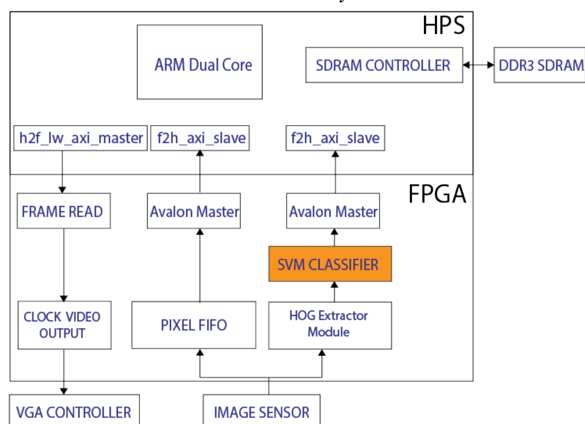

Fig. 5. SVM classifier in the pedestrian detection

The SVM classifier is integrated into the pedestrian detection system built in [8] as in Figure 5. Input images from the sensor are entirely processed on-fly by FPGA hardware pipeline. Only the final confidence values are written to the external DDR3 memory.

The confidence values are written to the external DDR3 SDRAM memory by a custom Avalon Memory-Mapped Master via the f2h_axi_slave bridge. Similarly, another custom Avalon bus master is used to send image pixels to the DDR3 memory. These two memory locations are set to be dedicated to FPGA. This transfer method provides good performance because data are transmitted in parallel with the HPS's CPU execution.

In the opposite direction, pixels in the memory and detection results are sent to the VGA controller to visualize in real time.

V. RESULTS

The SVM classifier can be parameterized to work with different input image size. The hardware resource reported in Table I is for an input size of 640x480. The SVM classifier itself only occupies 97 Kbits on-chip memory which is the smallest in the state of the art even though fixed-point numbers are used for the design. The maximum operating frequency of the design is 87 MHz, which is high enough to be integrated into the detection system. In fact, the design could be optimized to work at a higher frequency by inserting more pipeline registers into the critical paths. However, the operating frequency of the detection system, in which the SVM classifier is integrated, is just 68 MHz. Therefore, we keep the design working at this frequency to save hardware resources.

TABLE I

COMPILATION REPORT FOR CYCLONE V DEVICE.

Design	Block memory Kbits	Adaptive Logic Modules (ALMs)	DSP blocks	Registers	Fmax (MHz)
SVM classifier	67 (2%)	716 (2%)	0	684	87
Detection system	340 (9%)	13,755 (43%)	40 (46%)	17,655	68

A comparison of hardware resource usage to other implementations is given in Table II.

Our design uses less resource in the number of LUTs, DSPs, and Registers. Since the memory usage is apparently affected by the input image width, we divide the memory usage by the input image's width and get the results in the column "*Memory per width*". According to this metric, our design saves 23% and 73% on-chip memory compared with the two references.

Based on the fact that the pedestrian detection system works at 50MHze clock frequency, the SVM classifier is only constrained to work at 87 MHz. However, we could totally optimize the clock frequency to operate at a much higher frequency by inserting pipeline stages.

Besides, we opt to use logic cells for implementing the multiplication circuit instead of DSP blocks. Therefore the design does not use any DSP block.

The SVM classifier is demonstrated in the pedestrian detection system. It speeds up 1.9x the detection speed. The system's throughput is 11 FPS and 21 FPS corresponding to the system with software SVM classifier and the system with SVM accelerator respectively.

TABLE II

RESOURCE COMPARISON.

Design	Frame size	FPGA	Max frequency (MHz)	Memory per width (Kb/pixel)	FPGA resources			
					Memory (Kb)	LUTs	DSPs	Registers
[4]	1920x1080	Virtex 7	266	0.13	252	1,246	37	1,534
[5]	1920x1080	Cyclone IV	140	0.37	706	1,251	36	1259
Ours	640x480	Cyclone V	87	0.1	67	716	0	684

VI. CONCLUSION

An SVM classifier accelerator has been designed and integrated into the pedestrian detection. The detection system achieves 1.9x higher throughput thanks to the accelerator. The design is optimized to save the precious on-chip resources, especially the memory. While keeping high accuracy through fixed-point number representation, the memory cost is only 77% compared to the best implementation reported.

ACKNOWLEDGMENT

This work was supported by Spanish projects TEC2014-59679-C2-2.

REFERENCES

[1] N. Dalal and W. Triggs, “Histograms of Oriented Gradients for Human Detection,” *2005 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. CVPR05*, vol. 1, no. 3, pp. 886–893, 2004.

[2] J. Fowers, G. Brown, P. Cooke, and G. Stitt, “A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications,” *Proc. ACM/SIGDA Int. Symp. F. Program. Gate Arrays - FPGA ’12*, vol. 9, no. 4, p. 47, 2012.

[3] S. M. Afifi, H. Gholamhosseini, and R. Sinha, “Hardware Implementations of SVM on FPGA : A State-of-the-Art Review of Current Practice,” *Int. J. Innov. Sci. Eng. Technol.*, vol. 2, no. 11, pp. 733–752, 2015.

[4] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, “FPGA-Based real-time pedestrian detection on high-resolution images,” *IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, pp. 629–635, 2013.

[5] J. Dürre, D. Paradzik, and H. Blume, “A HOG-based Real-time and Multi-scale Pedestrian Detector Demonstration System on FPGA,” pp. 163–172, 2018.

[6] T. Joachims, *Learning to Classify Text Using Support Vector Machines*, vol. 29. 2001.

[7] C.-C. Chang and C.-J. Lin, “Libsvm,” *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 1–27, 2011.

[8] V. Ngo, A. Casadevall, M. Codina, D. Castells-Rufas, and J. Carrabina, “A High-Performance HOG Extractor on FPGA.”