IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 24, 2020, accepted April 7, 2020, date of publication April 21, 2020, date of current version May 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2989267

Analysis and Comparison of FPGA-Based Histogram of Oriented Gradients Implementations

SINA GHAFFARI, PARASTOO SOLEIMANI, KIN FUN LI, (Senior Member, IEEE) AND DAVID W. CAPSON, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC V8W 2Y2, Canada

Corresponding author: Sina Ghaffari (sinaghaffari@uvic.ca)

This work was supported in part by the Doctoral Fellowships from the University of Victoria, and in part by the National Science and Engineering Research Council of Canada through the Discovery Grants under Grant 36401 and Grant 04787.

ABSTRACT One of the commonly-used feature extraction algorithms in computer vision is the histogram of oriented gradients. Extracting the features from an image using this algorithm requires a large amount of computations. One way to boost the speed is to implement this algorithm on field programmable gate arrays, to benefit from flexible designs such as parallel computing. In this paper, we first, provide a summary of the steps of the histogram of oriented gradients algorithm. We then survey the implementation techniques of the histogram of oriented gradients on field-programmable gate arrays in the past decade. We group the different techniques into four main categories and analyze various enhancement methods in each category. The first group is the optimization of the algorithm computation which involves the steps of input selection, magnitude calculation, orientation and bin assignment, and normalization. The second category is data manipulation techniques which include numerical representation, data flow modification, and memory optimization. The third group contains modified features based on the histogram of oriented gradients and their hardware implementation, and the fourth one is the implementations in hardware-software co-design of the algorithm. We compare the different implementations using a speed metric called pixels per clock cycle, and resource utilization. Finally, we provide design summary tables for efficient implementation with respect to the speed metric, accuracy, and resource utilization.

INDEX TERMS Histogram of oriented gradients, field programmable gate arrays, hardware acceleration, hardware design.

I. INTRODUCTION

One of the most well-known feature extraction algorithms in computer vision is the histogram of oriented gradients (HOG). Dalal and Trigs [40] present the HOG algorithm in 2005 and over the years, this algorithm has proven to be useful in many object detection applications. The main idea behind the HOG algorithm is to compute gradients as local descriptors and normalize them locally, and then obtain location invariant features which are robust to illumination changes in the image. HOG features have many applications such as face recognition [42], [43], texture classification [44], vehicle detection [45], and human activity recognition [46]. A complete object detection model can be designed by using HOG features coupled with a classifier such as Adaboost [1] or Support Vector Machines [2], [3]. Some work focus on modified extended features based on HOG descriptors [11], [25].

Since many applications in computer vision have real-time constraints and are implemented as an embedded system, much research has been focusing on the hardware acceleration of computer vision algorithms. Although HOG has shown outstanding detection capacity, it is computationally expensive and requires extensive operations to extract the features of a single frame. Due to this large amount of computation, its software implementation on a stand-alone Central Processing Unit (CPU) may not meet performance expectations. Therefore, there have been many efforts to implement

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

the HOG algorithm (and its variants) on parallel hardware platforms such as Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Since many applications require mobility and low power consumption, the implementation of the HOG algorithm has been more popular using FPGAs than GPUs.

FPGA implementations can potentially run faster, with less resource utilization and power consumption, which are important in embedded systems that require real-time processing. Ma *et al.* [9] compare CPU, GPU and FPGA implementation of the HOG algorithm. They implement the HOG algorithm on an Intel Xeon E5520 CPU processor, an Nvidia Tesla K20 GPU and a Xilinx Virtex-6 FPGA. To process a single frame, their FPGA implementation consumes 130x less energy than that of the CPU and 31x less energy than that of the GPU, while the speed is about 68x faster than the CPU and 5x faster than the GPU.

Over the years, many researchers propose FPGA implementations for the HOG algorithm in many different applications. Table 1 shows the applications of the FPGA-based HOG algorithm reviewed in this paper.

TABLE 1. Applications of FPGA-based HOG in the surveyed references.

Application	FPGA-based HOG implementation projects
Pedestrian detection*	[5][6][7][9][12][13][17][20][21][24][26][27][28][31][38][39]
Human detection*	[1][2][4][8][11][14][29][32][34]
Car detection	[18][19][23]
Traffic sign detection	[10]
Crowd density estimation	[29]
General object detection	[3][33][36]
Object tracking	[16][25]
Feature matching	[31]
Anomaly detection	[15]
Digit recognition	[22]

* The most common applications are pedestrian and human detection.

One of the most popular applications which has used HOG features is pedestrian detection. Pedestrian detection and tracking have been employed in driving assistance, surveillance, and robotics. Other popular applications of the HOG algorithm are for human detection. The difference between pedestrian and human detection is that pedestrians are mostly standing and walking in different directions while in the case of human detection, people may be in any possible position such as playing a sport, dancing, or just stationary. Some papers demonstrate the HOG algorithm in traffic sign detection [10] and car detection [18], which are useful in autonomous vehicle systems. Blair and Robertson [15] propose an application of the HOG algorithm to locate illegally parked cars in urban areas. Other applications, as shown in Table 1, include crowd density estimation, digit recognition, and general object detection.

This paper is an extended version of our previous work [41]. We review the research work on FPGA-based HOG implementations from 2010 to 2019, collected from IEEE Xplore, Science direct, and NCBI databases. The selected papers are the most relevant articles that we could identify for HOG implementation on FPGA platforms. Table 2 presents the notations and their description used in this paper.

TABLE 2. Table of notations.

Notation	Description
θ	Orientation
BRAM	Block RAM
CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
DSP	Digital Signal Processor
FIFO	First In First Out
FIND	Feature Interaction Descriptor
FPGA	Field-Programmable Gate Arrays
G_x	Gradient in horizontal direction
G_y	Gradient in vertical direction
GPU	Graphics Processing Unit
h	Histogram vector
h_n	Normalized histogram vector
HOG	Histogram of Oriented Gradients
HSG	Histogram of Significant Gradients
HW/SW	Hardware/Software
IoT	Internet of Things
IP	Intellectual Property
LUT	Look-Up Table
RGB	Red, Green, Blue color channels
SVM	Support Vector Machine
SoC	System on Chip

The typical parameters in an embedded real-time FPGA-based system which could be optimized are speed, power, resource usage, and accuracy. There is always a trade-off between these parameters. For example, by parallelizing the design and using more FPGA resources, one can improve speed. Most of the papers in this survey focus on speed and resource utilization. Only a few of them have reported on power consumption while others have assumed that implementation on an FPGA consumes less power than GPUs and CPUs, and therefore is naturally of lower power.

In this survey, we discuss the different techniques and methods of implementing the HOG algorithm on FPGA. First, we review the HOG algorithm in section II. We then group these methods into four main categories based on different techniques to enhance HOG implementation. Fig. 1 shows the organization of the various categories of the survey provided in this paper. In section III, we present innovative methods which optimize the computation of different steps of the algorithm. In section IV, we discuss in detail the techniques in recent work which are related to data structure and manipulation. In section V, we present the FPGA implementation of some modified versions of the HOG algorithm. In section VI, we review the methods which benefit from hardware-software co-design. Finally, we have a critique on these methods in section VII, and a comparison of the results of recent papers is presented.

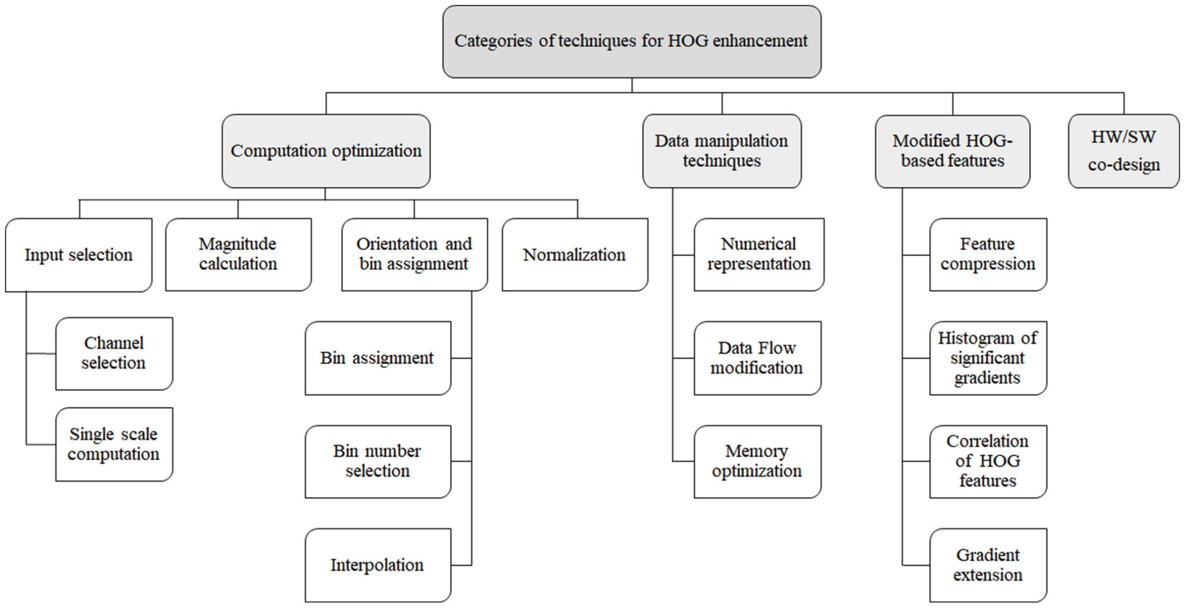

FIGURE 1. **Categorization of HOG algorithm performance enhancement.**

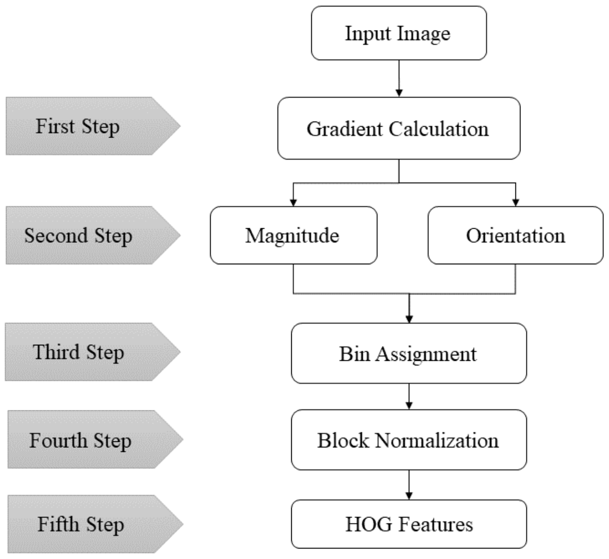

FIGURE 2. **A flowchart of the HOG algorithm. The HOG algorithm is sequential, but the second step (magnitude and orientation) can be computed in parallel.**

II. THE HOG ALGORITHM

The basic idea of the HOG algorithm is to use gradient information of each pixel to extract discriminating features for object detection. HOG features are normally extracted from various window sizes in the image. Fig. 2 shows a flowchart of this algorithm.

