High-Performance SIFT Hardware Accelerator for Real-Time Image Feature Extraction

Feng-Cheng Huang, Shi-Yu Huang, *Member, IEEE,* Ji-Wei Ker, and Yung-Chang Chen, *Fellow, IEEE*

***Abstract*—Feature extraction is an essential part in applications that require computer vision to recognize objects in an image processed. To extract the features robustly, feature extraction algorithms are often very demanding in computation so that the performance achieved by pure software is far from real-time. Among those feature extraction algorithms, scale-invariant feature transform (SIFT) has gained a lot of popularity recently. In this paper, we propose an all-hardware SIFT accelerator—the fastest of its kind to our knowledge. It consists of two interactive hardware components, one for key point identification, and the other for feature descriptor generation. We successfully developed a segment buffer scheme that could not only feed data to the computing modules in a data-streaming manner, but also reduce about 50% memory requirement than a previous work. With a parallel architecture incorporating a three-stage pipeline, the processing time of the key point identification is only 3.4 ms for one video graphics array (VGA) image. Taking also into account the feature descriptor generation part, the overall SIFT processing time for a VGA image can be kept within 33 ms (to support real-time operation) when the number of feature points to be extracted is fewer than 890.**

***Index Terms*—Feature extraction, hardware accelerator, object recognition, real-time, rotating SRAM banks, scale-invariant feature transform (SIFT).**

I. Introduction

SINCE THE Moravec operator [1], [2] was proposed, the feature detection has been an interesting problem in computer vision field. Moravec defined those points in an image that have large intensity variation in every direction as key points. These key points represent the distinctive regions and the features of the objects contained in the image being processed.

One of the first key points detection algorithms which has been widely used is the Harris corner detector [3]. Harris and Stephens developed this detector by addressing the limitations of the Moravec operator, and thereby achieving great improvement in terms of noise immunity and invariance to intensity change and rotation.

Several other feature detectors have also been proposed. Among them, Lowe proposed the scale-invariant feature transform (SIFT) algorithm [4] which has been considered as one of the most robust approaches. The SIFT features are invariant to: 1) scale; 2) rotation; and 3) illumination, which are essential in many applications like robot navigation [5], image stitching for panoramic photographs [6], stereo matching, image registration, object detection, and recognition [7], [8].

However, the SIFT algorithm has a high computational cost and memory requirement, which prevents it from real-time applications if implemented by pure software.

Speed-up feature detectors such as *SURF* [9] and *fast SIFT* [10] have thus been proposed. They attempted to reduce the computational time by sacrificing the quality of features extracted. Even so, the improvement achieved on processing speed may not be great enough for real-time applications. Graphic processing unit (GPU) based systems were also proposed to achieve near real-time feature detection. In [11], a GPU-based system for 3-D object recognition using SIFT was proposed. Nvidia-GeForce 8500GT GPU is used to speed up about three times as compared with the Intel Pentium 2.2 GHz CPU. In [12], another GPU-based system was proposed to detect eye blinking with SIFT—with the processing speed reaching about 25 frames/s. There is a downside of these GPU based systems—they often require excessive hardware resources and consume too much power and thus may not be very suitable for an embedded system used in a portable device.

In an application to calibrate the stereo images with an online approach, two hardware modules were proposed to perform SIFT on field-programmable gate array (FPGA) boards [13], [14]. To reduce the complexity, only the feature detection part is realized by hardware, whereas the feature descriptor generation is done by software. Overall, these approaches marked the first attempts to accelerate SIFT operation by hardware.

There are several other SIFT hardware designs proposed in the literature based on the system on programmable chip (SOPC) approach [15], [16]. In [17], an object recognition system is demonstrated with a feature detection time of about 62 ms for QVGA images. In [18], a stereo vision system for the simultaneous localization and mapping (SLAM) operation used in a robot incorporates the SIFT algorithm to build a map. A customized SIFT algorithm was implemented on FPGA to

Manuscript received February 21, 2011; revised April 21, 2011; accepted June 8, 2011. Date of publication July 22, 2011; date of current version March 7, 2012. This work was supported in part by Novatek Microelectronics Corporation, Hsinchu, Taiwan. This paper was recommended by Associate Editor G. Lafruit.

F.-C. Huang and J.-W. Ker are with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: fchuang@larc.ee.nthu.edu.tw; jwker@larc.ee.nthu.edu.tw).

