A Fully Pipelined FPGA Architecture for Multiscale BRISK Descriptors With a Novel Hardware-Aware Sampling Pattern

Sina Ghaffari, David W. Capson, *Senior Member, IEEE*, and Kin Fun Li, *Senior Member, IEEE*

***Abstract*— Binary descriptors have been shown to be faster than nonbinary descriptors while producing comparable results in image matching applications. In recent years, there have been many attempts to design hardware accelerators for extraction of binary descriptors to achieve higher processing rates. One of the well-known methods is the binary robust invariant scalable key point (BRISK) algorithm, which has shown outstanding results in various applications. In this work, we propose a multiscale field-programmable gate array (FPGA)-based hardware architecture for the BRISK descriptor. In addition, a new image sampling pattern for the BRISK algorithm is described which is shown to be more efficient than the original sampling pattern for hardware implementation. Our new sampling pattern decreases the size of the patches containing the key point to one-quarter of the size of that used in the original BRISK algorithm, which leads to a reduction in FPGA resource utilization while maintaining the accuracy of the image matching application. Our proposed design is fully pipelined and achieves a frame rate of 78 fps on images with full HD resolution.**

***Index Terms*— Accelerator, BRISK, field-programmable gate array (FPGA), hardware implementation, sampling pattern.**

I. Introduction

KEYPOINT detection and description has many applications in computer vision. Key point detection is the process of finding the location of key points (or interest points), which are the points in the image such as corner features that represent important information. Extracting features from a patch (a small window of image which is being processed by the descriptor) around the key point is called key point description. Features are any information from the patch that can be used for specifying each key point individually. Extracted features should have high similarity for a key point which is visible in two different images while having low similarity with the features of other key points in the same image and other images. Invariance to illumination, scale, and rotation are important characteristics of feature descriptors.

There are many feature detection and description algorithms proposed in the literature. These algorithms are commonly categorized into two groups. The first one is nonbinary descriptors including SIFT [1], SURF [2], and HOG [3]. The second category is binary descriptors including BRIEF [4], FREAK [5], BRISK [6], and ORB [7]. Nonbinary descriptors usually generate histograms of image features such as gradients and use them for describing a patch of an image. On the other hand, binary descriptors are commonly based on the comparison of the intensity of different pairs of pixels in a patch around a key point. Since nonbinary descriptors process more information, they typically produce more accurate results. However, the main advantage of binary feature descriptors over nonbinary ones is faster computation while maintaining comparable accuracy.

A. Hardware Implementation of Descriptors

Although binary descriptors have been shown to produce a noticeable performance enhancement in terms of speed compared to nonbinary descriptors, they remain computationally expensive. This has led many researchers to work on hardware implementation of binary descriptors to achieve higher speed. Implementing hardware accelerators can make computer vision applications more practical due to the benefits of processing multiple computations in parallel. Hardware accelerators do not have the limitations of processors with conventional architectures. In particular, field programmable gate arrays (FPGAs) are popular platforms for implementing hardware accelerators for their ease of implementation and reasonable time-to-market in comparison with application-specific integrated circuits. There are many attempts to implement descriptor algorithms such as HOG, ORB, and FREAK on FPGAs due to their low power consumption and parallel computation capabilities. As an example, an analysis of FPGA-based implementation of the HOG algorithm is provided in our previous work [8].

B. BRISK Algorithm

The BRISK algorithm combines the AGAST detector [9] and a new descriptor [6]. Since the focus of our work is on the description portion of the BRISK algorithm, we briefly introduce the description process of BRISK in this section.

When a key point is detected in an image, a patch of pixels around that key point is extracted. Then, specific locations of pixels around the key points are used as samples of the patch to generate the descriptors. The original BRISK algorithm uses a sampling pattern as shown in Fig. 1 [6].

Manuscript received September 23, 2021; revised January 13, 2022; accepted February 4, 2022. Date of publication March 2, 2022; date of current version May 23, 2022. This work was supported in part by Doctoral Fellowships from the University of Victoria, and in part by the Natural Sciences and Engineering Research Council of Canada under Discovery Grant 36401 and Discovery Grant 04787. *(Corresponding author: Sina Ghaffari.)*

The authors are with the Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC V8W 2Y2, Canada (e-mail: sinaghaffari@uvic.ca).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2022.3151896.

Digital Object Identifier 10.1109/TVLSI.2022.3151896

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

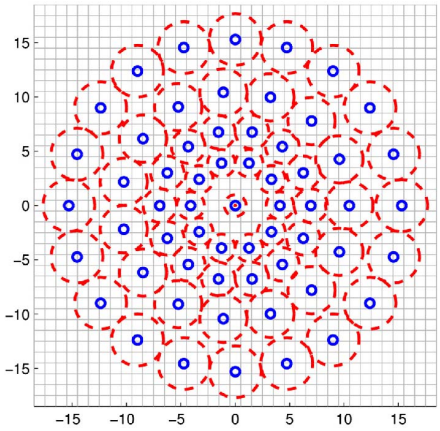

Fig. 1. Original BRISK sample points [6] (© [2011] IEEE). The blue circle in the center represents the key point. Other blue circles represent sample points. The red dashed circles represent the relative variance of the Gaussian filter which is applied around each sample point. The key point is positioned on (0,0) and the numbers on the vertical and horizontal axes represent the relative position from the origin, in pixels.

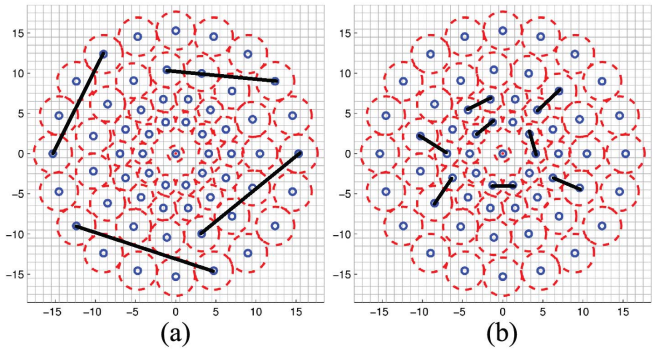

Fig. 2. Examples of (a) long pairs and (b) short pairs based on the BRISK sample pattern [6] (© [2011] IEEE).

The sampled pixels around a key point in a patch form groups as pairs and depending on the distance between each sample in a pair, are labeled as a *long distance pair* or a *short distance pair* [6]. If the distance is larger than a specific threshold, that pair is labeled as a long pair. Otherwise, it is labeled as a short pair. Fig. 2 shows examples of long pairs and short pairs. In the original BRISK algorithm, 870 long pairs and 512 short pairs are selected from the 60 samples including the key point. After selecting the long pairs, they calculate the orientation of the patch. The first step is to compute the gradient of each long pair as follows:

$$g(P_i, P_j) = \frac{P_i - P_j}{\|P_i - P_j\|} \times \frac{I(P_i, \sigma_i) - I(P_j, \sigma_j)}{\|P_i - P_j\|} \tag{1}$$

where P_i and P_j are the coordinates of sample points in each pair and $I(P_i, \sigma_i)$ is the intensity of an image smoothed by a Gaussian kernel with variance of σ_i in location P_i. After computing the gradient for each pair, the summation of all the gradients of long pairs is computed in horizontal and vertical directions together as shown in the following:

$$G = \begin{pmatrix} g(x) \\ g(y) \end{pmatrix} = \frac{1}{L} \sum_{P_i, P_j \in \text{patch}} g(P_i, P_j). \tag{2}$$

Finally, the patch orientation is determined by computing the arctangent of G as shown in the following:

$$\theta = \arctan\left(\frac{g(y)}{g(x)}\right) \tag{3}$$

where θ is the orientation of the patch. Each patch has its own main orientation, which represents the main gradient direction of that patch. If we rotate all the patches so that their main orientation is in one direction, then the patches of the same key point in two different images with different orientations will transform into the same direction. Therefore, in the next step, the algorithm rotates each patch by its orientation. Then, BRISK compares the intensity value of each pixel from short pair samples with the other pixel in the same pair. Instead of comparing the raw intensity of the pixels, they smooth the image using the pixels around it. Smoothing has a significant effect on the accuracy of the algorithm. If the smoothed value of the first pixel is greater than the second one, they set the corresponding bit in the descriptor to 1 and otherwise they set it to 0.

C. New Approaches to Enhance Acceleration

There has been little publication in the academic literature focusing on implementation of the BRISK algorithm on FPGAs, particularly at multiple scales. In this work, we propose and evaluate a design of a multiscale hardware implementation of the BRISK algorithm to achieve scale invariant performance.

We propose a novel design of a multiscale pipeline architecture, including innovations in various stages of the pipeline. Scale invariant descriptors can match objects of various sizes in images. There are two conventional methods for implementing multiple scales of descriptors in hardware. The first method is to extract descriptors from multiple frame sizes sequentially, as done by Liu *et al.* [10] for the ORB descriptor. The second method is to replicate the buffers and computational components for the number of scales to process multiple scales in parallel as used by Sun *et al.* [11]. The second method is faster but requires more hardware resources in comparison with the first method.

Our multiscale hardware implementation of the BRISK algorithm requires less hardware resources than trivial replication of the circuits for single scale. In particular, we store and process a quarter of the resized image in each scale without loss of accuracy. The pipeline architecture in our design is synchronized so that we can share one of the key computation stages of the algorithm between multiple scales.

We also introduce an innovative hardware-aware sampling pattern which is designed based on minimization of hardware resources, which facilitates the implementation of the BRISK algorithm for multiple scales. Using this sampling pattern, we process one-quarter instead of the full image data in each scale while maintaining comparable accuracy. We demonstrate a fully pipeline architecture for this algorithm which has extensive parallelism in each stage of the pipeline. In addition, we use a variety of techniques such as resource sharing, clock gating, and computational approximations to make our design more efficient in terms of power and resource utilization.