In the original HOG algorithm [40], the image window is divided into several blocks, and each block is divided into several cells. As an example, each block may contain four cells, and each cell may contain 16 (4×4) pixels. The first step of the HOG algorithm, as shown in Fig. 2, is to compute the gradients of each pixel in each cell, that is, to compute the derivatives in horizontal and vertical directions using the pixels around them. The gradient of the image is computed as shown in (1) and (2):

$$G_x(x,y) = I(x+1, y) - I(x-1, y) \quad (1)$$

$$G_y(x,y) = I(x+1, y) - I(x-1, y) \quad (2)$$

where I(x,y) is the image pixel with coordinates x and y, G_x is the gradient of the horizontal direction and G_y is the gradient of the vertical direction. After calculating the gradients, the second step of the HOG algorithm is to compute the magnitude and orientation of each pixel as shown in (3) and (4):

$$\text{Magnitude}(x,y) = \sqrt{G_x^2(x,y) + G_y^2(x,y)} \quad (3)$$

$$\text{Orientation}(x,y) = \tan^{-1}(G_y(x,y)/G_x(x,y)) \quad (4)$$

The third step of the HOG algorithm is bin assignment in which a histogram is created based on the calculated orientation of pixels in each cell, which could be between 0 to 180, or 0 to 360 degrees, depending on the implementation configuration. Dalal and Triggs [40] use nine bins each corresponding to 20 degrees in their original work. The magnitude of each pixel is added to the value of the bin which contains the orientation of that pixel. In order to reduce aliasing, the weighted magnitude of each pixel is added to two adjacent bins based on the distance of its orientation to the center of the bins.

In the fourth step, the histograms of cells within each block are normalized separately. Finally, HOG features are obtained by concatenating all histogram values in the selected window in the fifth step. Fig. 3 shows how the input image window is

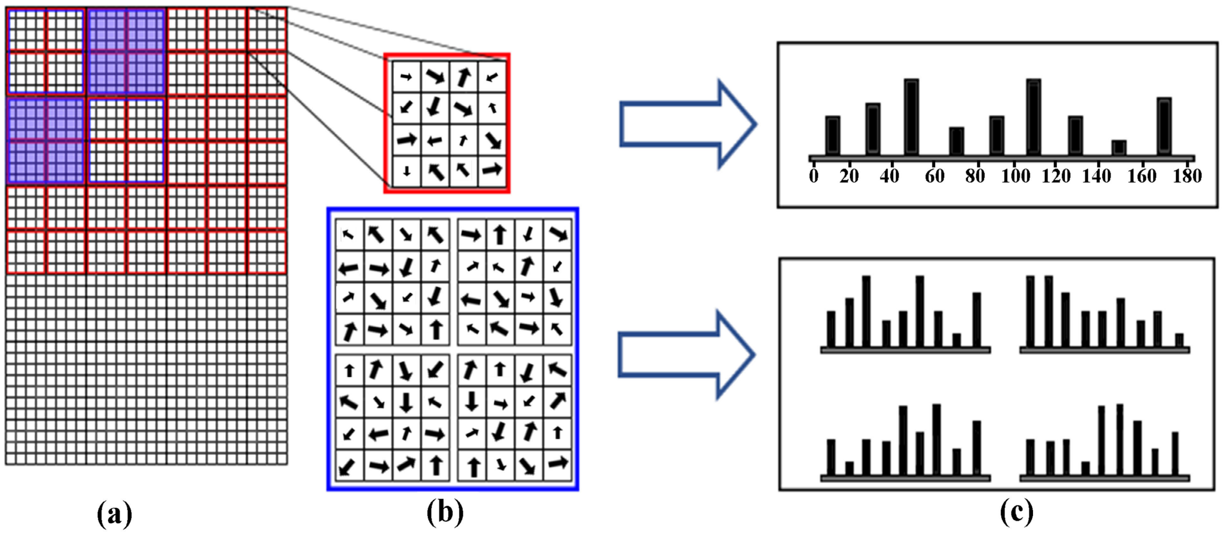

FIGURE 3. Visualization of cell and block in the HOG algorithm. (a) shows how pixels in an image window are grouped by cells and blocks. Each 4 by 4 pixels (red square) is a cell and each blue 8 by 8 pixels (blue square) is a block. The top part of (b) shows orientation of gradients in one cell and the bottom part of (b) shows the orientations in one block. The top part of (c) shows an example histogram for each bin created from one cell and the bottom part of (c) shows an example of 4 histograms in one block which are going to be concatenated and normalized in the next step. The number under each bin limit on the upper figure shows the ranges of the orientations of that bin.

divided for an example in which each cell contains 16 pixels, and each block contains four cells. The arrows in each pixel represent the orientations of the gradients in that cell, and a histogram is created for each cell.

The flowchart in Fig. 4 shows the steps of the HOG implementation for each cell of the image. After calculating the histograms for each cell, the cell histograms within a block are normalized together. Therefore, for this example, in each cell, 32 gradients, 16 magnitudes, and 16 orientation values are computed. Then, by comparing the orientations of the pixels with bin limits, the appropriate bin is assigned for all 16 pixels. After that, we have four histograms that are divided by the L2-norm of themselves.

In general, for an M by N input image, the first step of the HOG algorithm requires 9xMxN multiplications and 2xMxN additions to compute its gradient in x and y directions. In the second step, the algorithm computes the magnitude and arctangent for each pixel. Therefore, MxNx2 multiplications, MxN additions, and MxN square root and arctangent evaluations are required. In the third step, the angle for each pixel is compared with the limit values between 0 to 180 (or -90 to 90) degrees. Depending on the algorithm and data, the number of comparisons in this step might differ. But for nine bins, the maximum number of comparisons is nine. Then, for each pixel, the magnitude value is added to a bin value. Therefore, there are MxN additions in this step. After that, in the normalization step, the histograms of a block are normalized. For K cells in each block, we have 9K divisions, 9K multiplications and additions, and one square root computation. The total number of computations is shown in Table 3.

In Table 3, M and N are the dimensions of the input image, K is the number of cells in each block and B is the total

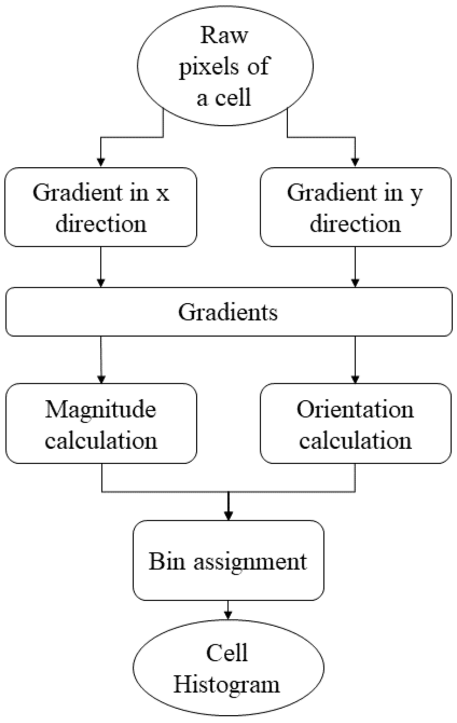

FIGURE 4. The flowchart of calculating a histogram of a cell from raw pixels. Parallel computation is possible for gradients in x and y directions and after that for magnitude and orientation calculation. The final result of this stage is a histogram of one cell.

number of blocks in the image. Table 3 is useful as it can show us by simplifying each stage of the algorithm, how many operations are affected. As an example, we can transform the multiplications in the gradient calculation step to subtractions

TABLE 3. Operations required for the HOG algorithm for an M × N image.

Stages of HOG	Multiplication	Addition	Square root	Arctangent	Comparison	Division
Gradient	9MN	2MN	----	----	----	----
Magnitude	2MN	MN	MN	----	----	----
Orientation	----	----	----	MN	----	MN
Bin assignment	----	----	----	----	9MN	----
Normalization	9KB	9KB	B	----	----	9KB
Overall	11MN+9KB	3MN+9KB	MN+B	MN	9MN	MN+9KB

Each line shows the required number of operations for each stage of the HOG algorithm. The last row presents the overall summation of required operations. The symbols represent: M = height of the image, N = width of the image, K = number of cells in each block, B = total number of blocks in an image.

and additions. Also, we have two steps containing square root. If we could optimize the calculation of square root, both these steps would benefit from it. For example, suppose that M = N = 200 and we have 16 pixels in each cell and 4 cells in each block. Therefore, K and B would be 2500 and 625, respectively. In this case, we have about 14M multiplications and additions, 40k square root calculations, 40k arctangent operations, 14M divisions, and 360k comparisons.