S.-Y. Huang and Y.-C. Chen are with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: syhuang@ee.nthu.edu.tw; ycchen@ee.nthu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSVT.2011.2162760

TABLE I
COMPLEXITY AND PERFORMANCE OF TWO MOST RECENT WORKS ON THE SIFT HARDWARE DESIGN

	V. Bonato, TCSVT'08	L. Yao, ICFPT'09
Device	Xilinx Virtex-5	Altera Stratix II
LUTs	35 889	43 366
Registers	19 529	19 100
Block RAM	3240 kb	1350 kb
DSP	97 DSP elements	64 DSP blocks
External memory	256K×32 bits SRAM and SDRAM	16M bits SDRAM
Feature detection time per frame	31 ms/per VGA frame	33 ms/per QVGA frame

TABLE II
RUN TIME ANALYSIS OF A SIFT SOFTWARE PROGRAM FOR A VGA ''BEAVER'' IMAGE ON INTEL CORE 2 DUO 2.09 GHZ CPU

Steps in Algorithm	Run Time (s)	Percentage
Gaussian pyramid construction	2.1	72.85%
DoG space construction	0.004	0.14%
Key-points detection	0.16	5.71%
Gradient and orientation assignment	0.08	2.85%
Descriptors generation	0.52	18.57%
Entire SIFT operation	2.87	100%

TABLE III
RUN TIME ANALYSIS OF A SIFT SOFTWARE PROGRAM FOR A VGA ''BEAVER'' IMAGE ON ALTERA 100 MHZ NIOSII CPU ON DE2-70 FPGA BOARD

Steps in Algorithm	Run Time (s)	Percentage
Gaussian pyramid construction	2278.27	85.63%
DoG space construction	3.62	0.11%
Key-points detection	9.08	0.33%
Gradient and orientation assignment	57.56	2.16%
Descriptors generation	311.80	11.69%
Entire SIFT operation	2660.66	100%

provide a performance of about 60 ms per video graphics array (VGA) image frame (with 640 × 480 pixels).

A detailed hardware architecture of SIFT was ever proposed in [19], which incorporates both hardware and software in an FPGA system for vision SLAM. The hardware implementation part can detect the SIFT features with a performance of 30 QVGA frames/s. Furthermore, the computation of the local minima in the key-point detection stage is skipped for reducing the hardware amount and for speeding up the process. Such a simplification could potentially overlook some important key points and thus may not be very suitable for some other applications. Furthermore, the second part of the SIFT operation—the feature descriptor generation—may still form the bottleneck of the overall system. Another SOPC approach is to implement the image matcher with optimized SIFT in [20]. The system has the ability to detect the features in a VGA image within 31 ms. However, their optimized algorithm simplifies the stability checking of the key points by scaling down the difference-of-Gaussian (DoG) pixels with only integer values, which may again sacrifice the accuracy and lead to misdetection of some features.

In general, we target to compare our work with the systems proposed in [19] and [20], considering that they are more complete and recent. As shown in Table I, we list several important aspects of a SIFT hardware system. It is worth mentioning that only the performance of key-point identification part has been reported in these two works. But in our work, we will perform the entire SIFT operation including the feature descriptor generation part.

Table II shows the run time analysis of executing an SIFT software program on a 2.1 GHz Intel CPU. Overall, it takes about 2.87 s to complete the SIFT operation of a VGA image, with 72.85% of the time spent on the "Gaussian pyramid construction" and 18.57% on the final "descriptors generation." Table III shows the run time analysis of executing an SIFT software program on an embedded system using DE2-70 FPGA board, featuring a soft-core of 100 MHz 32 bit Nios II CPU and a 32 bit Avalon bus. Overall, the run time skyrockets to 2660 s in this embedded system—which is 926 times that of a high-performance desktop PC. These run time analyses clearly show that a hardware accelerator is almost inevitable if the SIFT algorithm is to be performed in an embedded system. The significant performance difference between software and custom hardware is mostly due to the following reasons. First, the parallelism level in a SIFT algorithm is very high and we could exploit it by using many concurrent data-path units in a SIFT hardware accelerator. In a CPU, the number of instructions that can be executed simultaneously is often much smaller than in custom hardware. Second, the bandwidth between the data path units and the memory can be tremendously enhanced with the aid of some custom memory buffer equipped with many input/output ports. The last but not the least, we can use fixed-point number in hardware accelerator to facilitate faster execution than the floating-point number in a CPU.

As mentioned above, the most time-consuming step in a SIFT algorithm is the Gaussian filtering, and the second is the feature descriptor generation. Unlike most previous works that focused mostly on the first bottleneck, our work will also accelerate the second performance bottleneck, i.e., the feature descriptor generation, so as to make the entire SIFT operation efficient and applicable to images with even higher resolutions. What we will propose in this paper can be considered as an all-hardware high-performance SIFT design with pipelined architecture. One unique aspect of this design is a memory scheme that can not only reduce the hardware requirement by 50% but also simultaneously meet the high-speed data bandwidth in a data-streaming manner required by the computational modules.

The rest of this paper is organized as follows. Section II reviews the overall SIFT algorithm proposed by Lowe in 1999 [4], [7]. Section III presents the architecture of our SIFT hardware accelerator. Section IV shows the simulation results, and Section V concludes this paper.