TABLE I
COMPARISON OF EXAMPLES OF BINARY DESCRIPTORS

Descriptor algorithm	Original detector	Scale invariance	Sampling method	Rotation computation method
BRIEF [4]	CenSure [18]	No	Random pairs of pixels in the patch	—-
FREAK [5]	Multi-scale AGAST [9]	Yes	Predefined samples for orientation and description	Main direction computed based on predefined samples
BRISK [6]	Multi-scale AGAST [9]	Yes	Long pairs for orientation, short pairs for description	Main direction computed based on long-pair samples
ORB [7]	FAST [19]	Yes	Random pairs of pixels in the patch	Main direction computed based on intensity centroid of the patch

D. Organization of this Article

The remainder of this article is presented as follows. In Section II, we compare well-known binary descriptors and discuss recently published FPGA implementations of these algorithms. In Section III, we introduce our new hardware-aware sampling pattern for the BRISK algorithm. In Section IV, we present a novel multiscale hardware implementation of the BRISK algorithm. In that section, we provide our hardware design and solutions in detail for each part of the BRISK algorithm. We provide a comprehensive analysis of the benefits of our contributions in Section V and evaluate our innovations in comparison with recently published, state-of-the-art results. Finally, we conclude this article in Section VI.

II. RELATED WORK

In this section, we provide a comparison of commonly-known binary descriptors. We continue this section by reviewing recent advances in hardware implementation of binary descriptors then focus on work which implements the BRISK algorithm.

A. Comparison of Binary Descriptor Algorithms

Although binary descriptors operate similarly in that they use comparison for generating their output, there are important differences in the associated algorithms. Table I shows the main differences in sampling and rotation computation among four well-known binary feature descriptors. In this work, we propose a new design for the BRISK descriptor since it has many applications in various specialized computer vision fields such as bone age assessment [12], SAR-based automatic target recognition [13], content-based image retrieval [14], and emotion recognition [15]. It requires fewer computations than nonbinary descriptors such as SIFT and SURF while it produces comparable results. The FREAK descriptor is very similar to BRISK with the difference between them being the sampling pattern and rotation invariant calculation. The samples in FREAK are mostly focused on the center of the patch while BRISK has a more uniformly distributed pattern throughout the patch. Therefore, for various applications they produce close but different results. BRISK achieves better results than BRIEF since it is rotation and scale invariant. Also, since the original BRISK generates 512-bit descriptors while ORB generates 256-bit descriptors, it has been shown to outperform the original ORB in specific applications [16]. However, BRISK is more computationally demanding since it processes a larger number of pixels from the same patch size. Mouats *et al.* [17] evaluated BRISK, FREAK, ORB, and nonbinary descriptors in poor lighting conditions. In most test cases, BRISK outperforms other binary descriptors in accuracy.

Binary descriptors are faster than nonbinary descriptors since they provide a binary vector based on the comparisons of pixel values in the final feature vector, while nonbinary descriptors normally have more complex computations such as histogram generation and dense gradient computation. Speed comparison of descriptors have been addressed in previous work [20]–[22]. However, a descriptor such as BRISK will still require a high number of computations. In the original work, for each key point, the algorithm requires 870 long-pair computations and 512 short-pair comparisons which are processed sequentially on a conventional CPU.

In our work, we propose an FPGA-based design to compute the descriptors faster in multiple scales by performing the calculation of long pairs and short pairs in parallel. Multiscale description results in a more accurate image matching system.

B. FPGA-Based Implementations of Binary Descriptors

There are many FPGA-based implementations of binary descriptors. Sun *et al.* [11], [23], de Lima *et al.* [24], Liu *et al.* [10], and Tran *et al.* [25] implement the ORB algorithm on hardware. Fang *et al.* [26] implement ORB algorithm on an FPGA for two scales of 640 $\times$ 480 and 533 $\times$ 400. They stop the streaming input whenever a key point enters the line buffers so that the descriptor has enough time to calculate the features. This design choice leads to lower frame rate. They achieve a frame rate of 67 fps with maximum frequency of 203 MHz for 640 $\times$ 480 resolution. Kalms *et al.* [27] and Kapela *et al.* [28] proposed hardware architectures for the FREAK descriptor on an FPGA and Pham *et al.* [29] design an FPGA-based architecture for rotation-aware BRIEF algorithm. Although the BRIEF algorithm requires less calculation than BRISK, they achieve 60 fps for 1920 $\times$ 1080 images. For comparison, in our work, we achieve 78 fps on 1920 $\times$ 1080 images.

C. FPGA-Based Implementations of BRISK Descriptor

Despite the large number of computations for orientation compensation, there has been little work focusing on hardware implementation of the BRISK algorithm. Ulusel *et al.* [30] implement the BRISK algorithm on an FPGA. They implement a single-scale version of the BRISK algorithm with an

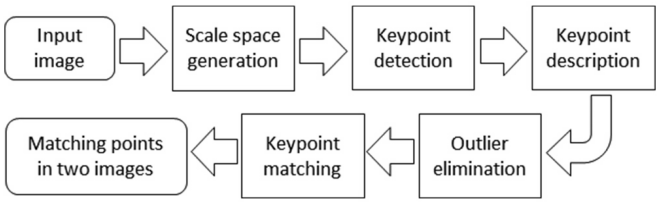

Fig. 3. Image matching pipeline.

800 × 480 image resolution. Their work does not contain details of orientation compensation which is an important part of the description. They also do not provide any results for demonstrating accuracy. Azimi *et al.* [31] proposed a fully pipelined and parallel hardware architecture for the detector part of the BRISK algorithm which is a multiscale FAST [19] algorithm. Since their focus is only on detection, they have not implemented the descriptor part of BRISK.

A complete image matching system which includes scale space generation, key point detection, key point description, key point matching, and outlier elimination is shown in Fig. 3. In this work, we focus on the descriptor and implement it with a hardware accelerator. We also implement a FAST detector as a part of the matching system. However, the descriptor could be combined with any feature detector unit whether implemented in software or hardware.

III. Hardware-Aware Sampling Pattern for BRISK

One of the key differences among binary descriptors is their image sampling pattern. In this section, we propose a new, novel sampling pattern which is more efficient in resource utilization for hardware implementation than the sampling pattern of the original BRISK algorithm. The main idea is to constrain the sampling points to be only in even (or odd) rows and columns. We achieve similar accuracy with patterns in even coordinates, while reducing the processing requirements to one-quarter of the original algorithm.

Fig. 4 presents examples selected from the sequence of steps for generating the proposed sampling pattern. Similar to BRISK, our proposed sampling pattern is based on a 33 × 33 patch. To obtain this pattern, we start with a grid of 17 × 17 pixel locations. First, all the pixels around the center of the grid are selected as initial sampling points as shown in step 0 of Fig. 4. Then, for a specific number of rotations n, we determine the angle step θ based on the following:

$$\theta = \frac{360}{n}. \tag{4}$$

We rotate the grid for every $\theta = 10°$ ($n = 36$) and compute the Euclidean distance of the center of each rotated pixel with the nearest unrotated ones. If this distance is less than a predefined threshold T, we mark it as an acceptable overlap. For each degree, we calculate the acceptable overlaps between samples. The samples that do not have acceptable overlaps are removed as shown in the examples of Fig. 4. Last, we have the intersection of all acceptable overlaps for all rotations. The resulting samples are the pixels which have acceptable overlaps with each other in all angle steps θ for the specified threshold. If we reduce the threshold, the number of acceptable overlaps becomes smaller, which is shown in Table II. Higher

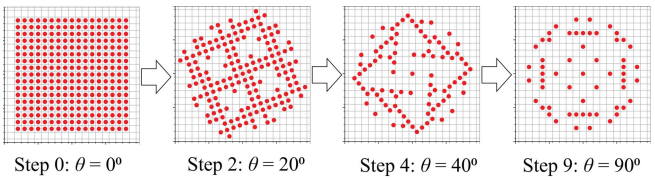

Fig. 4. Examples selected from the sequence of steps for generating the proposed sampling pattern (with $T = 1$, $n = 36$). The initial sampling points are shown as step 0. Each step rotates the sampling points by $\theta = 10$ and the samples without acceptable overlap are removed. Not all steps are shown for brevity.

TABLE II
Effect of Changing the Distance Threshold T on the Number of Sample Points With $n = 36$ ($\theta = 10$)

Distance threshold T	Number of sample points
0.98	21
1.00	57
1.02	65
1.04	73

TABLE III
Effect of Changing the Number of Rotations on the Number of Sample Points With $T = 1$

Number of rotations n	Angle step θ	Number of sample points
1	360	289
12	30	169
24	15	89
36	10	57
48	7.5	33
60	6	1

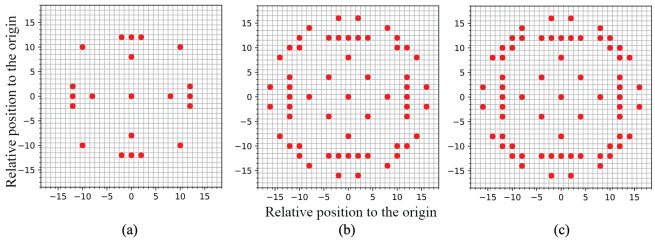

Fig. 5. Examples of sampling patterns resulting from changing threshold (T) to produce different sampling patterns for the same number of rotations n. For all three patterns $n = 36$ ($\theta = 10$). (a) $T = 0.98$, (b) $T = 1$, and (c) $T = 1.02$.

thresholds lead to a greater number of acceptable overlaps. However, if we accept overlaps with higher distances it will decrease the accuracy of matching since the matching algorithm would assume those pixels have the same position in rotated images. As shown in Table III, decreasing the angle step θ leads to more precise rotation of the pixels. Therefore, the acceptable overlaps are selected from the intersection of more pixel rotations, and the number of pixels which are available in all possible rotations decreases. As a consequence, the number of acceptable overlaps decreases.