III. OPTIMIZING THE COMPUTATION OF THE ALGORITHM

The HOG algorithm contains several steps which are shown in the flowchart of Fig. 2. In this section, first, we categorize the different input selection choices in section A. The methods which optimize magnitude calculation are reviewed in section B. Then, since many work have integrated orientation calculation and bin assignment techniques, we review these two steps together in section C. Finally, the normalization step is discussed in section D.

A. INPUT SELECTION

In this section, we review different color channels and inputs to the HOG algorithm.

1) CHANNEL SELECTION

Several papers implement the HOG algorithm on the luminance channel. Rettkowski *et al.* [21] propose to first convert the RGB image into a luminance image for hardware implementation. Then, they compute the gradients of the luminance image using line buffers. Advani *et al.* [11] propose an initial stage to find the dominant channel (from RGB) for HOG calculation. In this method, first, the gradients of each cell of the image for the three RGB channels are calculated and based on the accumulation of their values, a comparator chooses which channel is the most suitable for further processing. Ilas [23] simplifies the input channel even further by proposing to compute HOG on binary images instead of grayscale images. First, the image is transformed from a greyscale image to a binary image by comparing each pixel to a threshold. Then, the HOG features are extracted.

Observations and Conclusions: Performing HOG on a binary image reduces the delay and hardware utilization since only one bit is used for each input pixel. However, since there is less information embedded in the extracted features, the accuracy of the model decreases as well. This method can be useful if the speed and hardware utilization are very critical in the specific application and the contour of the object of interest in the image does not contain many details. Otherwise, the luminance channel is the most commonly used in HOG feature extraction.

2) SINGLE SCALE COMPUTATION

While some researchers such as Blair *et al.* [4], Ma *et al.* [9], and Li *et al.* [36] implement the HOG algorithm on multi-scale images, Negi *et al.* [1] simplify the algorithm and compute HOG on a single scale image. Ma *et al.* [9] work on 640 × 480 pixel frames and use 1.05 scaling factor for multi-scale detection with a window stride of four pixels. They employ 34 scales for HOG extraction and achieve 68.18 frames per second with 640 × 480 frames.

Observations and Conclusions: Multi-scale detection, which means using different sizes of the image to extract features, requires extra hardware resources for resizing the input image and buffering it inside the FPGA. It should only be used if the accuracy is application dependent or there are varying sizes of the object of interest in the image; otherwise, single-scale detection is faster than multi-scale detection.

B. MAGNITUDE CALCULATION

The next step in the HOG algorithm is to compute the gradients of each pixel. In order to do so, the difference of pixels in rows and columns are calculated. The magnitude of gradients is computed as shown in (3) in section II. Since square and square root calculations are complex in hardware, several papers approximate the magnitude calculations. For FPGA implementation, in order to reduce the number of square root calculations in magnitude computations, Chen *et al.* [16] approximate the gradient amplitude for each pixel and simplify it according to (5) and (6):

$$\begin{aligned}\text{Magnitude(x,y)} &= |I(x+1,y) - I(x-1,y)| \\ &\quad + |I(x,y+1) - I(x,y-1)| \qquad (5)\\ \text{Magnitude(x,y)} &= |G(x)| + |G(y)| \qquad (6)\end{aligned}$$

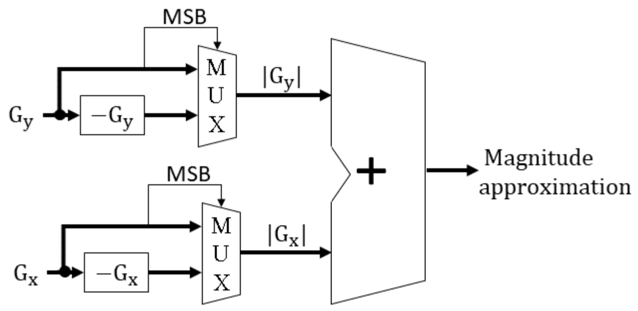

FIGURE 5. A basic hardware architecture for magnitude approximation by the summation of absolute values of gradients in horizontal and vertical directions. MSB is the most significant bit of the signal and the opposite sign is computed as two's complement. This architecture requires one adder, two multiplexers, and two two's complement operations.

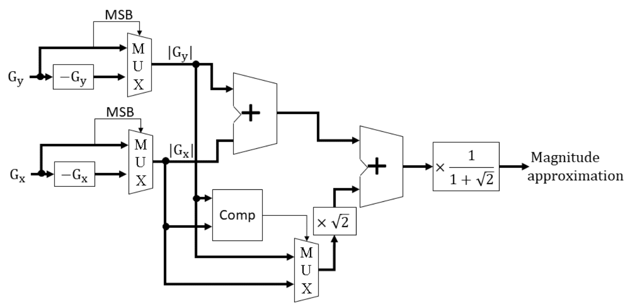

FIGURE 6. A basic hardware architecture for magnitude approximation. MSB is the most significant bit of the signal and the opposite is computed as two's complement. This architecture requires three multiplexers, two adders, two multipliers, one comparator (shown as Comp) and two two's complement operations.

Fig. 5 shows the block diagram of the hardware architecture described in (6).

Blair *et al.* [4] use the approximation mentioned by Qasaimeh *et al.* [31], as shown in (7):

$$\text{Magnitude} = \frac{1}{1+\sqrt{2}}\left(|G_x| + |G_y| + \sqrt{2}\max\left(|G_x|, |G_y|\right)\right) \tag{7}$$

The block diagram of the hardware architecture of (7) is shown in Fig. 6.

Chen *et al.* [8], B.K. *et al.* [22], and Wang and Zhang [26] use the square root approximation technique [37], which approximates magnitude calculation using (8):

$$\text{Magnitude(x,y)} = \max((0.875a + 0.5b), a) \tag{8}$$

where $a = \max(G_x, G_y)$ and $b = \min(G_x, G_y)$. In this way, the magnitude can be estimated using comparators and shifts only, thus simplifying the hardware. Fig. 7 presents the block diagram of the hardware design of (8).

Rettkowski *et al.* [21] store the magnitudes in look-up tables and compute the square root of magnitudes by retrieving the approximate values from the look-up tables. The total memory required for these look-up tables is 72 kbyte which can be stored in a block RAM in the FPGA.

Some researchers use pre-developed IP cores for magnitude calculation in the HOG algorithm. Sledeviè *et al.* [25],

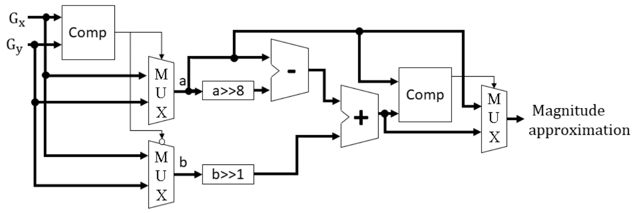

FIGURE 7. A basic hardware architecture for magnitude approximation. The "≫n" sign means signed shifting to the right by n bits. This architecture requires three multiplexers, one adder, one subtractor, two shifting operations, and two comparators (shown as Comp).

Luo and Lin [27], and Li *et al.* [36] use the Altera ALTSQRT IP core to calculate the square root function. Huang *et al.* [29] use a Xilinx IP core for square root calculation.

Observations and conclusions: there are several ways to improve speed and hardware utilization of the magnitude computation. The simplest method, which is used by Chen *et al.* [16], uses only two two's complement units, two multiplexers, and one adder. More complex method, such as the work proposed by Blair *et al.* [4], which requires two multiplications, two additions, three multiplexers and one comparison, results in a more accurate approximation of the magnitude computation. On the other hand, the square root approximation technique consumes fewer hardware resources as it uses only shift, add and comparison. Since the magnitude should be computed for every pixel, it has a great effect on the overall speed of the circuit. Therefore, methods proposed by Chen *et al.* [8], B.K. *et al.* [22], and Wang and Zhang [26] are suitable for high speed requirements. If accuracy is more important than speed, using IP cores for exact computation leads to a higher accuracy. If hardware utilization is not a concern in the design, the method provided by Rettkowski *et al.* [21] can be fast and accurate as well.

C. ORIENTATION AND BIN ASSIGNMENT

The next step of the HOG algorithm is computing the gradient orientation and assigning the magnitudes to the proper bins. Several papers make innovative contributions to this part of the algorithm.

1) BIN ASSIGNMENT

Many papers suggest that bin assignment can be performed without computing the value of arctangent for gradient orientation. Blair *et al.* [4] use the method described in the work provided by Bauer *et al.* [39] for assigning gradient magnitudes to bins. For orientation calculation, Blair *et al.* [4] use approximate values for tangent of the orientation ($\tan(\theta)$) and compare the $G_y(x,y)$ (gradient in the vertical direction) value with $\tan(\theta)$ multiplied by $G_x(x,y)$ (gradient in the horizontal direction). This method, which consumes less hardware resources than computing the tangent of the division of G_y over G_x for each value, is shown in (9).

$$G_x(x,y)\tan(\theta_i) \leq G_y(x,y) < G_x(x,y)\tan(\theta_{i+1}) \tag{9}$$

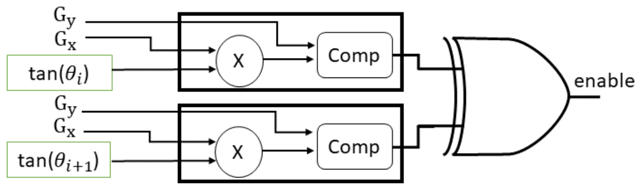

FIGURE 8. A basic hardware architecture for bin assignment. In this architecture, gradient in horizontal direction is multiplied by the tangent limit and is compared with the gradient in vertical direction. The two signals from the comparators (shown as Comp) of two adjacent blocks are XORed. If they are different, an enable signal is issued to write the value into the selected bin.

where θ_i is one of the limit angles and θ_{i+1} is the limit after that. Hahnle *et al.* [5], Hemmati *et al.* [7], Chen *et al.* [8], Zhou *et al.* [10], Wang and Zhang [26], and Luo and Lin [27] implement bin assignment in the same way. By using this method, they decrease the amount of hardware required for their HOG implementation.