II. Underlying SIFT Algorithm

This section describes the SIFT algorithm based on [4]. The input is a source image. After the SIFT processing, a number of feature descriptors are reported, with each feature descriptor having 132 values (to be detailed later). Sometimes, such a feature descriptor is called a 132-D vector as well.

The algorithm consists of four basic stages: 1) to build the Gaussian-filtered images and DoG images; 2) to identify and adjust the key-points. Only those robust ones will survive after a process called stability checking; 3) to calculate the principle orientation of each key-point; and finally 4) to generate the feature descriptor according to a specified format to represent each key-point identified.

A. Gaussian Pyramid and DoG Pyramid

To identify the key points, the first step is to build up a Gaussian pyramid. First, we convolve the given source image, denoted as I(x, y), with a Gaussian function G(x, y, σ), where x and y denote the horizontal and vertical spatial coordinates, respectively, and σ is a variable scaling factor. The result is a Gaussian-filtered image, denoted as L(x, y, σ), that is

$$L(x,\ y,\ \sigma) = G(x,\ y,\ \sigma) * I(x,\ y) \tag{1}$$

where

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)}. \tag{2}$$

A DoG image, denoted as D(x, y, σ), is defined as the difference of two adjacent Gaussian-filtered images, for example

$$D(x,\ y,\ \sigma) = L(x,\ y,\ k\,\sigma) - L(x,\ y,\ \sigma)$$

where k is a multiplicative factor, $k = 2^{\frac{1}{3}}$ which is a suggested value by [4].

The process to construct the DoG images, D(x, y, σ), can be illustrated as Fig. 1. There are three so-called *octaves*, each having a group of six Gaussian-filtered images. For example, in the first octave, there are six Gaussian-filtered images, namely $\{L(k_0), L(k_1), \ldots, L(k_5)\}$, where $L(k_i)$ is the Gaussian-filtered image obtained by the given source image I(x, y) convolved with $G(k_i)$ which is G(x, y, $k^1\sigma$). In other words, the standard deviations of the Gaussian filters to be implemented include $\{\sigma, k\sigma, k^2\sigma, k^3\sigma, k^4\sigma, k^5\sigma\}$, where σ is 1.6 and k is a constant factor of $2^{1/3}$. The Gaussian filter is associated with a mask size (which decides the size of the 2-D window operation). The mask size is often taken as five times the standard deviation [4]. In other words, we need to handle Gaussian filters with different mask sizes in $\{9 \times 9$, 11×11, 13×13, 15×15, 21×21, and $25 \times 25\}$.

The second octave is basically the down-sampled version of the first octave, and the third octave is the down-sampled version of the second octave. Since there are five adjacent Gaussian-filtered image pairs in each octave, we will generate five DoG images in each octave, leading to overall $5 * 3 = 15$ DoG images in the final DoG pyramid.

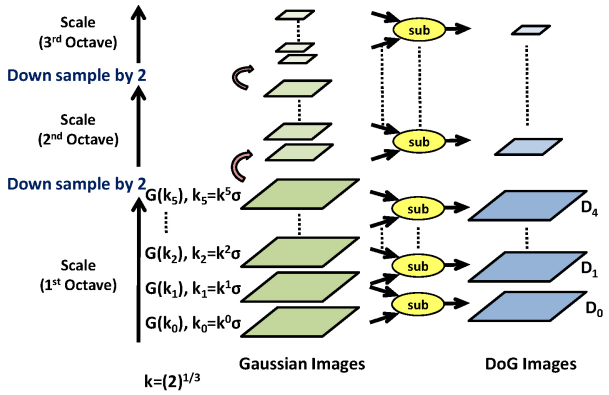

Fig. 1. Gaussian pyramid and difference-of-Gaussian (DoG) pyramid. There are three octaves, each having six Gaussian images.

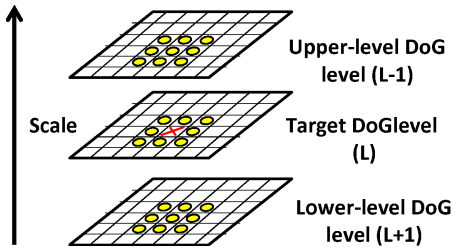

Fig. 2. Key-point detection: to detect the maxima and minima of the difference-of-Gaussian (DoG) images as the key-point candidates, each pixel (marked with X) is compared to its 26 neighbors (marked with circles).

B. Key-Point Detection and Stability Checking

The key-points are detected by analyzing the DoG images, i.e., by finding the local maxima or the local minima. For every pixel in a DoG image, it is compared to its eight surrounding neighbors in the same DoG image, and the nine surrounding neighbors in its upper-level DoG image, and the nine surrounding neighbors in its lower-level DoG image, as shown in Fig. 2. The pixel is identified as a key-point candidate if it is the maximum or the minimum out of the total 26 neighboring pixels.