Finally, we multiply each coordinate by 2 so that the samples would be extracted from only even rows and columns of a 33 × 33 patch. Fig. 5 shows examples of sampling patterns using different thresholds. Fig. 6 presents different sampling patterns obtained from different angle steps.

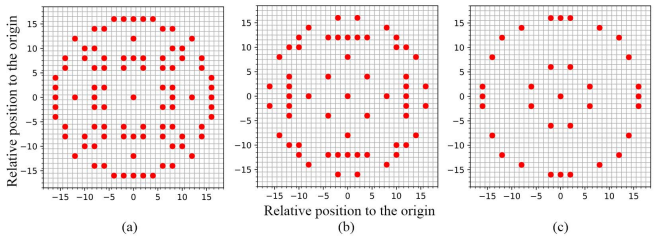

Fig. 6. Examples of sampling patterns resulting from changing the number of rotations n to produce different sampling patterns for the same threshold T. For all three patterns $T = 1$. (a) $n = 24$ ($\theta = 15$), (b) $n = 36$ ($\theta = 10$), and (c) $n = 48$ ($\theta = 7.5$).

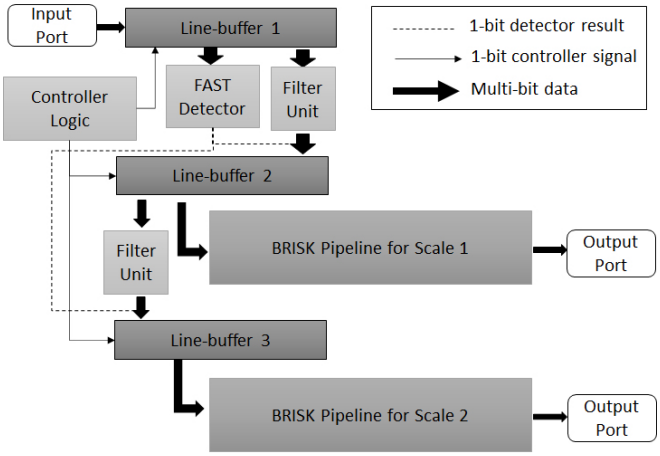

Fig. 7. Overall architecture of our design. This design includes two parallel pipeline paths for two scales. The controller logic is a combination of multiple finite state machines. The dashed line is a single-bit result of the detector which is concatenated with the pixel values.

We choose a threshold of 1 pixel and 36 angle steps for the sampling pattern. This means that we have samples for each 10° of rotation and the sample points are the ones which have the shortest distance to at least one sample point in another rotation of the same sampling pattern. This design choice gives us enough samples to describe a patch with adequate accuracy but not so many samples such that FPGA routing is unnecessarily complicated.

IV. Hardware Implementation of the Multiscale BRISK Algorithm

In this section, we introduce the architecture of our design for the BRISK algorithm implementation. The proposed design is a fully pipelined architecture so that at each clock cycle, a new pixel enters the pipeline without a need to pause. In this design, our main focus is first, to achieve higher speed by parallelizing the computations required in each step of the algorithm and second, to reduce hardware resource usage by using our novel sampling pattern. The proposed architecture is shown in Fig. 7. The controller is a combination of finite state machines (FSMs). We discuss the multiscale structure of this architecture in more detail in Section IV-B. The streaming input pixel enters the design one pixel at each clock cycle. The pixels enter a line buffer which has a size of $W \times 11$ (where W is the width of the image). After an initial waiting phase, the line buffer becomes full and the output of the line buffer which is an 11×11 patch will have valid values. We use the 11×11 patch at the end of the line buffer as an input for the FAST detector and the filter unit.

The FAST detector tests each 11×11 patch to determine if the center pixel of the patch is a key point. This module has two main parts, the lighter pixels key point test and the darker pixels key point test. These two parts process the pixels in parallel. First, we extract the values of the 16 pixels in a 7×7 patch around the center pixel shaping a Bresenham circle as shown in Fig. 8. We define an upper limit and a lower limit which specify a neighborhood around the center pixel value. We calculate the upper limit by adding the value of the center pixel to a predefined threshold, and at the same time, we calculate the lower limit by subtracting the center pixel from the same threshold. This threshold is used for reducing the effect of noise on detection result. We choose the value of 10 for this threshold based on our experiments. The upper limit is used for the lighter pixels test and the lower limit is used for the darker pixels test.

For the lighter pixels test, we compare all 16 pixels around the center pixel with the upper limit simultaneously to see if nine consecutive pixels are lighter than the center pixel. We use logic gates to AND the "greater than" output of every nine consecutive comparators. If any one of the AND operations result is true, the center pixel is identified as a key point. We logically OR the results of the AND operations. The result of the OR gates is a 1-bit key point detection output.

Simultaneously, we compare the pixels around the center pixel with the lower limit to check the criterion for darker pixels. For darker pixels, we check the "less than" output of the comparators. Finally, we OR the result of the darker and lighter key point detection to form the output of the detector module. Since we have 16 comparators and 16 9-bit AND gates, 8 of the inputs of each two adjacent AND gates are common between them. Note that in Fig. 8, signals *gt_1* to *gt_9* are connected to the top AND gate and signals *gt_8* to *gt_16* are connected to the bottom AND gate as an example. In this example, *gt_8* and *gt_9* are common inputs of these two gates.

The filter unit contains a constant weight window that has higher values in the center and lower values toward the borders. The highest value in the center is 1 and the lowest value at the borders is 0.5 as shown in Fig. 9. Due to the focus on the pixel in the center and keeping the effect of the surrounding pixels, these weights lead to an acceptable accuracy for our design. We multiply each pixel in the input patch with the corresponding value of the filter patch. Since the weights are constants, we use logical shift and adder logic. We approximate each multiplication by logical shift and adder logic with four terms as shown in the following:

$$M_{i,\text{approx}} = \sum_{j=1}^{4} M_i \gg n_i(j) = \sum_{j=1}^{4} M_i \times \left(\frac{1}{2^{n_i(j)}}\right). \quad (5)$$

In (5), M_i is the multiplicand, $M_{i,\text{approx}}$ is the approximated multiplicand, and n_i is an array of four values. The values for each $n_i(j)$ is selected so that the difference between M_i and $M_{i,\text{approx}}$ is acceptable. As an example, for appoximating 0.9, we can use $n_i = 1, 2, 3, 5$ which results in $M_{i,\text{approx}} = 0.90625$.

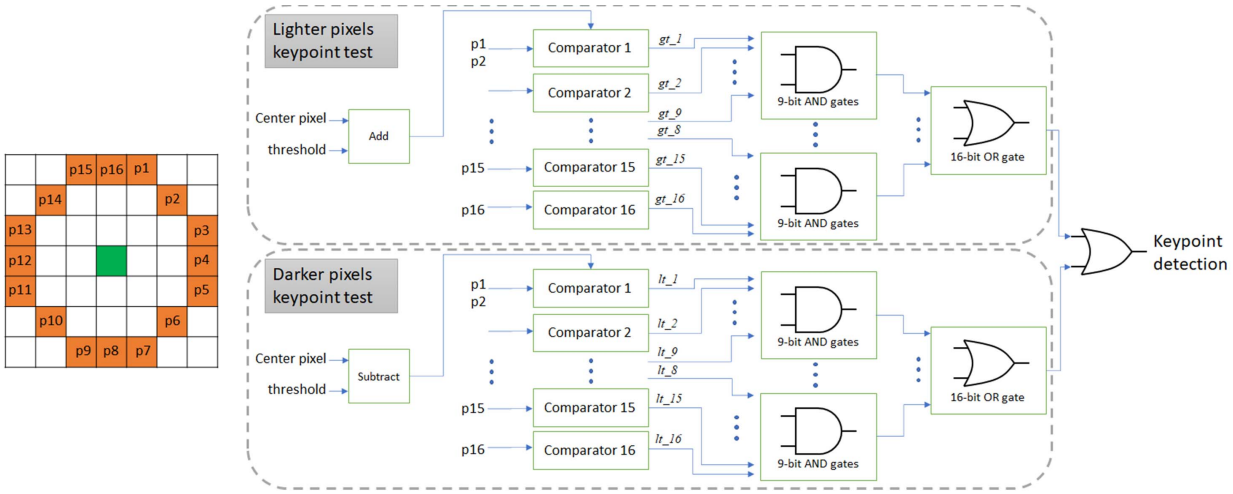

Fig. 8. Architecture of the FAST key point detector. The window on the left shows the location of the 16 pixels around the center pixel. The two key point tests process the pixel values in parallel.

TABLE IV
APPROXIMATIONS OF MULTIPLICATIONS IN THE FILTER UNIT

Desired multiplier	Approximated multiplier	n_i
0.9	0.90625	1, 2, 3, 5
0.8	0.796875	1, 2, 5, 6
0.7	0.695312	1, 3, 4, 7
0.6	0.6015625	1, 4, 5, 7
0.5	0.5	1

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.5
0.5	0.6	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.6	0.6	0.5
0.5	0.6	0.7	0.8	0.8	0.8	0.8	0.8	0.7	0.6	0.6	0.5
0.5	0.6	0.7	0.8	0.9	0.9	0.9	0.8	0.7	0.6	0.6	0.5
0.5	0.6	0.7	0.8	0.9	1.0	0.9	0.8	0.7	0.6	0.6	0.5
0.5	0.6	0.7	0.8	0.9	0.9	0.9	0.8	0.7	0.6	0.6	0.5
0.5	0.6	0.7	0.8	0.8	0.8	0.8	0.8	0.7	0.6	0.6	0.5
0.5	0.6	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.6	0.6	0.5
0.5	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Fig. 9. Weights of the filter unit that is shown in Fig. 7.