For creating the histogram, Rettkowski *et al.* [21] use eight bins for degrees between $-\pi/2$ and $\pi/2$. They also scale the numbers so that instead of division, only integer multiplication and shifting are enough to find the correct bin for each histogram.

Ilas [19] provides slope computation as a replacement for arctangent calculation. Since the angle value is only used for bin assignment, the author suggests that slope value can do the same task with fewer computations. The slope can be computed as a division of gradient in y orientation by the gradient in x orientation. However, since slope changes more rapidly and can have large values, Ilas uses an adapting scaling method to saturate the value of slope in predetermined bins. Therefore, the HOG features computed are completely different than the conventional HOG. But the result shows that it has an accuracy near the original HOG in the case study of car detection systems.

Similar to magnitude computation, some work use IP-cores for orientation computation as well. Meus *et al.* [24] and Ngo *et al.* [28] use CORDIC IP cores for orientation computations.

Observations and Conclusions: The most commonly used method which many papers [4], [5], [7], [8], [10], [26], [27] use is to compute the tangent value of border angles and compare the y-direction gradient with the multiplication of the x-direction gradient and tangent limits. In this way, no trigonometric calculation is required. The method presented by Ilas [19] is similar in that there is no need for trigonometric calculation. However, it has less accuracy than the exact orientation computation in the HOG algorithm.

2) BIN NUMBER SELECTION

Some researchers propose methods to simplify the HOG algorithm. As an example, some papers use less than nine bins for histogram creation. For example, Rettkowski *et al.* [21] use eight bins, therefore each bin covers a domain of 180/8 degrees. Ilas [23] proposes to use only four bins instead of nine bins in the histogram computing stage, which correspond to 0, −45, +45, and 90 degrees. In this way, only 2.6% of LUTs are used. However, the accuracy is decreased. The author suggests that one can use this algorithm to find candidates and perform the original HOG only on selected candidates to improve the detection time. Chen *et al.* [16] propose to divide the linear gradients into only six orientation bins in 0° to 180° to reduce the computation complexity, with negligible accuracy loss compared to nine bins. Ilas [18] divides the orientation bins for every 16 degrees instead of 20 degrees. Therefore, this implementation consumes fewer hardware resources, and the division by 20 can be replaced by division of 16 which is implemented conveniently as a 4-bit shifter. As a result, 6% fewer LUTs and 17% fewer registers are used.

Observations and Conclusions: Using a smaller number of bins can reduce hardware utilization; however, it decreases the accuracy in some applications. Therefore, this is suitable for applications where hardware resources are limited but accuracy reduction is tolerable. Furthermore, the method proposed by Ilas [23] which uses a simpler classifier (with four bin HOG) first and a more precise HOG after that, can perform faster by using more hardware resources.

3) INTERPOLATION

Another area of algorithm simplification is interpolation between bins. Although some authors such as Chen *et al.* [8] implement bilinear interpolation, Ilas [18] assumes that if no interpolation is considered between bins in both training and testing phases, the accuracy will not change. On the other hand, Chen *et al.* [8] compute the weight of gradient magnitude for two adjacent bins using (10):

$$\alpha = (\mathrm{n}+0.5) - \mathrm{b}\frac{\theta(\mathrm{x,y})}{\pi} \tag{10}$$

in which, b (the total number of bins) is 9 and n is the bin to which θ belongs. After that, the magnitudes are weighted as (11) and (12) for two adjacent bins:

$$\mathrm{m_n} = (1-\alpha)\mathrm{m(x,y)} \tag{11}$$

$$\mathrm{m_{nearest}} = \alpha \mathrm{m(x,y)} \tag{12}$$

Observations and Conclusions: Performing interpolation in the bin assignment makes histograms smoother. However, Ilas [18] shows that if interpolation is not used in both training and testing phases of feature extraction, the overall accuracy reduction is negligible. Therefore, interpolation is useful in cases which accuracy is critical, and the hardware utilization and delay caused by this unit are acceptable.

D. NORMALIZATION

The next step in the HOG algorithm is the normalization of histograms in each block. After assigning the magnitudes to different bins, the histograms of cells inside a block are normalized. Several papers propose techniques to make this normalization computation efficient.

Mizuno *et al.* [2], Chen *et al.* [8], Ma *et al.* [9], and B.K. *et al.* [22] use the Newton method to approximate L_2 normalization. Chen *et al.* [8] and B.K. *et al.* [22] use IEEE754 standard floating-point representation to normalize the histograms in each block. The authors also use the Newton-Raphson method to approximate inverse square root.

The formula for Newton approximation is shown in (13):

$$\frac{1}{\sqrt{x}} = \mathrm{y_d}\frac{\left(3 - x\left(\mathrm{y_d}\right)^2\right)}{2} \tag{13}$$

where

$$\mathrm{y_d} = \mathrm{Decimal}\{(\mathrm{x_{IEEE754}} \gg 1) - \mathrm{0x5F3759DF}\} \tag{14}$$

where $\mathrm{x_{IEEE754}}$ is the IEEE754 floating-point representation of x. Decimal {h} is the decimal representation of the hexadecimal value of h. Also, 0x5F3759DF is the magic number [39] for Newton-Raphson approximation so that there is no need for iteration.

Wang and Zhang [26] use shifting instead of division for block normalization according (15).

$$\mathrm{h_n} = \mathrm{h} \gg [\log_2(\mathrm{sum(h)})] \qquad \text{for } \mathrm{h} \neq 0 \tag{15}$$

where sum(h) is the summation of values in vector h and $\mathrm{h_n}$ is the normalized vector.

Observations and Conclusions: Although the method provided by Wang and Zhang [26] consumes less hardware resources since it uses only shifting and addition, it is less accurate. The Newton approximation method which is used in some work [2], [8], [9], [17], [22] for inverse square root is more often used for normalization since it consumes fewer hardware resources than computing the more exact value of inverse square root using IP cores.

IV. DATA MANIPULATION TECHNIQUES

In this section, we review the methods which change the parameters that affect the whole algorithm. In section A, bit-width and numerical representation and their effects on FPGA implementation are discussed. Then, in section B, the methods that contribute to data path optimization of the HOG algorithm are surveyed. Finally, in section C, the methods which are more focused on memory usage optimization are presented.

A. NUMERICAL REPRESENTATION

One of the main techniques for efficient implementation of mathematical algorithms on an FPGA is choosing an appropriate numerical representation. If more than the necessary number of bits are used in the implementation, without any changes in accuracy, more memory is used, and the total latency of the circuit is increased. On the other hand, if the bit-width is too small or the representation is too simple, it will result in accuracy loss.

While some researchers such as Komorkiewicz *et al.* [3], Chen *et al.* [8], and B.K. *et al.* [22] use floating-point representation similar to that of the original HOG algorithm [40], others such as Ma *et al.* [6], Hemmati *et al.* [7], and Ngo *et al.* [28] use a constant number of bits for fixed-point representation for HOG. Ma *et al.* [6] investigate using fixed-point calculations for HOG feature extraction. The authors suggest that using 13-bit fixed-point can preserve the accuracy of an HOG-SVM pedestrian detector and even enhance it. Although reducing bit-width decreases area consumption, it may not preserve accuracy for some applications.

Some research groups use different fixed-point representations for different parts of the HOG algorithm [1], [2], [9]. Negi *et al.* [1] use 19 bits for gradient calculations, 14 bits for each histogram, and 33 bits for normalized histograms. Although they achieve 62.5 frames per second, due to the fixed-point implementation, the accuracy of their model decreases. Mizuno *et al.* [2] dedicate 9 bits for gradient magnitude, 6 bits for gradient orientation, 11 bits for orientation histogram, and 25 bits for L_2 normalization which they use Newton method to approximate it. In order to preserve the accuracy, first, Chen *et al.* [9] use 27-bit fixed points for HOG calculations. Then, for each part of the calculations, they decrease the bit-width and compare the results with software implementation to determine if the accuracy has decreased or not.

Observations and Conclusions: Overall, data representation is a trade-off between accuracy, speed, power, and resource consumption. The methods which use floating-point numbers [3], [8], [22] obviously lead to higher accuracy. However, floating-point computation requires more hardware resources and is not as fast as fixed-point computations. Therefore, although the ones that use fixed-point calculations [6], [7], [28] are less accurate, they perform faster and use fewer FPGA resources than the others. On the other hand, methods such as those proposed by Negi *et al.* [1], Mizuno *et al.* [2], and Ma *et al.* [9] are more balanced since they try to optimize the hardware usage and also preserve the accuracy of the model.

B. DATA FLOW MODIFICATION

In this section, we review the methods which optimize the data path of the HOG algorithm. Mizuno *et al.* [2] present an architectural study on HOG feature extraction. The authors propose a cell-based pipeline for HOG computations which reduces memory bandwidth. They use an external CPU to control the pipeline for HOG computation. They process images with 1920 × 1680 pixels and achieve 30 frames per second with 76 MHz clock frequency. Komorkiewicz *et al.* [3] propose using a 32-bit single-precision floating-point numbers in HOG computation for object detection. They use a complete pipeline without the need for memory in intermediate calculations and achieve 60 frames per second for 640 × 480 frames. Their intent is to improve accuracy by using more hardware resources.

Luo and Lin [27] implement the HOG algorithm fully on an FPGA. They test their design on 800 × 600 images with 150 MHz clock frequency. Three line-buffers of 800

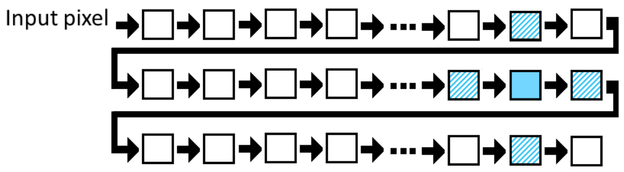

FIGURE 9. A pipeline of three rows of line-buffer registers [16], [27]. The pixels enter the registers from the top left register and flow into the buffer. At each clock cycle, the solid blue register is the main pixel and the values of dashed pixels on top, bottom, left, and right of the main pixel, which have fixed positions, are used to compute the gradients.