Each key-point candidate will have to pass a subsequent stability checking procedure in order to become a true key point. During this process, the candidates with relatively low contrasts (considered as flat points) are rejected, and the candidates located on the edges (considered as not distinct) are also eliminated as well.

C. Gradient Histogram of Orientation

For each key-point, we need to calculate its principle orientation. First, a gradient histogram of orientation is computed from the neighborhood of the key-point in the corresponding Gaussian-filtered image.

Definition 1 (KP-Region): Throughout this paper, we refer to the neighborhood region around a key-point in the Gaussian-filtered image (not the DoG image) as a key-point region, abbreviated as KP-region. The size of this region varies with the scaling factor it is related to.

The gradient of a pixel located at (x, y) is represented by two parts—the part in horizontal direction (Gx), and the

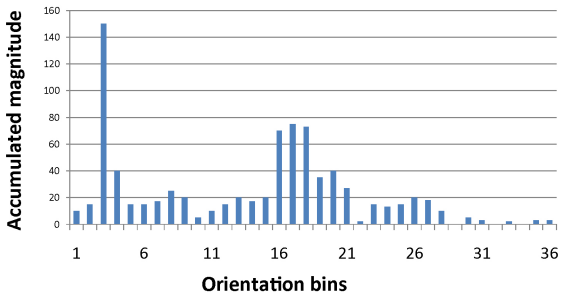

Fig. 3. Example of "gradient histogram of orientation" calculated from pixels within a key-point (KP) region on a Gaussian filtered image.

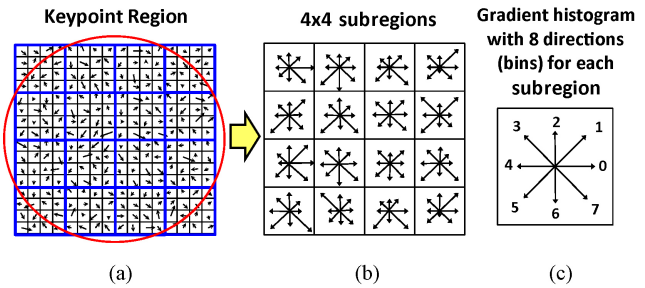

Fig. 4. Generation of feature descriptor. (a) KP-region around a key point. (b) 4x4 subregions within a KP-region. (c) Gradient histogram of one subregion.

part in vertical direction (Gy), calculated by the following expressions:

$$\mathrm{Gx} = \mathrm{L}(\mathrm{x}+1,\ \mathrm{y}) - \mathrm{L}(\mathrm{x}-1,\ \mathrm{y}),\ \text{and}$$
$$\mathrm{Gy} = \mathrm{L}(\mathrm{x},\ \mathrm{y}+1) - \mathrm{L}(\mathrm{x},\ \mathrm{y}-1).$$

For each pixel inside a KP-region, we compute its gradient magnitude and orientation by (3) and (4) that follows, where $mag(\mathrm{x}, \mathrm{y})$ is the gradient magnitude and $\theta(\mathrm{x}, \mathrm{y})$ is the orientation

$$mag(\mathrm{x},\ \mathrm{y}) = (\mathrm{Gx}^2 + \mathrm{Gy}^2)^{\frac{1}{2}} \tag{3}$$

$$\theta(\mathrm{x},\ \mathrm{y}) = \tan^{-1}(\mathrm{Gy}/\mathrm{Gx}). \tag{4}$$

Next, we divide the overall 360° of a circle into 36 bins, each covering 10°. Then, we accumulate the gradient magnitudes within each bin, leading to a so-called gradient histogram of orientation, as shown in Fig. 3. The peak in the gradient histogram is the principle orientation of the key point—which will be called key-point orientation (or KP orientation for short) in the sequel.

D. Feature Descriptor Generation

This stage is to transfer the detected key points and their neighboring pixels into specified feature descriptors. Taking a key point as the center, its KP-region is divided into $4 \times 4 = 16$ square subregions on the Gaussian-filtered image hosting the target key-point, as illustrated in Fig. 4(a) and (b). The gradient histogram of orientation (defined previously) is computed for each subregion, and each histogram now has eight orientation bins as shown in Fig. 4(c). In other words, each bin covers 45°. There is a subtle detail—the gradient histograms of orientation are weighted by a Gaussian function as specified in Lowe's algorithm [4]. To achieve rotation invariance, the pixels within each subregion are further rotated with the key-point orientation as we have computed previously. Overall, these 16 histograms will be represented by $(16) \times 8 = 128$ values.