Table IV shows the values of each n_i and the approximations we use in this stage. Subsequently, we compute the summation of weighted pixels using a 7-level adder tree. Finally, we divide the output by the summation of all weights in the weight window in Fig. 9. Since this value is a constant, we use right shift and adder logic instead of division to approximate this value, which produces a negligible error.

The filter unit and the detector unit work in parallel. We concatenate the detector result (which is one bit indicating if the pixel is a key point or not) to the 8-bit value of the pixel. The result, which is a 9-bit value, enters the second line buffer which contains the smoothed values of the pixels in the image. The length of the second line buffer is half of the image width. By using even rows and columns, we do not lose any information since our sampling pattern is located on the pixel locations that remain in this line buffer. The advantage of this technique is that we can consider only one-quarter of the pixels in the image from this step forward since the sampling pattern does not require pixel values from adjacent locations. The output of the second line buffer is a 17×17 patch which can be accessed when this line buffer becomes full of valid data. We send this patch to the first step of the BRISK pipeline which is the long-pair subtraction unit.

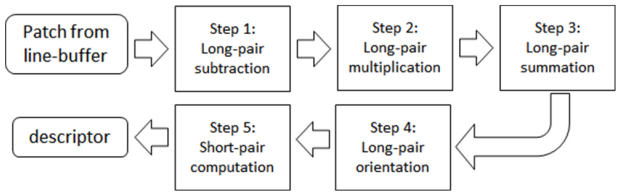

Fig. 10. Block diagram of a pipeline architecture for one scale.

A. BRISK Pipeline

The steps for BRISK pipeline include long-pair subtraction, multiplication, summation, orientation calculation, and short-pair comparison. In the proposed design, these stages are separated using registers to form a pipeline architecture. Therefore, at each clock cycle, each stage processes the data of a specific patch while the previous stage is processing the adjacent patch on the left. Fig. 10 shows the block diagram of the architecture of the pipeline for one scale in our design. We use the detector bit (the 9th bit) of the output value of the line buffer as an enable signal for the pipeline stages. Therefore, if the pixel entering the pipeline architecture is not a key point, the pipeline will not continue to work and we save dynamic power. We discuss the advantages of this method in Section V. In this design, we compute the gradients in horizontal and vertical directions in a different order than the original BRISK algorithm. Since in the hardware implementation the coordinates of the samples are known, we can

factor the coordinate terms and precompute some parts of the expressions in (1) and (2), and use them as coefficients as in (6) and (7):

$$g(x) = \sum_{P_i, P_j \in \text{patch}} \frac{1}{L} \frac{(P_i(x) - P_j(x))}{\|P_i - P_j\|^2} (I(P_i, \sigma_i) - I(P_j, \sigma_j)) = \sum_{k=1}^{870} c_1(k)(I(P_{k,1}, \sigma) - I(P_{k,2}, \sigma)) \quad (6)$$

$$g(y) = \sum_{P_i, P_j \in \text{patch}} \frac{1}{L} \frac{(P_i(y) - P_j(y))}{\|P_i - P_j\|^2} (I(P_i, \sigma_i) - I(P_j, \sigma_j)) = \sum_{k=1}^{870} c_2(k)(I(P_{k,1}, \sigma) - I(P_{k,2}, \sigma)) \quad (7)$$

where

$$c_1(k) = \frac{1}{L} \frac{P_{k,1}(x) - P_{k,2}(x)}{|P_{k,1} - P_{k,2}|^2}$$
$$c_2(k) = \frac{1}{L} \frac{P_{k,1}(y) - P_{k,2}(y)}{|P_{k,1} - P_{k,2}|^2} \quad (8)$$

and L is the number of long pairs which is 870. Note that c_1 and c_2 are precomputed constants and we can use their values for long-pair calculations.

The first step of long-pair calculation, which is shown in Fig. 10, is to compute the difference between the intensity of pixels in each long pair. In this step, we compute 870 subtractions in parallel in one clock cycle which results in 9-bit outputs to accommodate overflow. The second step is multiplication by c_1 and c_2 coefficients. For this step, we should multiply the 870 differences once by c_1 and once by c_2. Therefore, we have two channels of multiplication with the total number of 1740 multiplication operations which we perform in parallel in one clock cycle. We use add and signed arithmetic shifts instead of complex multiplier circuits since c_1 and c_2 are constants. In this way, fewer hardware resources are used for this step and the output is computed in a single clock cycle.

The third step is the summation step as shown in (6). For this step, we use two parallel 10-level adder trees. The input of the first level for each adder tree is the results of the multiplications of each channel. The two outputs of this step are the values of $g(x)$ and $g(y)$. The block diagram of this step is shown in Fig. 11. We use a pipeline register between levels 5 and 6 since this step is one of the critical timing paths of the design. The input values of the first level of adder tree, which are the output of the multiplication step, are 18-bit integers. The output of each level of the adder tree should have one more bit than the input level to accommodate overflow in addition. Increasing the number of bits in each level will result in 28-bit output at the final stage. In this step, we take advantage of the fact that in the next step, the result of the adder tree for the vertical direction is divided by the result of another adder tree for the horizontal direction. Therefore, if both these values are divided by a constant, the final result will not change.

In order to save hardware resources, we use a stepwise approximation in the adder tree. At each level, we divide the result of the addition by a specific power of 2. Division by power of 2 is equivalent to selecting bits and propagating them to the next level. By using this method, the final result of the next step (after division) is comparable to the result using accurate addition using all bits. For the first level, we divide the output by 8. For the second to fifth level, we divide the output by 4. For the rest of the levels, we divide the output of each level by 2. We choose these values empirically to minimize hardware resource utilization while maintaining similar accuracy. Table V shows the number of bits before approximation and after approximation in each level. The fourth step of the long-pair calculation is orientation computation. In this step, we compute the multiplication of $g(x)$ and different values of $\tan(\theta)$ limits and compare them to $g(y)$ as in the following:

$$g(x)\tan(\theta_{i+1}) > g(y) \geq g(x)\tan(\theta_i). \quad (9)$$

To make the descriptor rotation invariant, we should rotate the samples based on the orientation from step four. Based on the orientation value, we dynamically select the rotated samples using multiplexers working in parallel after the pipeline registers. This method does not require reading any data from memory and is faster than computing the rotated sample locations. The block diagram of the fifth step, which is short-pair computation, is shown in Fig. 12. After finding the correct angle, instead of rotating the patch, we use the sample pairs for that specific orientation. In this step, we have 1024 36-input multiplexers and 512 8-bit comparators. The orientation is

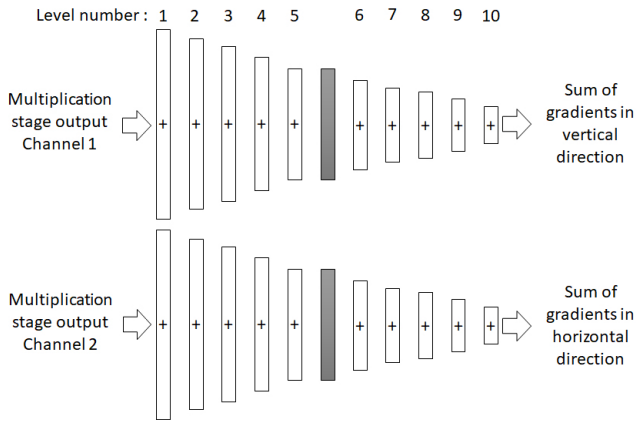

Fig. 11. Two parallel 10-level adder trees for two channels. Each level has half of the number of adders in the previous level. The solid bar represents a pipeline register which is used to make the critical path shorter.

TABLE V
ADDER TREE NUMBER OF ADDITIONS AND APPROXIMATIONS

Adder tree levels	Number of additions in each level	Pre-approximation output (bits)	Approximate output (bits)
1	864	19	16
2	432	20	15
3	216	21	14
4	108	22	13
5	54	23	12
6	27	24	12
7	14	25	12
8	7	26	12
9	4	27	12
10	2	28	12

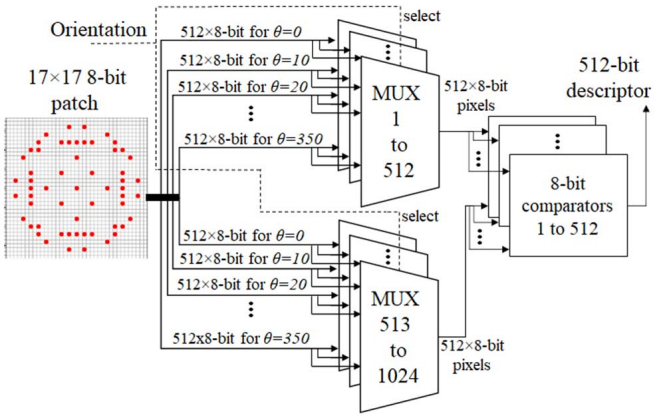

Fig. 12. Short pair computation step. In this stage, we have 72 buses which connect the pixels in the sampling pattern to the multiplexers. Each multiplexer selects one group of samples and passes it to the 512 comparators. The descriptor comprises the outputs of the comparators.

used as a selector of the multiplexers. The output of the multiplexers are the rotated samples and the output of each two multiplexers corresponding to one short pair are connected to a comparator. Finally, the output of the 512 comparators are concatenated as a single 512-bit vector which is the final descriptor of a key point.

B. Multiscale BRISK

We use two similar parallel pipelines to implement multiscale description. This architecture is shown in Fig. 7. The only difference between the two pipelines is the streaming input data. The input data of the second pipeline is extracted from the second line buffer. An 11×11 patch from the second line buffer enters the filter unit of the second pipeline. Then, the filtered result enters the third line buffer which produces a 17×17 patch as input to the second pipeline.