8-bit words are used to accommodate the pixels required for calculating the gradients simultaneously. After assigning magnitudes to the bins, the cell values are computed using several shift registers on the fly, and the result is given to a block normalization module. The authors use FIFO memories to save the blocks which overlap with other blocks, and use them later without redundant computations.

Chen *et al.* [16] propose to use a shift register which covers three rows of pixels since HOG feature extraction for each pixel requires four other surrounding pixels. By using buffers for every three lines, after an initial setup time and when the buffers are full, at each clock cycle, the required pixels are available. Therefore, at each clock cycle, the values of the required pixels are extracted from the shift registers, and the computations are done. Fig. 9 shows the architecture of line buffers in that paper. Finally, when the orientation of each pixel is computed in the HOG engine, the L_1 distance of the HOG features and the HOG of the object model are computed to classify the object. The line buffers enable the algorithm to run faster but on-chip registers are used for those buffers. Using a fewer number of bins allows the algorithm to run faster however it also decreases its accuracy.

Qasaimeh *et al.* [31] propose a systolic array structure for HOG computation. For sliding window operation, they propose to compute the histograms for each 3×3 window in an overlapping manner. When the histogram of one window is computed, for the next window, the contribution of the last column is subtracted from the histogram, and the contribution of the new column is added. Therefore, at each window, one column calculation is preserved. Fig. 10 shows how this method affects the computed histogram.

Observations and Conclusions: Data path optimization techniques can affect the overall speed of the implementation. Designing the circuit using pipeline architecture can reduce the clock period and increase maximum clock frequency. On the other hand, using line buffers is a common solution for parallelizing the input data which is read at the beginning of the algorithm. These techniques lead to performance enhancement in speed.

C. MEMORY OPTIMIZATION

Hemmati *et al.* [7], Ma *et al.* [9], Chen *et al.* [16], and Luo *et al.* [27] propose innovative methods for memory usage. One of the innovations of the work provided by

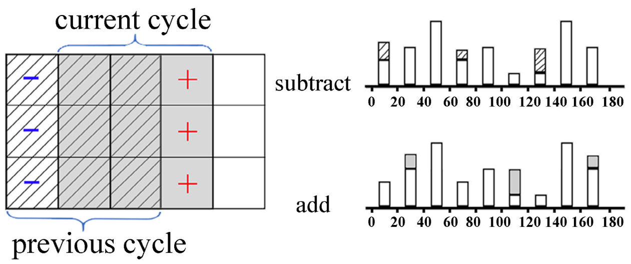

FIGURE 10. Sliding window proposed by Qasaimeh *et al.* [31]. The left image shows two consecutive cycles. In the current cycle, the contribution of pixels from left side of the previous window (blue -) are subtracted from the histogram and the contribution of new pixels (red +) are added. The top right image shows that some values are subtracted from bin values and in the bottom right image new values are added. The number under each bin limit shows the ranges of the orientations of that bin.

Hemmati *et al.* [7] is the use of four different memories, each containing one cell of each block. Therefore, gradient calculation, histogram generation, and block normalization can be done in parallel by accessing the cell values from four independent memories.

Ma *et al.* [9] design the HOG hardware to fetch pixels from two rows of cells and to process two rows at the same time to have more parallelism. Since cells in one row can be used in the next row for block normalization, alternating between odd and even rows can prevent computing the histograms two times and therefore lead to a speed-up in their implementation. In addition, the authors pack each pair of magnitude and orientation into a single 32-bit integer. In each memory access, a 64-bit value is returned which is two pairs of magnitude and orientation. Still, in their implementation, they compute the magnitude and orientation in software and send them to the FPGA for further computations and block normalization. They attempt to maintain the accuracy of the HOG algorithm by not reducing the number of bits, the number of bins, and scaling factor for classification. They propose a method to make the whole algorithm (without simplification) faster. However, they use software for more complex computational parts such as magnitude and orientation computation. The main goal of their work is to preserve the accuracy and then improve the speed of their design.

Observations and Conclusions: Memory usage optimization has an important role in data throughput. However, storing the data as required in the memory might not be achievable in all situations, since it is dependent on the application and platform.

V. MODIFIED HOG-BASED FEATURES

In order to increase the accuracy of the overall algorithm, some papers have modified the HOG features. In this section, we review four methods which implement modified HOG algorithm on FPGAs.

A. FEATURE COMPRESSION

Advani *et al.* [11] propose feature compression. From the 72 features that are produced after block normalization in their paper, a concatenation of 18 features (accumulated bins for each cell), 9 features (coupling 18 bins into nine bins) and 4 features (accumulating bins of each cell) are generated. Overall every 72 features are compressed into 31 features. Therefore, the final classifiers (SVM in this case) only have to process 31 features.

Rettkowski *et al.* [21] propose to transform the final HOG features to binary values. They convert the features to binary values by comparing each value of the histograms with 8/128 (which is a 4-bit shift) and transform every 14-bit value in the histograms into a 1-bit feature, which is very beneficial in memory usage reduction.

B. CORRELATION OF HOG FEATURES

Nishizumi *et al.* [17] introduce sparse FIND (Feature Interaction Descriptor) features which are extracted from HOG features. They first compute the HOG features and then calculate FIND features by obtaining the correlation between HOG features and normalizing the correlations. Because of the complexity of the calculations, they only extract the elements with high validity in identification from HOG (which have values more than a threshold). They calculate this threshold using (16) where k is a sparsification parameter, m is the number of histogram bins in the block, and h_i is the HOG value in each bin. Equation (17) also shows the parameter α which is used for normalization.

$$\mathrm{threshold} = (k/m)\sum h_i \tag{16}$$

$$\alpha = 1/\sum h_i^2 \tag{17}$$

Finally, they compute the correlation as shown in (18).

$$f(h_i, h_j) = \alpha \times h_i \times h_j \quad \text{if} \quad h_i, h_j > \mathrm{threshold} \tag{18}$$

where h_i and h_j are values of the bins in the histogram that are bigger than the threshold. They use the correlation to reduce the number of dimensions. Using these correlation values as new features reduces the number of previous features, and the classification step becomes simpler. However, for computing sparse FIND features, multiplication and division are required which consume more hardware resources. Their method improves the accuracy but consumes more area in the feature extraction part.

C. HISTOGRAM OF SIGNIFICANT GRADIENTS

Bilal *et al.* [20] introduce HSG (Histogram of Significant Gradients) which is a modified version of HOG. In this method, they compute the average magnitude in each cell, and if a gradient magnitude is higher than the average value, that magnitude casts a binary vote to the histogram bin (the value of that bin is incremented by one unit). One advantage of this method is that there is no need to normalize HOG features and another advantage is that the final feature vector contains integer values only, hence, the hardware computation is simplified.

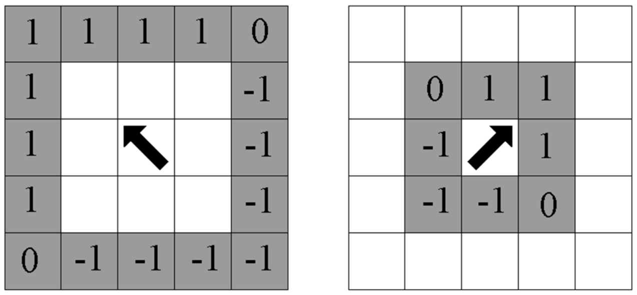

FIGURE 11. The extension of HOG features used by Sledeviè *et al.* [25]. The left image shows the example orientation in the main pixel calculated by 16 pixels with 1-pixel distance from the main pixel. The right image shows the example orientation of the pixels around the main pixel. The histograms derived by both methods are concatenated together as final features.

D. GRADIENT EXTENSION

Sledeviè *et al.* [25] increase the number of HOG features by using pixels which are one pixel further from the center pixel. Their implementation uses both the 8 pixels around the main pixel and the 16 pixels around the first 8 pixels to compute the gradient, and this method processes 24 pixels for each pixel. Then for each block, they add the value to the bin dedicated to the specific direction. Fig. 11 shows the histogram bins for one pixel using the method provided by Sledeviè *et al.* [25]. Higher accuracy is expected by using more features to get more information about the local variations of the image. They use the HOG algorithm for tracking objects.

Observations and Conclusions: In this section, we reviewed four variants of the HOG algorithm and their hardware implementation. The methods described in the feature compression subsection can be used where FPGA resources are limited or the features are to be sent through a communication channel (such as IoT devices). Sparse FIND features require more computations for feature extraction. Although the authors decrease the number of features, more hardware recourses are utilized to compute them. HSG features simplify the histogram computation by creating binary features which is a trade-off for accuracy. In contrast, extension of gradients for 16 further pixels uses more hardware to have a better accuracy for their application.

VI. HARDWARE-SOFTWARE IMPLEMENTATIONS

While some work focus on pure FPGA implementation of the HOG algorithm, others propose a hardware-software co-design approach. Mizuno *et al.* [2] use CPU to control the logic of a pipeline for HOG computations. For intermediate computations such as storing histograms of processed cells and the SVM coefficients, they use an external SRAM memory. Ma *et al.* [9] implement the HOG-SVM system on a Convey HC-2ex platform, which is composed of two Intel Xeon four-core processors and four Xilinx Virtex-6 FPGAs that can communicate with each other through shared memory. They compute the magnitude and orientation in software and store them in memory. Then, the FPGA cores read the data from the memory for further computations and block normalization.