III. Proposed Hardware Architecture

The overall architecture of the proposed SIFT hardware is shown in Fig. 5. It consists of two major components: 1) the key-point identification part, and 2) the descriptor generation part. In the first part, there are three stages: 1) Gaussian filtering and DoG image computation; 2) key-point detection; and 3) gradient magnitude and orientation computation. The stage 1 is designed for building the Gaussian-filtered images and difference-of-Gaussian images, these images are the input of stage 2 and stage 3. Stage 2 receives DoG images to identify the key points with stability checking. The stage 3 computes the gradient magnitude and orientation of each pixel in the Gaussian images. The second part—feature descriptor generation part—receives the locations (including x and y coordinates) of the key points from stage 2 and the related gradient histograms from stage 3. It will perform key-point orientation assignment and then the feature descriptor generation.

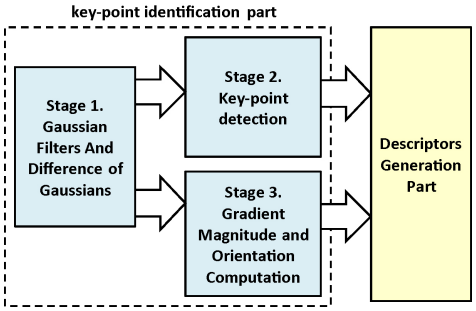

Fig. 5. Overall architecture of the proposed SIFT hardware.

A. Overall Hardware Architecture

In general, we treat the two components as two interactive design blocks, with the first block designated as the main processor, and the second block as a co-processor, as shown in Fig. 6. The main processor reads in the source image and processes the key-points identification. Every time when it identifies a key point, it invokes the co-processor to start the feature descriptor generation for the identified key point. After the co-processor has completed the feature descriptor generation for one key point, the main processor resumes identifying the next key point. The interaction continues until the entire source image has been analyzed.

Fig. 7 shows the processing flow of the SIFT operation, taking a beaver image as an example. There is one objective we wish to achieve in this architecture—we will scan the source image only once and keep the size of the internal memory buffers as small as possible, which leads us to a segment-based processing flow described below.

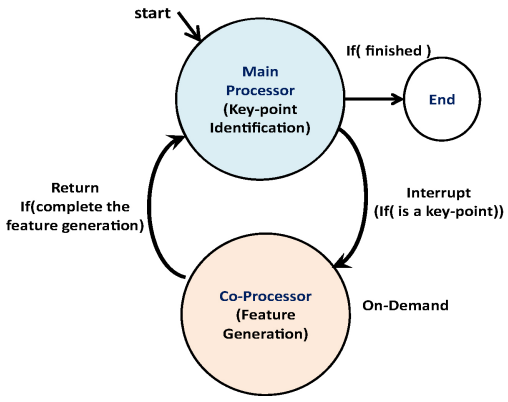

Fig. 6. State transition diagram of two interactive components.

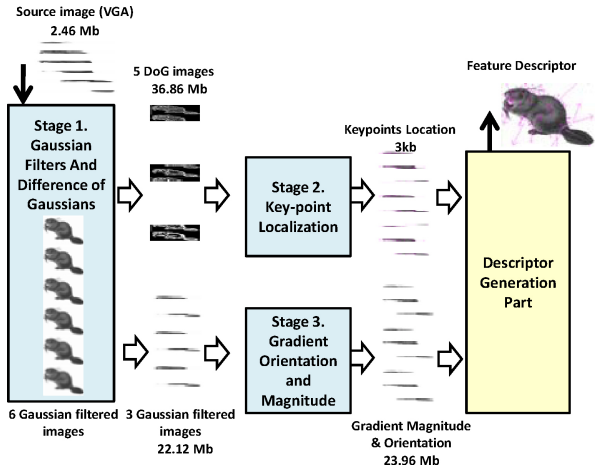

Fig. 7. Processing flow and intermediate images in the SIFT operation for one octave. (Note: there are three octaves in total).

B. Segment-Based Processing Flow

Definition 2 (Segment): A contiguous block of a source image, or a Gaussian-filtered image, or a DoG image can all be referred to as a segment (buffer) in this paper. A segment consists of several contiguous rows of pixels. There are many segment buffers in our design with the number of rows varying from as small as 3 to as large as 69, depending on its usage. The number of pixels inside a row also varies. Fig. 8 shows the overall segment buffers needed in the SIFT processing flow for one octave.

1) *Source Image Buffer and Segment Based Operation:* In general, the source image is processed on a segment basis. For example, we use a source image buffer of 27 rows by 640 columns. Upon such a source image buffer, the operations are classified into three basic operations as illustrated in Fig. 9: 1) OP1 (named DoG for short, but indeed it includes the Gaussian filtering and DoG computation); 2) OP2 (named KPD representing key-point detection); and 3) OP3 (named GMO representing the gradient magnitude and orientation computation). Once a source image buffer has been processed, we replenish it from the external memory where the entire source image is located, and then start another round of segment-based operation.