The first line buffer has the dimensions of $W \times 11$ (recall W is the image width). We choose this size since the purpose of this line buffer is to provide parallel input for the filtering unit, which contains a weight window of 11×11. The second line buffer has the dimensions of $W \times 17$. This line buffer has two outputs. The first one is a 17×17 patch which is the input of the first pipeline and the second output is an 11×11 patch which enters the second pipeline stage. The third line buffer has the dimension of *half of the second line buffer width* $\times$ 17. Since the original input enters the system each clock cycle and we are scaling each line buffer to have approximately half of the size of the previous one, valid data in the second line buffer is available every two clock cycles. Similarly, the valid data in the third line buffer is produced once every four clock cycles.

Our design provides the output of two scales for the modified BRISK algorithm. Additional scales can be added to the design by adding more pipelines similar to the second scale as shown in Fig. 7.

V. Results and Discussion

In this section, first, we discuss the performance of our new sampling pattern. Second, we present the advantages of clock gating on our implementation and the benefits of sharing FPGA resources for the multiplication stage. Then, we present the FPGA resource usage of our implementation of the BRISK algorithm in detail. After that, we demonstrate the accuracy of our implementation.

A. Assessment of the New Sampling Pattern

It is important to note that we use even rows and columns in our implementation but if odd rows and columns are used, the concept and the result will be the same. Using only the samples in even rows and even columns gives us an opportunity to process an image of diminished size without loss of information. Therefore, we can have a more efficient design regarding the resource usage on the FPGA. In the original BRISK, each patch has a minimum size of 33×33. If we load the patch and the surrounding pixels from the memory so that we can smooth the patch around the samples in the borders, we should use a 33×33 patch. Therefore, implementing the original BRISK for eight scales requires $W \times 61.875$ registers for line buffers according to (10). However, by using the new sampling pattern, we can reduce the number of required registers to about 27 times the width of the image as in (11). As shown in (10) and (11), for each scale, the width of the image is reduced to half size, and therefore, the width of the line buffer in each scale is equal to width of the image divided by the scale factor of that scale

$$R_{\text{BRISK}} = 33 \times \left(\frac{W}{1} + \frac{W}{2} + \frac{W}{4} + \frac{W}{8}\right) = 61.875 \times W \tag{10}$$

$$R_N = 17 \times \left(\frac{W}{2} + \frac{W}{4} + \frac{W}{8} + \frac{W}{16}\right) + 11 \times W = 26.9375 \times W. \tag{11}$$

In (10) and (11), W is the width of the image, R_{BRISK} is the number of registers required for a 4-scale implementation of the original BRISK pattern line buffers and R_N is the number of registers required for our design. We use 17×17 patches for the scales and we have a buffer of $W \times 11$ for prefiltering the first patch. Equations (10) and (11) show that we have 56% reduction in the number of registers in the design. Another advantage of reducing the size of the patch is that we propagate fewer number of signals through the pipeline. Table VI shows the number of registers used in each pipeline stage. In total, the number of registers decreases by 73% which is another benefit of our proposed sampling pattern.

B. Effect of Clock Gating

The first two steps of an image matching system are key point detection and key point description. For the first part, we should apply the key point detection algorithm on all pixels in the image to identify the key points. Unless a dense description of all pixels is required, the description part is applied to patches around a detected key point which is the common procedure used in most applications. Our proposed architecture can produce dense descriptors for all pixels in the image. However, we can use the output of the detection stage as a control signal to produce the description results, only for the key points. In this way, the part of the circuit which is

TABLE VI
NUMBER OF REQUIRED REGISTERS FOR PIPELINE IMPLEMENTATION OF THE BRISK ALGORITHM IN ORIGINAL BRISK AND OUR PROPOSED PATTERN

Stages of the pipeline implementation	Original BRISK**	Our proposed pattern
Long-pair subtraction, multiplication, and short-pair computation	5×(33×33)	5×(17×17)
Long-pair summation and orientation	2×(33×33×2*)	2×(17×17×2*)
Total number of registers for passing through the pipeline	7×33×33=7623	7×17×17=2023

*We have two channels in the long-pair summation stage and in the input of long-pair orientation stage.

**Values of this column are our estimation of the resources for implementing the original BRISK algorithm.

TABLE VII
POWER CONSUMPTION IMPROVEMENT USING CLOCK GATING

Power metrics		Ungated clock (watts)	Gated clock (watts)	Improvement
Static power		0.486	0.482	1%
Dynamic power	Clocks	0.733	0.372	49%
	Signals	0.102	0.013	87%
	Logic	0.255	0.190	25%
	I/O	0.003	0.003	0%
	Total	1.093	0.578	47%
Total Power		1.579	1.060	32%

dedicated for description does not consume dynamic power when there is no key point detected.

To implement this feature, we use a clock gating method. We concatenate the detection bit to the 8-bit pixels in the second and third line buffers. If the pixel is a key point, the 9th bit is 1; otherwise it is 0. We use the detection bit as the control signal for the pipeline registers on the FPGA. When a key point pixel reaches the end of the line buffers, the 9th bit, which indicates if the pixel is a key point or not, is used as an enable signal for the clock routing to all registers at that scale.

Since the computation of each pixel requires nine clock cycles in the descriptor pipeline, the clock of the pipeline registers stays active for nine consecutive clock cycles after each key point. Table VII shows the effect of gating the clock for the descriptor on power consumption versus using the same clock for all units in the circuit. In this table, the pipeline registers are controlled by the common clock of the circuit for the ungated clock design. As shown in Table VII, the static power is not affected much by this design decision. However, the dynamic power has improved by 47%. Since the static power is about the same in both cases, the total power has improved by 32%.

C. Sharing the Multiplication Stage

In this work, we implement the BRISK descriptor for two scales of an image. Since we are using only even rows and columns of an image and the image pixels are read at each clock cycle from the memory, the data in the pipeline of the first scale are only valid every two clock cycles. For the second scale, the data are valid every four clock cycles. We synchronize the two pipelines so that the data of the second pipeline are produced in the unused cycles of the first pipeline. We take advantage of this fact by sharing the hardware resources for the multiplication stage since it is a hardware consuming part of the pipeline. The idea is to use the same hardware resources for both scales. We use a multiplexer before the multiplication stage and according to the clock signal, we select the values from one of the pipelines. Therefore, this unit can produce valid results in three out of every four clock cycles without any conflict between the two pipelines. This will lead to a 14% less usage of lookup table (LUT) resources. Fig. 13 shows the idea of sharing the multiplication stage between two scales. Fig. 14 shows the timing of the valid data on each scale and the allocation of the multiplication stage.

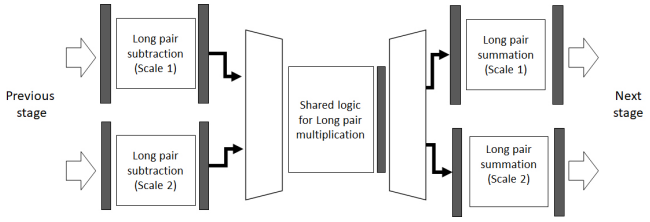

Fig. 13. Shared multiplication logic between two scales. We use a multiplexer before the shared logic to select the data from one of the pipelines and a demultiplexer after the unit to propagate results to the next stages of the pipeline.

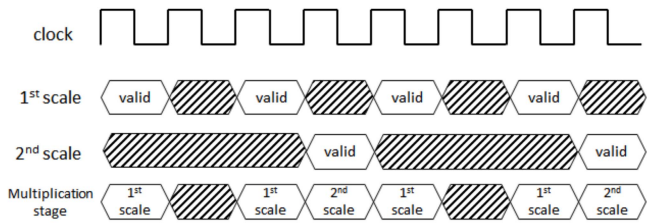

Fig. 14. Timing of the valid data on the first and second scales. The bottom waveform shows the allocation of the multiplication stage to each scale.

D. Implementation Results

We use the KCU105 FPGA board [32] which contains a Kintex[1] Ultrascale FPGA for all the experiments in this work. The resource consumption and timing information of the design are provided in this section. Table VIII shows the amount of resource usage of the proposed system with two scales on a 1920 × 1080 image resolution. We use the notations $\text{HWBRISK}_{\text{initial}}$ and $\text{HWBRISK}_{\text{final}}$ to indicate the importance of adder tree bit-width approximation and sharing the multiplication stage between two pipeline scales. Note that the $\text{HWBRISK}_{\text{initial}}$ column shows the FPGA resource usage before sharing the multiplication stage and approximating the adder tree. The $\text{HWBRISK}_{\text{final}}$ column shows the final results of our implementation. As shown in Table VIII, our implementation does not use any DSP core or block RAMs of the FPGA and all the intermediate computing results are stored in the registers. This design choice reserves FPGA resources for other computation. Multiplications and divisions are implemented on LUTs. In addition, the locations of long

[1]Registered trademark.