Rettkowski *et al.* [21] implement a software version, a HW/SW codesign version, and a hardware version of the HOG algorithm. In their paper, they show that hardware design is 503x faster than the software version and the HW/SW version is 9x faster than the software version. For the HW/SW version, they use the Xilinx tool SDSoc to convert C code to hardware implementation. They compute the gradient, magnitude and orientation in software, and histogram generation and block normalization in hardware. The main contribution by Huang *et al.* [29] is in the classification part which they classify all blocks with the Adaboost classifier, and give only the most probable candidate windows (which have 36x(number of blocks) features) to a linear SVM for human detection applications. The authors implement HOG as a hardware accelerator to be used in a HW/SW system. They send one block of pixels at each time to the FPGA to get the normalized histograms of that block. Then, they perform the classification using an ARM processor. Ngo *et al.* [28] implement the HOG algorithm including the sliding window step in hardware and the classification step in software. Most of the work in our survey use Hardware Description Languages such as Verilog or VHDL. However, B.K. *et al.* [22] and Meus *et al.* [24] use high level synthesizers for FPGA implementation.

Observations and Conclusions: Deciding whether or not to use HW/SW implementation depends on the application and the other parts of the design. As mentioned by Rettkowski *et al.* [21], pure hardware implementation is usually faster than HW/SW version. However, if programmable logic is limited in a circuit, software solutions can be useful. In applications and scenarios where FPGA resource usage is more critical than speed, the HW/SW design approach performs better. However, it is important to separate the algorithm into the appropriate parts for each platform to get the best results.

VII. DISCUSSION

In this section, first, we compare the performance of different work with respect to frame rate and speed. Then, in the second part, we suggest design guidelines for optimized implementation of the algorithm while considering the limitation of the system.

A. SPEED COMPARISON

In Table 4 and Table 5, we summarize the results for the surveyed papers for HOG implementation. One of the most commonly used metrics for speed evaluation of an image processing algorithm is the number of frames processed in one second. However, since FPGA implementation of an algorithm depends on the clock frequency and the size of the input image, frames per second may not be a fair evaluation metric for comparing different algorithms. The reason for this argument is that in the same design, frames per second can easily be increased by decreasing the size of the input frame or by increasing the clock frequency. Therefore, we use the method presented by Ngo *et al.* [28] which computes the pixels per clock cycle, according to (19) and (20).

$$\text{pixels per clock cycle} = \mathrm{W} \times \mathrm{H} \times \mathrm{FR} \times 1/f \tag{19}$$

$$\frac{\text{pixels}}{\text{clock cycle}} = \frac{\text{\#of pixels}}{\text{frame}} \times \frac{\text{frame}}{\text{s}} \times \frac{\text{s}}{\text{clock cycles}} \tag{20}$$

where W is the width of the frame, H is the height of the frame, FR is the frame rate, and f is the frequency. In order to compute this metric, first, the frames per second rate is multiplied by the input image size and the result is the number of pixels processed in each second. Then, the obtained value is divided by the clock frequency of the system so that all work can be compared in the same clock domain. Table 4 shows this measure for different implementations and hardware platforms. We compute this metric only for papers that reported the frame size, the clock frequency of the FPGA, and frames per second rate.

As shown in Table 4, some work use Zynq family FPGAs [7], [10], [16], [21], [24], [15] while others use Virtex series FPGAs [1], [3]–[5], [9], [11], [17], [25], [35]. On the other hand, [2], [20], [26]–[28] use Cyclone family FPGAs. Cyclone V devices have more available memory while Virtex family FPGAs have more logic elements than Cyclone and Zynq series. It can be seen that methods which are implemented on Zynq series are mostly in the upper half of Table 4 indicating their speed superiority. In addition, the results by Ngo *et al.* [28] who use Cyclone V FPGA is better than all Cyclone IV FPGAs. Although it seems that newer FPGAs and technology lead to faster systems, it cannot be concluded

TABLE 4. Speed comparison and FPGA used in each reference.

Reference	FPGA Platform	Frame rate (fps)	Pixels per clock cycle
Long *et al.* [33]	Altera Stratix IV	10000	8.192
Nishizumi *et al.* [17]	Virtex Ultrascale	46	1.063
B.K. *et al.* [21]	Zynq	39	1.001
Zhou *et al.* [10]	Zynq	236000	0.999
Ngo *et al.* [28]	Cyclone V	526	0.997
Hemmati *et al.* [7]	Zynq	60	0.995
Zhou *et al.* [10]	Zynq	116	0.995
Meus *et al.* [24]	Zynq	60	0.747
Komorkiewica *et al.* [3]	Virtex-6	60	0.737
Sledevie *et al.* [25]	Virtex-4	60	0.737
Advani *et al.* [11]	Virtex-7	30	0.622
Luo *et al.* [27]	Cyclone IV	162	0.518
Hahnle *et al.* [5]	Virtex-5	64	0.498
Mizuno *et al.* [2]	Cyclone IV	72	0.454
Bilal *et al.* [20]	Cyclone IV	162	0.422
Chen *et al.* [16]	Zynq	41	0.251
Xu *et al.* [12]	Spartan-6	47	0.225
Wang *et al.* [26]	Cyclone IV	60	0.170
Ma *et al.* [9]	Virtex-6	68	0.139
Negi *et al.* [1]	Virtex-5	62	0.108
Blair *et al.* [4]	Virtex-6	13	0.051
Ahmad *et al.* [35]	Virtex-6	6	0.002
Hsiao *et al.* [14]	Spartan-6	15	0.001

Pixels processed per clock cycle is shown in this Table. The FPGA platform is also shown for comparison.

TABLE 5. Resource utilization.

Reference	Implementation	FPGA	LUT	BRAM (Kbit)	DSP
Rettkowski *et al.* [21]	Only HOG core	Zynq	18987	0	4
Bilal *et al.* [20]	Only HOG core	Cyclone IV	751	43.7	0
Long *et al.* [33]	Full system	Stratix IV	266023	47	236
Bilal *et al.* [20]	Full system	Cyclone IV	65501	103	10
Xu *et al.* [12]	Only HOG core	Spartan-6	9955	208	66
Ngo *et al.* [28]	Only HOG core	Cyclone V	8610	326	74
Mizuno *et al.* [2]	Only HOG core	Cyclone IV	34403	334	68
Yu *et al.* [13]	Full system (SoC)	Spartan-6	15167	351	19
Hemmati *et al.* [7]	Only HOG core	Zynq	7649	432	26
Ngo *et al.* [28]	Full system (SoC)	Cyclone V	12138	437	65
Hahnle *et al.* [5]	Only HOG core	Virtex-5	5188	1188	49
Negi *et al.* [1]	Full system	Virtex-5	17383	1327	N/A
Chen *et al.* [16]	Full system	Zynq	10052	1620	2
Ranawaka *et al.* [34]	Only HOG core	Zynq	6069	2682	117
Adiono *et al.* [32]	Only HOG core	Cyclone IV	83000	2800	90
Wang *et al.* [26]	Only HOG core	Cyclone IV	94374	3042	N/A
Blair *et al.* [4]	Full system	Virtex-6	108518	3744	138
Blair *et al.* [15]	Full system	Virtex-6	77623	3906	108
Komorkiewicz *et al.* [3]	Full system	Virtex-6	113359	4284	72
Raj *et al.* [30]	Full system	Virtex-5	69120	4680	N/A
Li *et al.* [36]	Full system	Stratix IV	313349	9696	268
Advani *et al.* [11]	Full system	Virtex-7	137361	13500	202
Ma *et al.* [9]	Full system	Virtex-6	184953	13737	190

This table is sorted in an increasing order of BRAM (Block RAMs) usage. The second column shows that the hardware resources reported are for a full system or only the HOG core. Full systems usually contain a classifier such as SVM or Adaboost. The ones with SoC have used a processor besides the HOG core.

TABLE 6. Optimization for speed.

Parameters	Original Form	Technique or approximation for hardware implementation	Reference
Magnitude	$\text{Magnitude}(x,y)=\sqrt{G_x^2(x,y)+G_y^2(x,y)}$	$\text{Magnitude}(x,y)=\lvert G_x(x,y)\rvert+\lvert G_y(x,y)\rvert$	[16]
Orientation	$\theta_i \leq \arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right) < \theta_{i+1}$	$G_x(x,y)\tan(\theta_i) \leq G_y(x,y) < G_x(x,y)\tan(\theta_{i+1})$	[4]
Normalization	$h_n = \frac{h}{\sqrt{\lVert h\rVert^2 + e}}$	$h_n = h >> [\log_2(\text{sum}(h))]$ for $h \neq 0$	[26]
Bit-width	Floating-point number	13-bit fixed-point	[6]
Data path	----	Four different memories each for one cell in a block	[7]
		Line buffers for lines of an image	[9][16][27]
		Pipeline stages of HOG	[2][3]
Simplifications	----	Single-scale	[1]
		No interpolation	[18]

The symbols in the table are: h_n = normalized histogram of a block, h = histogram of a block before normalization, G_x = gradient in horizontal direction, G_y = gradient in vertical direction, θ_i = angle limits (normally from 0 to 180 and in 9 steps).

as a fact since the works with newer technology are the most recent ones and they have more effective innovations in their implementation as well.

Nishizumi *et al.* [17] use the CORDIC method for arctangent and square root operations for histogram generation. They also use the Newton method for square root division for normalization. While most of the work read the input data as one pixel per clock cycle, Long *et al.* [33] use a high-speed camera and receive the data by 64 pixels per clock cycle. Therefore, the overall speed of their system is higher than the others.