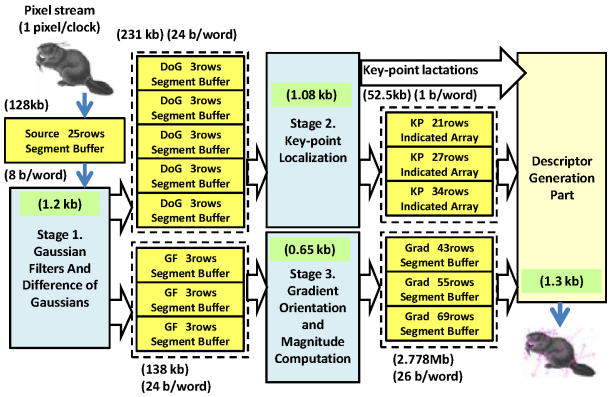

Fig. 8. Block diagram of SIFT computational modules and segment buffers needed for one octave.

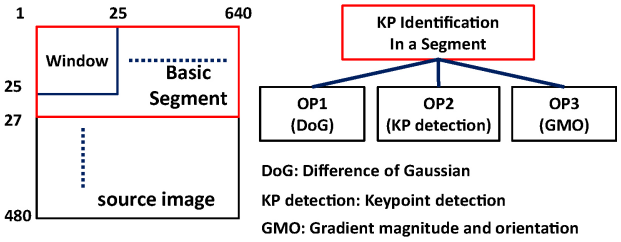

Fig. 9. Computation of key-point identification for one basic segment of 27×640.

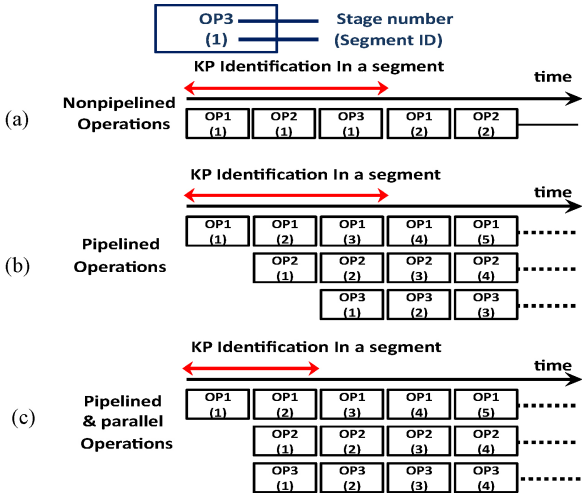

Fig. 10. Non-pipelined versus pipelined performance. (a) Non-pipelined three stages. (b) Pipelined three stages. (c) Pipelined three stages with parallel processing.

2) *Segment Based Pipeline:* The three OPs can be arranged with a three-stage pipeline processing, as shown in Fig. 10. This pipelined architecture increase the throughput from one result every three clock cycles to one result every clock cycle, implying a speedup of three times. Furthermore, since the data being processed at any given time by OP2 and OP3 are independent, they can be done in parallel, which cuts the input-to-output latency by one, as shown in Fig. 10(c).

3) *Inter-Stage Segment Buffer:* Between two processing stages, we may need segment buffers. In general, the data to be processed by OP1, OP2, and OP3 are often defined by a

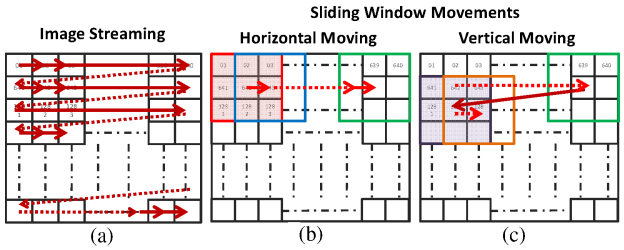

Fig. 11. Image streaming and sliding window movement. (a) Scanning path of an image. (b) Sliding window movement in horizontal direction. (c) Sliding window movement in vertical direction.

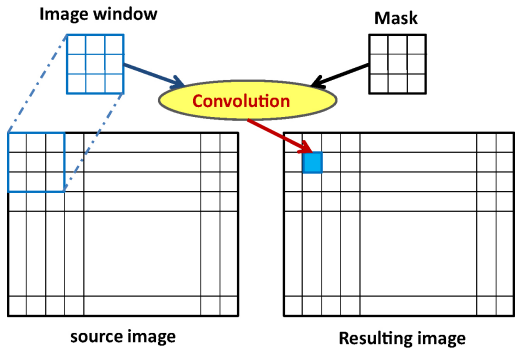

Fig. 12. Sliding window operation to support convolution between a window of source image and a specified mask.