TABLE VIII
TOTAL RESOURCE USAGE FOR TWO SCALES OF BRISK DESCRIPTOR ON KCU105 FPGA BOARD

Resources	HWBRISK$_{initial}$		HWBRISK$_{final}$	
	Amount used	Percentage	Amount used	Percentage
LUT	182375	75%	147029	61%
LUTRAM	11400	10%	11400	10%
FF	124502	26%	189310	39%
Block RAM	0	0	0	0
DSP	0	0	0	0

pairs and short rotated pairs are stored in LUTs. The LUT usage is reduced by 14% after sharing the multiplication stage and approximating the adder tree in the long-pair summation stage. The tradeoff for using less LUTs is an increase in flip-flop utilization. Since we use extra registers for storing the data in multiplication stage, there is an increase of 13% in flip-flop numbers. Table IX illustrates the FPGA resource usage of each stage of the design. As shown in Table IX, the long-pair summation stage consumes most of the LUT resources of the FPGA. The reason is that the outputs of the long-pair multiplication stage are two channels of 870 18-bit values which enter the summation stage. These 18-bit values are added together using 10-level adder trees. We can see that by approximating the adder tree in long-pair summation, the number of LUTs have decreased from 16 389 $\times$ 2 in HWBRISK$_{\text{initial}}$ to 14 271 $\times$ 2 in HWBRISK$_{\text{final}}$. The $\times 2$ notation emphasizes that we are using two channels of summation in each pipeline. Although the number of LUTs for the long-pair multiplication stage in HWBRISK$_{\text{final}}$ (47 202) is more than that for HWBRISK$_{\text{initial}}$ (36 223) in Table IX, the total LUT resources for this unit is decreased (from 2 $\times$ 36 223 to 47 202). The value presented in the column HWBRISK$_{\text{initial}}$ is for the multiplication stage in one pipeline. On the other hand, the LUT resources shown in the HWBRISK$_{\text{final}}$ column is the amount used for the shared unit for two pipelines, which should be compared with the resource usage of two pipelines (72 446). Therefore, we have reduced the LUT usage by 35% for this stage. The maximum frequency of the design is 168 MHz which leads to 78 fps for the full HD image size that is 1920 $\times$ 1080 pixels. We report the implementation design metrics of recently published FPGA-based binary descriptors in Table X. Since the algorithms have varying types and number of computations, we cannot directly compare them based on the reported numbers. As an example, the implementation of ORB by Sun *et al.* [11] leads to a 256-bit descriptor while BRISK produces a 512-bit descriptor. In addition, for orientation estimation they process pixels in a circular patch around the key point (approximately 800 pixels). For BRISK, we select different pairs from the same patch and process about 1740 pixels. This is why resources for BRISK implementation are higher than for ORB implementation. In [33], only the scale space generation is implemented. Huang *et al.* [34] implement the BRIEF algorithm which does not use an orientation estimation stage.

In Table X, we compare the resource usage of our design with that of [30] which is an FPGA-based design of the BRISK algorithm. Ulusel *et al.* [30] analyze the implementations of BRIEF and BRISK algorithms on an embedded CPU, an FPGA, and a GPU. For the FPGA implementation, they implement the pipeline for FAST detection and BRISK description only for one scale. They also use 11 of the Block RAMs on the FPGA. In our work, we report the resource utilization for two scales, and we store the precomputed patterns on the LUTs of the FPGA. We do not use any of the Block RAM resources of the FPGA. As a result, the LUT resources reported in [30] are fewer than in our work. Ulusel *et al.* [30] do not describe the details of the orientation estimation step. Since orientation estimation is a significant component of our design and we have employed extensive parallelism in the computations of this stage, we cannot fairly compare our design with that of [30] in terms of resources. Since the focus of this work is on comparison of hardware design metrics such as power, runtime, resource utilization, and energy, they have not reported the accuracy result of their BRISK implementation.

We compare the speed metric of our design and other work in Table XI. For the same image size, our design achieves a higher frame rate and throughput and lower latency with respect to the other BRISK implementation. We measure the speed of the BRISK algorithm on the first image of the Boat set in the Oxford Affine Covariant Regions dataset [35] on a CPU implementation (Intel Core-i7, 1.3 GHz processor with 6 GB of RAM) to compare with our FPGA implementation. For CPU implementation, we use a C code version of the BRISK algorithm based on OpenCV libraries. For 1920 $\times$ 1080 pixel images, the CPU implementation can achieve 3.5 fps while our FPGA implementation achieves 78 fps, respectively. It is important to note that a larger number of key points can lead to increased processing time in the CPU implementation while the timing of our design is independent of the number of key points. This comparison indicates that the speedup in our design (HWBRISK$_{\text{final}}$) is due to our implementation rather than the BRISK algorithm itself. We also provided the results of [26] to compare with our design. Our design achieves higher frame rate with lower frequency which leads to lower dynamic power.

E. Accuracy Evaluation

In order to evaluate the correctness of our design, we tested our implementation on the Oxford Affine Covariant Regions dataset [35]. The Oxford dataset consists of a variety of image sets which have different transformations of a scene. These transformations include changes in scale, rotation, blur, light, and JPEG compression. Recent work such as [11] and [29] has used a subset of this dataset to evaluate implementation as this dataset is representative of common image transformations.

To compare our design with the original BRISK algorithm, we use a recall versus 1-precision curve. This curve demonstrates a tradeoff between recall and precision and is commonly used in descriptor evaluation literature. High precision relates to a low false-positive rate which shows the accuracy of the algorithm, and high recall relates to a low false-negative rate which shows the percentage of accepted matches over found matches. A large area under the curve indicates both high recall and high precision. Figs. 15 and 16 present the recall over 1-precision curves for matchings between the

TABLE IX
DETAIL OF RESOURCE USAGE IN VARIOUS STEPS OF THE BRISK DESCRIPTOR

Design modules	HWBRISK$_{initial}$			HWBRISK$_{final}$		
	LUT resources	LUT as Memory	LUT as Logic	LUT resources	LUT as Memory	LUT as Logic
First Line-buffer	5324	5280	44	5324	5280	44
Second Line-buffer	4148	4080	68	4148	4080	68
Third Line-buffer	2108	2040	68	2108	2040	68
Filter unit	2168	0	2168	2168	0	2168
Long-pair subtraction	6050	0	6050	6050	0	6050
Long-pair multiplication	36223	0	36223	47202**	0	47202
Long-pair summation*	16389×2	0	16389×2	14271×2	0	14271×2
Long-pair orientation	1341	0	1341	577	0	577
Short-pair comparison	6589	0	6589	6589	0	6589
Controller	375	0	375	375	0	375

*We have two channels of long-pair summation in each pipeline.

**This value is for two scales since it is shared between two pipelines.

TABLE X
SUMMARY OF DESIGN METRICS FOR RECENT FPGA IMPLEMENTATIONS OF BINARY DESCRIPTORS

Design metrics	Soleimani et al. [33]	Fang et al. [26]	Sun et al. [11]	Huang et al. [34]	Ulusel et al. [30]	HWBRISK$_{final}$
Algorithm	AKAZE	ORB	ORB	BRIEF	BRISK	BRISK
FPGA	Kintex® Ultrascale™	Altera Stratix® V	Zynq® Ultrascale+™	Kintex-7®	Zynq® 7020	Kintex® Ultrascale™
LUT	184872	25648	28168	80472	25575	158429
BRAM	18Mb	9.44Mb	1.47Mb	35kb	396Kb	0
DSP	31	8	33	0	–	0
FF	65028	21791	9528	112166	7115	189310
Image resolution	1280×720	640×480	1920×1080	512×512	800×480	1920×1080
Frequency (MHz)	100	240	200	100	111	168
Frame rate (fps)	304	67	108	310	147	78
Throughput (pixels/cc)*	2.8	0.085	1.12	0.81	0.51	0.96
Latency (cc)**	36	1166	89	123	197	104
Total Power (mW)	1095	–	873	–	2400	1060

* Throughput is the number of pixels processed per clock cycle (cc).

** Latency is the estimated number of clock cycles for 100 pixels.

TABLE XI
COMPARISON OF OUR DESIGN AND OTHER WORK IN SPEED METRIC

Reference	Algorithm	Image size	Frequency	Frame rate
Fang et al. [26]	ORB	640×480	240 MHz	67 fps
Our design	BRISK	640×480	168 MHz	516 fps
Ulusel et al. [30]	BRISK	800×480	111 MHz	147 fps
CPU implementation*	BRISK	800×480	2.3 GHz	6.5 fps
Our design	BRISK	800×480	168 MHz	413 fps
CPU implementation*	BRISK	1920×1080	2.3 GHz	3.5 fps
Our design	BRISK	1920×1080	168 MHz	78 fps

*Intel Core-i7 processor with 6GB RAM in 1.3GHz.

first (reference) and the other images of each set. In these figures, we provide the matching results of the original BRISK algorithm, our proposed hardware (HWBRISK$_{\text{final}}$) with two scales, HWBRISK$_{\text{final}}$ with three scales, and HWBRISK$_{\text{final}}$ with three scales and three subscales. Each main scale is multiplied by 2 and subscales are levels between two main scales. To have a fair comparison, we use the FAST detector for the original BRISK descriptor as well.

Fig. 15 shows the matching results of the Boat, Wall, and Graffiti image sets which have zoom, rotation, and viewpoint transformations. Fig. 16 illustrates the matching results on the Leuven, Trees, and UBC image sets which contain light, blur, and JPEG compression transformations, respectively. Fig. 17 shows the mean area under curve (AUC) of a variety of image sets. Each value in Fig. 17 is the mean of the AUCs for the five images shown in Figs. 15 and 16. A higher value of AUC relates to a larger area under the recall versus 1-precision curve, which shows better performance. For example, the AUC value of HWBRISK$_{\text{final}}$ with two scales is 0.29 on the UBC dataset which is superior to the AUC of the original BRISK algorithm which is 0.25. HWBRISK$_{\text{final}}$ shows more recall in the same precision in comparison with the original BRISK in the Leuven, Trees, and UBC image sets. As an example for the matching of image 1 to 2 of the Leuven dataset in Fig. 16, HWBRISK$_{\text{final}}$ with two scales achieves a recall of 79% for 1-precision of 0.4 while the original BRISK attains 68% at the same precision. For the Boat and Wall image sets, the matching results are comparable. The original BRISK has better matching results in the Graffiti image set.

We use an exact fixed-point MATLAB[2] simulation model of our hardware design which generates identical output as the descriptor for this test. In all cases, we use the FAST algorithm as the detector since it is very similar to AGAST,

[2]Registered trademark.