The work proposed by Ngo *et al.* [28] has a frame rate of 526 which is higher than other projects with similar input stream size and throughput. The reason is that they use buffers for data path optimization and CORDIC IP cores for magnitude and orientation computation. As a trade-off, they use more registers than others. The next best performance with respect to frame rate are found in [27], [20], and [10]. Zhou *et al.* [10] use a pipeline architecture for HOG implementation and simplification on normalization formula. They also report the frame rate based on 32 × 32 frames which are much smaller than what others use and their frequency is

TABLE 7. **Optimization for accuracy.**

Parameters	Original Form	Technique	Reference
Magnitude	$\mathrm{Magnitude}(x,y)=\sqrt{G_x^2(x,y)+G_y^2(x,y)}$	$\mathrm{Magnitude}(x,y)=\sqrt{G_x^2(x,y)+G_y^2(x,y)}$	[40]
Orientation	$\theta_i \le \arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right) < \theta_{i+1}$	$G_x(x,y)\tan(\theta_i) \le G_y(x,y) < G_x(x,y)\tan(\theta_{i+1})$	[4]
Normalization	$h_n=\frac{h}{\sqrt{\lVert h\rVert^2+e}}$	$h_n=\frac{h}{\sqrt{\lVert h\rVert^2+e}}$	[40]
Bit-width	Floating-point number	Floating-point number	[3][8][22]
Simplifications	----	Multi-Scale Use interpolation in bin assignment	[4][9][36] [8]

The symbols in the table are: h_n = normalized histogram of a block, h = histogram of a block before normalization, G_x = gradient in horizontal direction, G_y = gradient in vertical direction, θ_i = angle limits (normally from 0 to 180 and in 9 steps).

TABLE 8. **Optimization for hardware resource utilization.**

Parameters	Original Form	Technique	Reference
Magnitude	$\mathrm{Magnitude}(x,y)=\sqrt{G_x^2(x,y)+G_y^2(x,y)}$	$\mathrm{Magnitude}(x,y)=\lvert G_x(x,y)\rvert+\lvert G_y(x,y)\rvert$	[16]
Orientation	$\theta_i \le \arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right) < \theta_{i+1}$	$G_x(x,y)\tan(\theta_i) \le G_y(x,y) < G_x(x,y)\tan(\theta_{i+1})$	[4]
Normalization	$h_n=\frac{h}{\sqrt{\lVert h\rVert^2+e}}$	$h_n = h >> [\log_2(\mathrm{sum}(h))]$ for $h \neq 0$	[26]
Bit-width	Floating-point number	13-bit fixed-point	[6]
Simplifications	----	Single-scale No interpolation A smaller number of bins Binary image input	[1] [18] [16] [23]

The symbols in the table are: h_n = normalized histogram of a block, h = histogram of a block before normalization, G_x = gradient in horizontal direction, G_y = gradient in vertical direction, θ_i = angle limits (normally from 0 to 180 and in 9 steps).

241 MHz which is higher than other work. Luo and Lin [27] use Altera IP cores for square root and line buffers for data path structure. Bilal *et al.* [20] remove the normalization step by using a threshold value in the bin assignment stage. In conclusion, data path optimization techniques, such as using pipeline architecture and line buffers, have a great effect on the final frame rate. Besides, the normalization step is one of the most time-consuming steps of HOG algorithm implementation.

B. RESOURCE UTILIZATION

The resource utilization of different approaches for FPGA-based implementation of HOG is shown in Table 5. These resources, which are normally used to compare designs, include look-up tables (LUT), block RAMs (BRAM) and digital signal processing units (DSP) on the FPGA. LUTs are the smallest programmable elements on the FPGA. Block RAMs are memories which are embedded on the FPGA. DSPs on the FPGA are used for mathematical operations such as multiplications. Since many papers do not report their hardware utilization explicitly, we include only the ones which report their results in Table 5. The work proposed by Rettkowski *et al.* [21] is on top of the table. This work does not use block memories but uses more LUTs instead. The method proposed by Bilal *et al.* [20] does not require the normalization step which has led to less resource usage. Considering the work which use most hardware resources, the method proposed by Ma *et al.* [9] has the most LUT and BRAM usage in Table 5 since they implement the HOG algorithm for 34 scales. Advani *et al.* [11] implement multi-scale detection while Komorkiewicz *et al.* [3] implement HOG using floating-point numbers. Li *et al.* [36] implement a highly parallel system which receives the input of 64 pixels per clock cycle. They also implement multi-scale detection. These are the main reasons for the large number of used resources in these papers.

C. DESIGN GUIDELINES

The most important criteria in embedded real-time systems are speed, accuracy, resource usage, and power. In this part, we summarize the techniques reviewed in this survey as a design guideline based on these criteria. Since FPGA-based designs are typically of lower power than CPUs and GPUs, in this section we focus on speed, accuracy and resource utilization of the designs. Table 6 shows the methods suggested for increasing the speed of the final design by using different techniques in existing work. Methods described in this table require a fewer number of computations and use more hardware resources. However, they are optimized for speed.

The methods described in Table 7 are optimized for accuracy. No simplification is used for the methods in Table 7 in order to have the same features as the software implementation. The methods described in Table 8 are optimized for the cases which hardware resources are limited.

D. CONCLUSION

In this paper, we reviewed the methods used to implement the histogram of oriented gradients algorithm on FPGAs in the past decade (2010-2019). Some of the reported techniques are related to individual steps of the algorithm, and some affect the whole algorithm. We also reviewed different simplification methods and the hardware implementation of modified features which were based on the original HOG algorithm. After that, we compared the recent work regarding speed, accuracy, and resource utilization of the designs. It was observed that the methods which focus on optimizing the data path of the design have the most effect on the speed of the circuit. Finally, we presented three design guidelines for FPGA-based HOG implementation which categorize different methods regarding the limitation and requirements of different applications.

The research community has considered many aspects of the hardware implementation of the HOG algorithm. However, many have not reported the accuracy of their proposed methods, possibly due to the fact that their main focus lies in other aspects such as processing speed and resource utilization, or accuracy is not a concern in their particular application. For the ones that reported accuracy, it is often less than that of a software implementation. Interpolation in bin assignment and normalization steps are the parts that most of the work approximated and simplified to gain higher speed, hence, leading to accuracy issues. Though, these two steps have great potential to enhance FPGA-based HOG implementation in a future work. Since the proposed methods for multi-scale detection are not the focus with respect to hardware resource consumption, one of our future directions is to find an optimum number of image scales to achieve higher accuracy while maintaining other design metrics.

REFERENCES

[1] K. Negi, K. Dohi, Y. Shibata, and K. Oguri, "Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm," in *Proc. Int. Conf. Field-Program. Technol.*, Dec. 2011, pp. 1–8.

[2] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "Architectural study of HOG feature extraction processor for real-time object detection," in *Proc. IEEE Workshop Signal Process. Syst.*, Oct. 2012, pp. 197–202.

[3] M. Komorkiewicz, M. Kluczewski, and M. Gorgon, "Floating point HOG implementation for real-time multiple object detection," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2012, pp. 711–714.

[4] C. Blair, N. M. Robertson, and D. Hume, "Characterizing a heterogeneous system for person detection in video using histograms of oriented gradients: Power versus speed versus accuracy," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 3, no. 2, pp. 236–247, Jun. 2013.

[5] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "FPGA-based real-time pedestrian detection on high-resolution images," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2013, pp. 629–635.

[6] X. Ma, W. Najjar, and A. R. Chowdhury, "High-throughput fixed-point object detection on FPGAs," in *Proc. IEEE 22nd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, Boston, MA, USA, May 2014, p. 107.

[7] M. Hemmati, M. Biglari-Abhari, S. Berber, and S. Niar, "HOG feature extractor hardware accelerator for real-time pedestrian detection," in *Proc. 17th Euromicro Conf. Digit. Syst. Design*, Aug. 2014, pp. 543–550.

[8] P.-Y. Chen, C.-C. Huang, C.-Y. Lien, and Y.-H. Tsai, "An efficient hardware implementation of HOG feature extraction for human detection," *IEEE Trans. Intell. Transp. Syst.*, vol. 15, no. 2, pp. 656–662, Apr. 2014.

[9] X. Ma, W. A. Najjar, and A. K. Roy-Chowdhury, "Evaluation and acceleration of high-throughput fixed-point object detection on FPGAs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 6, pp. 1051–1062, Jun. 2015.

[10] Y. Zhou, Z. Chen, and X. Huang, "A pipeline architecture for traffic sign classification on an FPGA," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 950–953.

[11] S. Advani, Y. Tanabe, K. Irick, J. Sampson, and V. Narayanan, "A scalable architecture for multi-class visual object detection," in *Proc. 25th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2015, pp. 1–8.

[12] X. Yuan, L. Cai-nian, X. Xiao-liang, J. Mei, and Z. Jian-guo, "A two-stage hog feature extraction processor embedded with SVM for pedestrian detection," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2015, pp. 3452–3455.

[13] Z. Yu, S. Yang, I. Sillitoe, and K. Buckley, "Towards a scalable hardware/software co-design platform for real-time pedestrian tracking based on a ZYNQ-7000 device," in *Proc. IEEE Int. Conf. Consum. Electron.-Asia (ICCE-Asia)*, Oct. 2017, pp. 127–132.

[14] P.-Y. Hsiao, S.-Y. Lin, and S.-S. Huang, "An FPGA based human detection system with embedded platform," *Microelectron. Eng.*, vol. 138, pp. 42–46, Apr. 2015.

[15] C. G. Blair and N. M. Robertson, "Video anomaly detection in real time on a power-aware heterogeneous platform," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 26, no. 11, pp. 2109–2122, Nov. 2016.

[16] X. Chen, J. Xu, and Z. Yu, "A fast and energy efficient FPGA-based system for real-time object tracking," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, Dec. 2017, pp. 965–968.

[17] Y. Nishizumi, G. Matsukawa, K. Kajihara, T. Kodama, S. Izumi, H. Kawaguchi, C. Nakanishi, T. Goto, T. Kato, and M. Yoshimoto, "FPGA implementation of object recognition processor for HDTV resolution video using sparse FIND feature," in *Proc. IEEE Int. Workshop Signal Process. Syst. (SiPS)*, Oct. 2017, pp. 1–6.

[18] M.-E. Ilas, "HOG algorithm simplification and its impact on FPGA implementation: With applications in car detection," in *Proc. 9th Int. Conf. Electron., Comput. Artif. Intell. (ECAI)*, Jun. 2017, pp. 1–6.