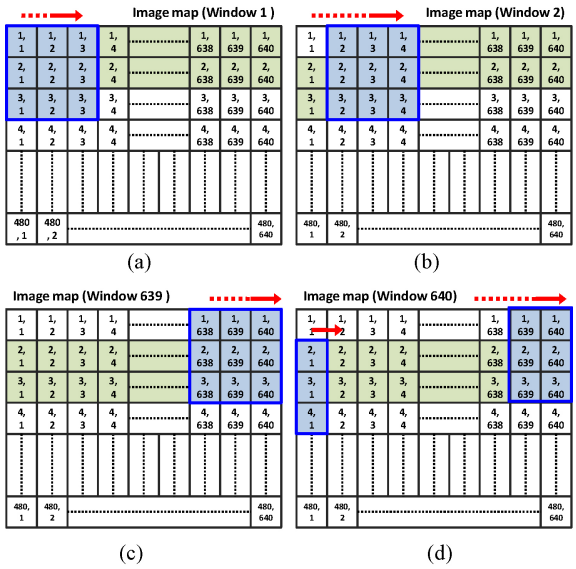

Fig. 13. 640×480 image with a 3×3 sliding window. (a) First 3×3 window in the image map. (b) Move to the second 3×3 window from (a) in horizontal direction. (c) Last window of the first 640×3 image segment. (d) Movement in vertical direction.

window which slides to the right in a row major order. The window size could be 3×3 as required by non-maximum suppression window or 25×25 as required by the largest Gaussian kernel. We need to support a quick sliding window moving scheme and thereby providing all the data inside the sliding window to the computing modules that needs the data (including OP1, OP2, and OP3). This will be detailed later.

Taking the Gaussian filtering performed by OP1 for example, the window size is sometimes 25×25, which means that 25 pixels in one column are needed simultaneously as the input each time when the window moves horizontally by 1 pixel. As a result, a 25-row segment buffer needs to be inserted before the hardware performing OP1, as also shown in Fig. 9.

Similarly, the detection of the extrema (i.e., the minima or the maxima) in OP2 operates on data in a 3×3 window in five potential DoG images produced by OP1. Consequently, we need five 3-row segment buffers between OP1 and OP2. For OP3, the input data is a set of three Gaussian-filtered images produced by OP1. We thus need three 3-row segment buffers between OP1 and OP3. Next, we will describe the implementation techniques of these segment buffers that support the sliding window operation efficiently.

C. Segment Buffer Supporting Sliding Window Operation

Fig. 11 shows the image streaming and the sliding window movement. The segment buffer is scanned from left to right and top to bottom in an order as shown in Fig. 11(a). Ideally, the pixels are shifted into our SIFT hardware serially in a streaming manner. Then, we start the sliding window operation—in which data defined in the current window area are fed to the corresponding computing module and then the window move horizontally until it hits the rightmost end of the segment buffer, as demonstrated in Fig. 11(b). After that, the window returns to the leftmost-end and moves down vertically by one row to start another round of horizontal scanning, as shown in Fig. 11(c).

1) *Foundation of Window-Based Processing:* In general, the sliding window operation involves a large amount of computations on massive amount of data. For instance, the 2-D convolution [21], [22] is one of the typical sliding window operations, as illustrated in Fig. 12. First of all, there is a 3×3 mask with one coefficient at each square. The mask size and the coefficients are decided by the target operator, such as the Gaussian filtering or the gradient operator. Second, it executes a mathematical function between the mask coefficients and corresponding data in the current area. Normally, each window operation produces a result at the central position of the window in the output image. This complex task could be time-consuming for most conventional microprocessor, considering that a microprocessor can only process a task serially due to the lack of data-path units and the limitation of the bandwidth between the CPU and the memory system. But with custom hardware, we can attempt to finish it in one clock cycle (if given a specialized memory buffer and sufficient data-path units).

2) *Segment Buffer Scheme:* For simplicity without losing generality, we take the 3×3 window operation in a 640×480 image as an example in Fig. 13. The 3×3 sliding window is defined by the thick lines. Fig. 13(a) shows the location of first window, and Fig. 13(b) shows the second window. Fig. 13(c) and (d) shows the conditions when the sliding window changes a row, which involves some vertical movement.