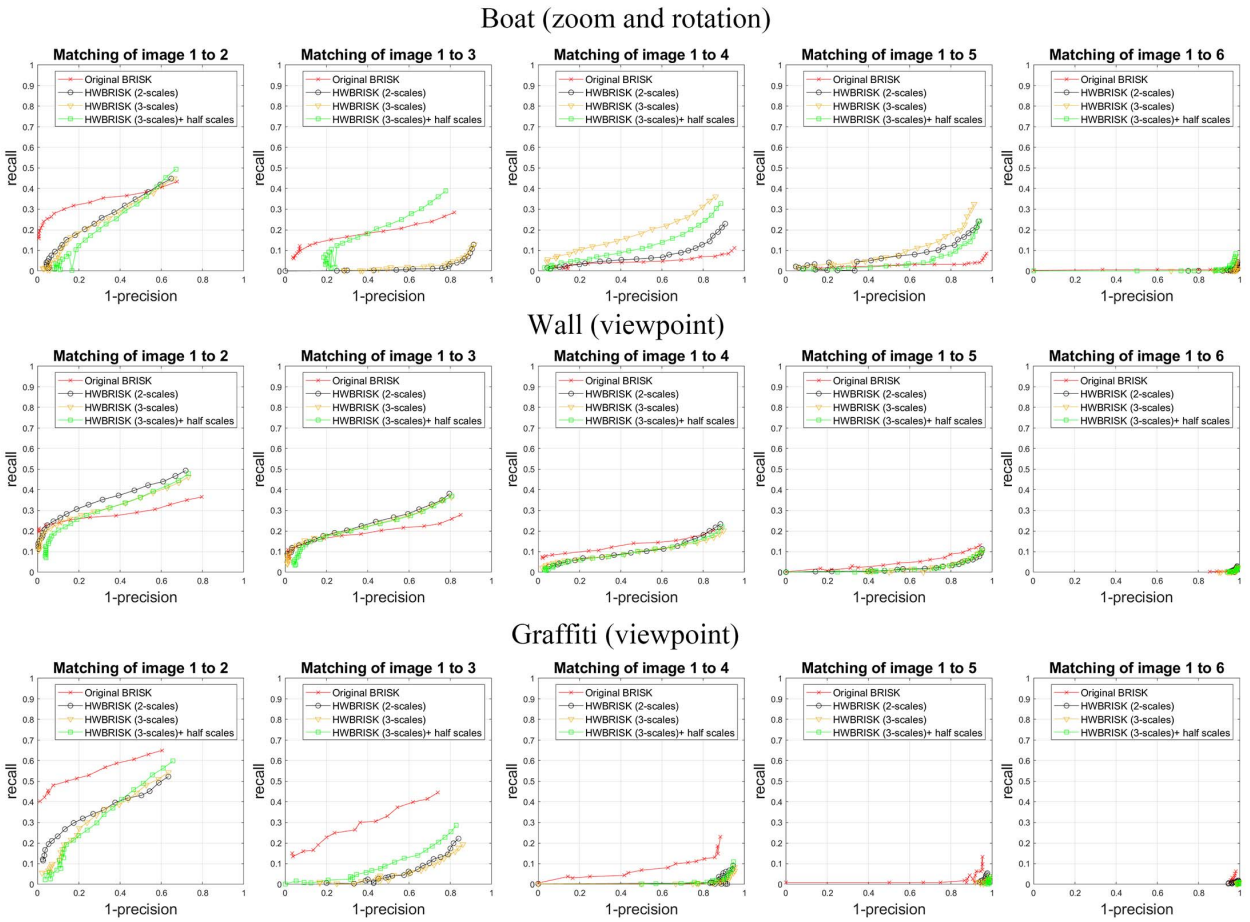

Fig. 15. Matching results using recall versus 1-precision curve for Boat, Wall, and Graffiti image sets of the Oxford Affine Covariant Regions dataset [35].

which is the detector used in the original BRISK algorithm. The only difference between them is in the decision tree which modifies the order of pixel testing to increase the speed of the detector. The results show that our proposed design is comparable in accuracy to the original BRISK algorithm. In most of the sets including Leuven, Trees, UBC, and Wall, our model achieves higher recall in the same 1-precision value while in the Graffiti image set, the original BRISK performs better. Table XII shows the direct comparison of mean AUC on Oxford imagesets for various sampling patterns. In this test, we use the same detector, filter method, and scale levels, with the only difference being the sampling pattern. $\text{HWBRISK}_{\text{final}}$ pattern achieves the highest AUC in comparison with other patterns shown in Table XII, and is comparable with the original BRISK. $\text{HWBRISK}_{\text{final}}$ achieves higher AUCs as shown in Fig. 17 since the parameters of our complete design are tailored for the proposed sampling pattern.

TABLE XII
DIRECT COMPARISON OF THE SAMPLING PATTERNS

Sampling pattern	Threshold T	Angle step n	mean AUC
Original BRISK (Fig. 1)	–	–	0.14
HWBRISK_{final} (Fig. 5 (b))	1.00	36	0.12
Pattern 1 (Fig. 5 (a))	0.98	18	0.02
Pattern 2 (Fig. 5 (c))	1.02	18	0.10
Pattern 3 (Fig. 6 (a))	1.00	24	0.09
Pattern 4 (Fig. 6 (c))	1.00	48	0.11

F. Discussion

In this section, we present a discussion on the characteristics of the proposed design for implementation of the BRISK algorithm on an FPGA. The main goal in this design is to achieve higher speed. Therefore, we designed each part of the algorithm to operate in parallel. For example, all the subtractions in long-pair calculations are computed in parallel. This design decision leads to higher resource usage which is shown in Table IX. There are two solutions to reduce resource usage as a tradeoff for speed. First, we can use shared hardware resources for each stage. As an example, we can use fewer number of subtraction modules or multipliers and perform the computations with more latency. Second, we can approximate the data between stages and reduce the bit-width of the data path. As shown in Table IX, long-pair summation consumes FPGA resources more than other parts of the system. The reason for this high resource consumption is that we implement two 10-level adder trees for which the inputs of the first level are each 18-bit. If resource consumption is critical in a specific application, we can decrease the bit-width of long-pair calculation units and approximate the computations.

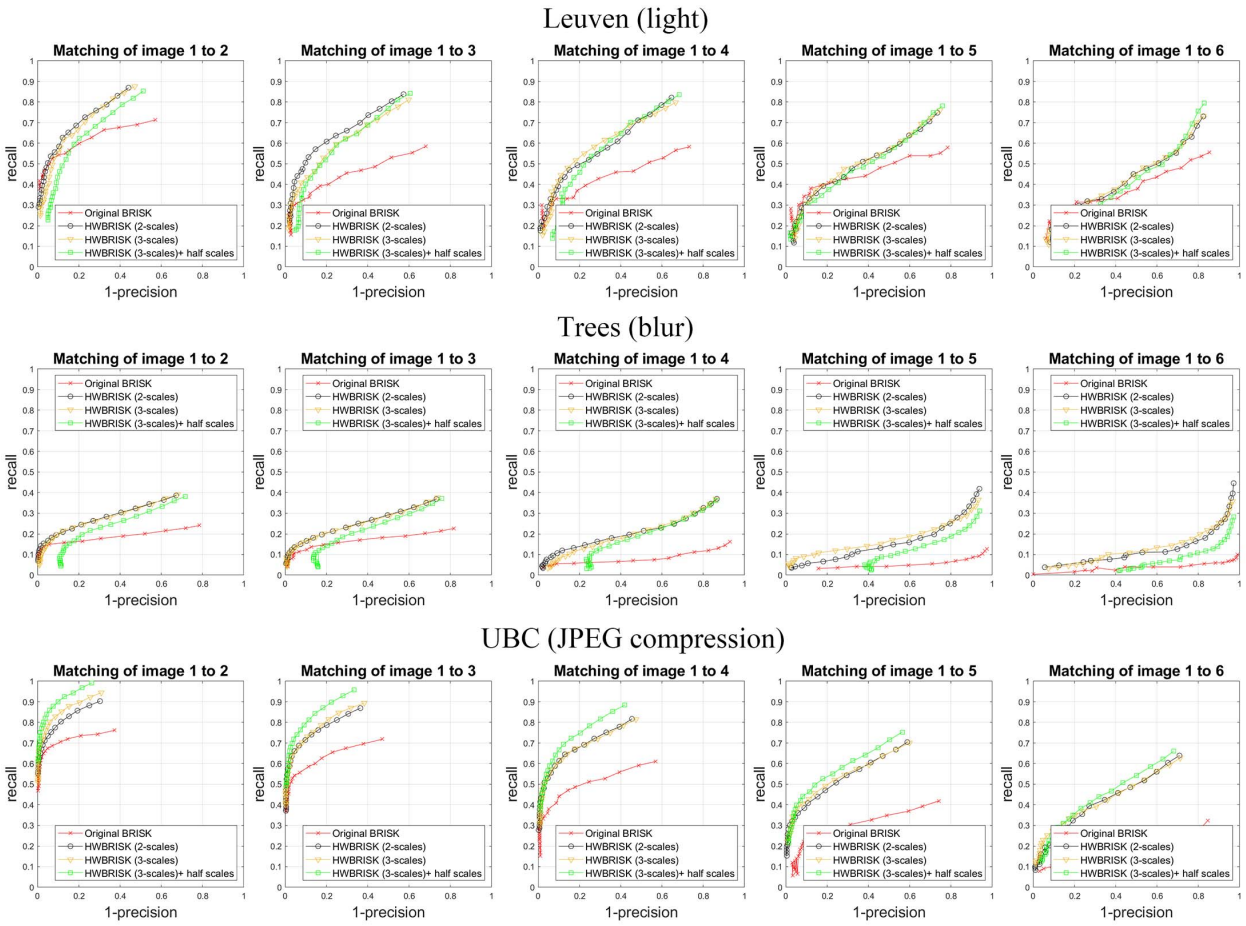

Fig. 16. Matching results using recall versus 1-precision curve for Leuven, Trees, and UBC image sets of the Oxford Affine Covariant Regions dataset [35].