[19] M.-E. Ilas, "New histogram computation adapted for FPGA implementation of HOG algorithm: For car detection applications," in *Proc. 9th Comput. Sci. Electron. Eng. (CEEC)*, Sep. 2017, pp. 77–82.

[20] M. Bilal, A. Khan, M. U. K. Khan, and C.-M. Kyung, "A low-complexity pedestrian detection framework for smart video surveillance systems," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 27, no. 10, pp. 2260–2273, Oct. 2017.

[21] J. Rettkowski, A. Boutros, and D. Göhringer, "HW/SW co-design of the HOG algorithm on a xilinx zynq SoC," *J. Parallel Distrib. Comput.*, vol. 109, pp. 50–62, Nov. 2017.

[22] A. B. K., V. Venkatraman, A. R. Kumar, and S. D. S., "Accelerating real-time computer vision applications using HW/SW co-design," in *Proc. Int. Conf. Comput., Commun. Electron. (Comptelix)*, Jul. 2017, pp. 458–463.

[23] M.-E. Ilas, "Improved binary HOG algorithm and possible applications in car detection," in *Proc. IEEE 23rd Int. Symp. Design Technol. Electron. Packag. (SIITME)*, Oct. 2017, pp. 274–279.

[24] B. Meus, T. Kryjak, and M. Gorgon, "Embedded vision system for pedestrian detection based on HOG+SVM and use of motion information implemented in zynq heterogeneous device," in *Proc. Signal Process., Algorithms, Archit., Arrangements, Appl. (SPA)*, Sep. 2017, pp. 406–411.

[25] T. Sledevie, A. Serackis, and D. Plonis, "FPGA-based selected object tracking using LBP, HOG and motion detection," in *Proc. IEEE 6th Workshop Adv. Inf., Electron. Electr. Eng. (AIEEE)*, Nov. 2018, pp. 1–5.
[26] M.-S. Wang and Z.-R. Zhang, "FPGA implementation of HOG based multi-scale pedestrian detection," in *Proc. IEEE Int. Conf. Appl. Syst. Invention (ICASI)*, Apr. 2018, pp. 1099–1102.
[27] J. Luo and C. Lin, "Pure FPGA implementation of an HOG based real-time pedestrian detection system," *Sensors*, vol. 18, no. 4, p. 1174, 2018.
[28] V. Ngo, A. Casadevall, M. Codina, D. Castells-Rufas, and J. Carrabina, "A high-performance HOG extractor on FPGA," 2018, *arXiv:1802.02187*. [Online]. Available: http://arxiv.org/abs/1802.02187
[29] S.-S. Huang, S.-Y. Lin, and P.-Y. Hsiao, "An FPGA-based HOG accelerator with HW/SW co-design for human detection and its application to crowd density estimation," *J. Softw. Eng. Appl.*, vol. 12, no. 1, pp. 1–19, 2019.
[30] E. P. R. Raj, B. S. Paul, and G. L. Narayanan, "Simplified SIFT histogram of oriented gradients bin locator on FPGA," in *Proc. 9th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2018, pp. 1–4.
[31] M. Qasaimeh, J. Zambreno, and P. H. Jones, "A runtime configurable hardware architecture for computing histogram-based feature descriptors," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 351–3513.
[32] T. Adiono, K. S. Prakoso, C. Deo Putratama, B. Yuwono, and S. Fuada, "Practical implementation of a real-time human detection with HOG-AdaBoost in FPGA," in *Proc. TENCON IEEE Region Conf.*, Oct. 2018, pp. 0211–0214.
[33] X. Long, S. Hu, Y. Hu, Q. Gu, and I. Ishii, "An FPGA-based Ultra-High-Speed object detection algorithm with multi-frame information fusion," *Sensors*, vol. 19, no. 17, p. 3707, 2019.
[34] P. Ranawaka, M. Ekpanyapong, A. Tavares, J. Cabral, K. Athikulwongse, and V. Silva, "Application specific architecture for hardware accelerating HOG-SVM to achieve high throughput on HD frames," in *Proc. IEEE 30th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, New York, NY, USA, Jul. 2019, pp. 131–134.
[35] I. Ahmad, Z. Ul Islam, F. Ullah, M. Abbas Hussain, and S. Nabi, "An FPGA based approach for people counting using image processing techniques," in *Proc. 11th Int. Conf. Knowl. Smart Technol. (KST)*, Phuket, Thailand, Jan. 2019, pp. 148–152.
[36] J. Li, X. Liu, F. Liu, D. Xu, Q. Gu, and I. Ishii, "A hardware-oriented algorithm for Ultra-High-Speed object detection," *IEEE Sensors J.*, vol. 19, no. 10, pp. 3818–3831, May 2019.
[37] D. D. Gajski, *Principles of Digital Design*. Upper Saddle River, NJ, USA: Prentice-Hall, 1997.
[38] T. Wilson, M. Glatz, and M. Hodlmoser, "Pedestrian detection implemented on a fixed-point parallel architecture," in *Proc. IEEE 13th Int. Symp. Consum. Electron.*, May 2009, pp. 47–51.
[39] S. Bauer, S. Kohler, K. Doll, and U. Brunsmann, "FPGA-GPU architecture for kernel SVM pedestrian detection," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2010, pp. 61–68.
[40] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, San Diego, CA, USA, Jun. 2005, pp. 886–893.
[41] S. Ghaffari, P. Soleimani, K. F. Li, and D. Capson, "FPGA-based implementation of HOG algorithm: Techniques and challenges," in *Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Process. (PACRIM)*, Victoria, BC, Canada, Aug. 2019, pp. 1–7.
[42] Z. Xiang, H. Tan, and W. Ye, "The excellent properties of a dense grid-based HOG feature on face recognition compared to Gabor and LBP," *IEEE Access*, vol. 6, pp. 29306–29319, 2018.
[43] M. Awais, M. J. Iqbal, I. Ahmad, M. O. Alassafi, R. Alghamdi, M. Basheri, and M. Waqas, "Real-time surveillance through face recognition using HOG and feedforward neural networks," *IEEE Access*, vol. 7, pp. 121236–121244, 2019.
[44] W. Xing, N. Deng, B. Xin, Y. Liu, Y. Chen, and Z. Zhang, "Identification of extremely similar animal fibers based on matched filter and HOG-SVM," *IEEE Access*, vol. 7, pp. 98603–98617, 2019.
[45] N. Laopracha, K. Sunat, and S. Chiewchanwattana, "A novel feature selection in vehicle detection through the selection of dominant patterns of histograms of oriented gradients (DPHOG)," *IEEE Access*, vol. 7, pp. 20894–20919, 2019.
[46] M. Ehatisham-Ul-Haq, A. Javed, M. A. Azam, H. M. A. Malik, A. Irtaza, I. H. Lee, and M. T. Mahmood, "Robust human activity recognition using multimodal feature-level fusion," *IEEE Access*, vol. 7, pp. 60736–60751, 2019.

SINA GHAFFARI was born in Tehran, Iran, in 1992. He received the B.S. degree in electronic engineering from the University of Tehran, Tehran, Iran, in 2015, and the M.S. degree in digital electronics from the Amirkabir University of Technology, Tehran, in 2017. He is currently pursuing the Ph.D. degree in electrical and computer engineering with the University of Victoria, Victoria, BC, Canada.

His research interests include computer vision, image processing, hardware design, and machine learning. He has been awarded the University of Victoria Doctoral Fellowship.

PARASTOO SOLEIMANI was born in Gorgan, Iran, in 1992. She received the B.S. degree in electronic engineering from the University of Tehran, Tehran, Iran, in 2015, and the M.S. degree in electrical engineering-integrated circuits of electronic from the K. N. Toosi University of Technology, Tehran, in 2018. She is currently pursuing the Ph.D. degree in electrical and computer engineering at the University of Victoria, Victoria, BC, Canada.

Her research interests include computer vision, image processing, hardware design, and machine learning. She has been awarded the University of Victoria Doctoral Fellowship.

KIN FUN LI is the Director of the two highly sought-after professional master of engineering programs in telecommunications and information security (MTIS) and applied data science (MADS) at the University of Victoria, Canada, where he teaches both hardware and software courses in the Department of Electrical and Computer Engineering. He dedicates his time to instructing and researching in computer architecture, hardware accelerators, education analytics, and data mining applications. He is actively involved in the organization of many international conferences, including the biennial IEEE Pacific Rim in Victoria and the internationally held IEEE AINA. He is also a passionate supporter and participant in numerous international activities to promote the engineering profession, education, and diversity.

Dr. Li is an Honorary Member of the Golden Key and a Registered Professional Engineer in the Province of British Columbia.

DAVID W. CAPSON (Senior Member, IEEE) received the B.Sc.Eng. degree in electrical engineering from the University of New Brunswick, Fredericton, NB, Canada, in 1979, and the M.Eng. and Ph.D. degrees in electrical engineering from McMaster University, Hamilton, ON, Canada, in 1981 and 1985, respectively.

He was a Visiting Scientist at the IBM Almaden Research Center, San Jose, CA, USA, in 1989, and worked with CRS Robotics, Burlington, ON, Canada, in 1995, and with Gennum Corporation, Burlington, ON, from 2004 to 2005. From 1984 to 2012, he was a Professor with the Department of Electrical and Computer Engineering, McMaster University, serving as a Department Chair, from 2008 to 2012. In 2007, he was the Winner of the McMaster Student Union Lifetime Teaching Achievement Award. He is currently a Professor with the Department of Electrical and Computer Engineering, University of Victoria, BC, Canada, and the Dean of the Faculty of Graduate Studies. His research interests include computational vision, algorithms and architectures for accelerated and embedded image analysis, and machine vision-based applications in robotics, metrology, inspection, and servo systems.

Dr. Capson is a Registered Professional Engineer in the provinces of British Columbia and Ontario and an Honorary Member of the Golden Key International Honor Society.