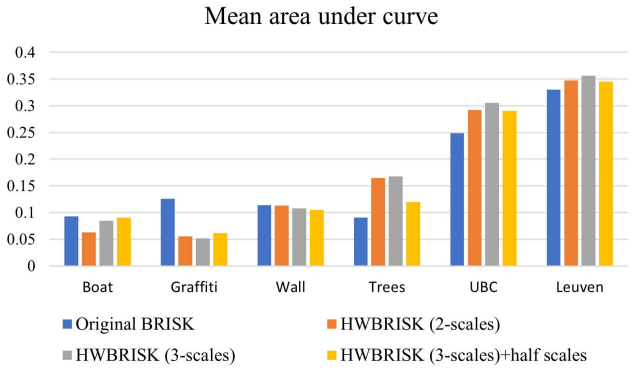

Fig. 17. Comparison of the original BRISK algorithm and $\text{HWBRISK}_{\text{final}}$ in different scales using mean AUC for images from the Oxford Affine Covariant Regions dataset [35].

Precision–recall curves shown in Figs. 15 and 16 demonstrate that $\text{HWBRISK}_{\text{final}}$ has higher accuracy than the original BRISK algorithm under image variations such as light, blur, and JPEG compression. It also has higher accuracy in most of the images in the Boat dataset which has variations in scale and rotation. However, the original BRISK performs better on the Graffiti dataset possibly due to the complexity of the Graffiti images and the more uniformly distributed pattern of the original BRISK algorithm which handles orientation estimation more precisely.

The proposed design can be used as a part of a larger vision processing system. The input image can be read from memory or received directly from a camera. Our design can be added to various key point detectors for various applications. The key point detector part can be implemented in hardware or software and it does not affect the efficiency of the descriptor part. In addition, we can store the descriptors in on-chip memory, off-chip memory, or output them using a streaming protocol depending on the application. Although we focused on the BRISK algorithm, the idea of using a hardware-aware sampling pattern that facilitates hardware implementation could be adapted to other binary descriptors as well.

VI. Conclusion

In this work, we introduced a multiscale FPGA-based implementation of the BRISK descriptor. We presented a new sampling pattern for the BRISK algorithm with a similar number of sampling points, which reduced the number of registers for line buffers by more than 50% and the pipeline registers up to 73%. The proposed design is fully pipelined and achieves a maximum operating frequency of 168 MHz. For images with

1920 × 1080 resolution, our proposed implementation has a frame rate of 78 fps.

There are multiple potential enhancements that can be addressed in the future of this research. First, we can implement a gated filter unit after the key point detection as the next stage of the pipeline to reduce power consumption. Second, we can replace the FAST detector with a more complex detector. Since key point detection is an early stage of an image matching system, choosing an appropriate detector can improve the accuracy depending on the application. Finally, since the image may be already loaded on on-chip memory in many applications, designing a nonstreaming input architecture for the BRISK algorithm is a potential future path for this research.

REFERENCES

[1] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, Feb. 2004.

[2] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded up robust features," in *Computer Vision—ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Germany: Springer, 2006, pp. 404–417.

[3] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 1, no. 1, Jun. 2005, pp. 886–893.

[4] M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, and P. Fua, "BRIEF: Computing a local binary descriptor very fast," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 7, pp. 1281–1298, Jul. 2012.

[5] A. Alahi, R. Ortiz, and P. Vandergheynst, "FREAK: Fast retina keypoint," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2012, pp. 510–517.

[6] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary robust invariant scalable keypoints," in *Proc. Int. Conf. Comput. Vis.*, Nov. 2011, pp. 2548–2555.

[7] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *Proc. Int. Conf. Comput. Vis.*, Nov. 2011, pp. 2564–2571.

[8] S. Ghaffari, P. Soleimani, K. F. Li, and D. W. Capson, "Analysis and comparison of FPGA-based histogram of oriented gradients implementations," *IEEE Access*, vol. 8, pp. 79920–79934, 2020.

[9] E. Mair, G. D. Hager, D. Burschka, M. Suppa, and G. Hirzinger, "Adaptive and generic corner detection based on the accelerated segment test," in *Computer Vision—ECCV 2010*, K. Daniilidis, P. Maragos, and N. Paragios, Eds. Berlin, Germany: Springer, 2010, pp. 183–196.

[10] R. Liu, J. Yang, Y. Chen, and W. Zhao, "ESLAM: An energy-efficient accelerator for real-time ORB-SLAM on FPGA platform," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.

[11] R. Sun *et al.*, "A flexible and efficient real-time ORB-based full-HD image feature extraction accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 565–575, Feb. 2020.

[12] M. Kashif, T. M. Deserno, D. Haak, and S. Jonas, "Feature description with SIFT, SURF, BRIEF, BRISK, or FREAK? A general question answered for bone age assessment," *Comput. Biol. Med.*, vol. 68, pp. 67–75, Jan. 2016.

[13] C. Belloni, N. Aouf, J.-M.-L. Caillec, and T. Merlet, "Comparison of descriptors for SAR ATR," in *Proc. IEEE Radar Conf. (RadarConf)*, Apr. 2019, pp. 1–6.

[14] U. Sharif, Z. Mehmood, T. Mahmood, M. A. Javid, A. Rehman, and T. Saba, "Scene analysis and search using local features and support vector machine for effective content-based image retrieval," *Artif. Intell. Rev.*, vol. 52, no. 2, pp. 901–925, Aug. 2019.

[15] L. Zhang, K. Mistry, M. Jiang, S. C. Neoh, and M. A. Hossain, "Adaptive facial point detection and emotion recognition for a humanoid robot," *Comput. Vis. Image Understand.*, vol. 140, pp. 93–114, Nov. 2015.

[16] D. Bekele, M. Teutsch, and T. Schuchert, "Evaluation of binary keypoint descriptors," in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2013, pp. 3652–3656.

[17] T. Mouats, N. Aouf, D. Nam, and S. Vidas, "Performance evaluation of feature detectors and descriptors beyond the visible," *J. Intell. Robotic Syst.*, vol. 92, no. 1, pp. 33–63, Sep. 2018.

[18] M. Agrawal, K. Konolige, and M. R. Blas, "CenSurE: Center surround extremas for realtime feature detection and matching," in *Computer Vision—ECCV 2008*, D. Forsyth, P. Torr, and A. Zisserman, Eds. Berlin, Germany: Springer, 2008, pp. 102–115.

[19] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," in *Proc. 10th IEEE Int. Conf. Comput. Vis. (ICCV)*, vol. 1, Oct. 2005, pp. 1508–1515.

[20] V. Balntas, K. Lenc, A. Vedaldi, T. Tuytelaars, J. Matas, and K. Mikolajczyk, "H-patches: A benchmark and evaluation of handcrafted and learned local descriptors," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 11, pp. 2825–2841, May 2020.

[21] Y. Dong, D. Fan, Q. Ma, and S. Ji, "Superpixel-based local features for image matching," *IEEE Access*, vol. 9, pp. 15467–15484, 2021.

[22] S. Madeo and M. Bober, "Fast, compact, and discriminative: Evaluation of binary descriptors for mobile applications," *IEEE Trans. Multimedia*, vol. 19, no. 2, pp. 221–235, Feb. 2017.

[23] R. Sun, P. Liu, J. Wang, C. Accetti, and A. A. Naqvi, "A 42fps full-HD ORB feature extraction accelerator with reduced memory overhead," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, Dec. 2017, pp. 183–190.

[24] R. de Lima, J. Martinez-Carranza, A. Morales-Reyes, and R. Cumplido, "Improving the construction of ORB through FPGA-based acceleration," *Mach. Vis. Appl.*, vol. 28, nos. 5–6, pp. 525–537, Aug. 2017.

[25] P. Tran, T. H. Pham, S. K. Lam, M. Wu, and B. A. Jasani, "Stream-based ORB feature extractor with dynamic power optimization," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 94–101.

[26] W. Fang, Y. Zhang, B. Yu, and S. Liu, "FPGA-based ORB feature extraction for real-time visual SLAM," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, Dec. 2017, pp. 275–278.

[27] L. Kalms, M. Hajduk, and D. Gohringer, "Efficient pattern recognition algorithm including a fast retina keypoint FPGA implementation," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 121–128.

[28] R. Kapela, K. Gugala, P. Sniatala, A. Swietlicka, and K. Kolanowski, "Embedded platform for local image descriptor based object detection," *Appl. Math. Comput.*, vol. 267, pp. 419–426, Sep. 2015.

[29] T. H. Pham, P. Tran, and S.-K. Lam, "High-throughput and area-optimized architecture for rBRIEF feature extraction," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 4, pp. 747–756, Apr. 2019.

[30] O. Ulusel, C. Picardo, C. B. Harris, S. Reda, and R. I. Bahar, "Hardware acceleration of feature detection and description algorithms on low-power embedded platforms," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–9.

[31] E. Azimi, A. Behrad, M. B. Ghaznavi-Ghoushchi, and J. Shanbehzadeh, "A fully pipelined and parallel hardware architecture for real-time BRISK salient point extraction," *J. Real-Time Image Process.*, vol. 16, no. 5, pp. 1859–1879, Oct. 2019.

[32] *KCU105 Board User Guide (v1.10) (UG917)*, Xilinx, San Jose, CA, USA, 2019. [Online]. Available: http://www.xilinx.com

[33] P. Soleimani, D. W. Capson, and K. F. Li, "Real-time FPGA-based implementation of the AKAZE algorithm with nonlinear scale space generation using image partitioning," *J. Real-Time Image Process.*, vol. 18, no. 6, pp. 2123–2134, Dec. 2021.

[34] J. Huang, G. Zhou, X. Zhou, and R. Zhang, "A new FPGA architecture of FAST and BRIEF algorithm for on-board corner detection and matching," *Sensors*, vol. 18, no. 4, p. 1014, Mar. 2018.

[35] K. Mikolajczyk *et al.*, "A comparison of affine region detectors," *Int. J. Comput. Vis.*, vol. 65, no. 1, pp. 43–72, 2005.