In Fig 13, the data in squares with colors are the data which need to be stored in advance in some segment buffer. When the window moves to the right by one position, the data in the

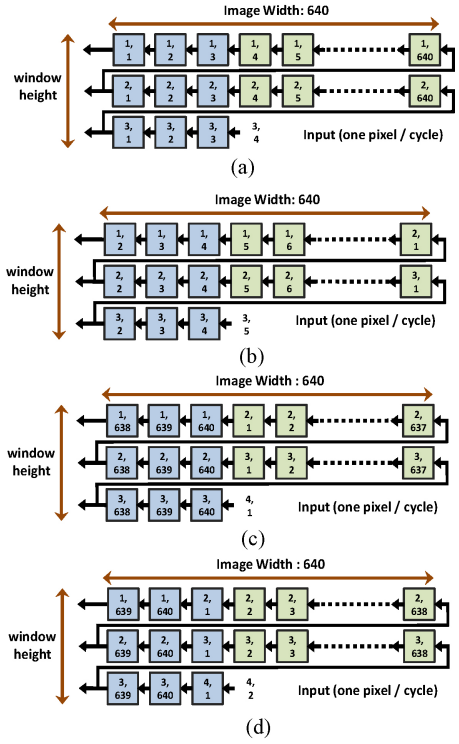

Fig. 14. Segment buffer using shift registers for 3×3 windows in a 640×480 image. Each illustration from (a) to (d) corresponds to one sliding window in Fig. 13(a)–(d).

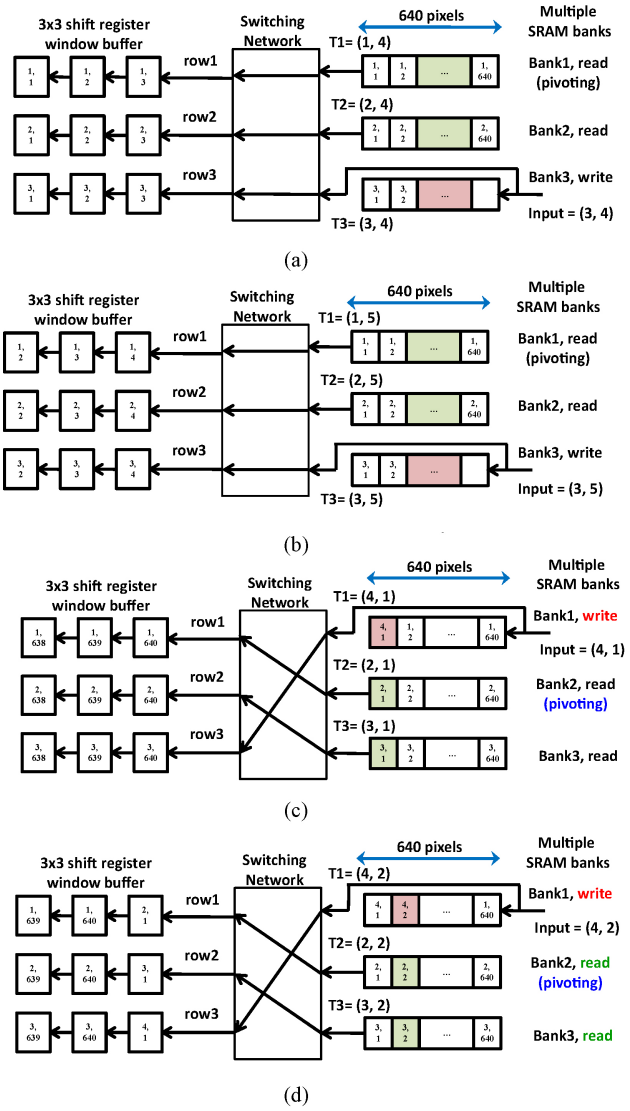

Fig. 15. Rotating SRAM banks for the 3 × 3 sliding window operation in a 640 × 480 image. Each illustration from (a) to (d) corresponds to the sliding window positions in Fig. 13(a)–(d).

new window area are divided into two kinds:

a) the data in the leftmost two columns can be inherited from the previous window data;
b) the data in the rightmost column are new and needs to be further supplied. Among them, two of them are available from the segment buffer, while the last one needs to be provided by some other external source.

Next, we will discuss two implementations, namely: 1) shift register based implementation [23], and 2) SRAM-based implementation for the segment buffers.

a) *Shift registers based implementation:* The window moving operation of a segment buffer can be implemented with shift registers as suggested in [23]–[25] illustrated in Fig. 14.

In this case, the window area does not need its own buffer space—it is simply leftmost part of the overall larger segment buffer. There are three rows of shift registers, which are further connected as a global chain with the serial input port at the lower-right corner. Taking a local view on the window part, the leftmost three registers are independent output ports that can be shifted out at each clock cycle. At this moment, three new data coming in from the window's right end simultaneously. In summary, at each clock cycle, one new data is fed to the overall segment buffer (which has three rows by 640 columns in this case) serially from the rightmost end of the last row from a global point of view. On the other hand, if we focus on the window area, then three data are shifted into the window area (from the overall segment buffer), while three data are shifted out and discarded, all in parallel. The registers in the 3 × 3 window area are directly connected to the input ports of the computing modules where the window operation is conducted. It is notable that this scheme supports streaming data flow and vertical window movement.

b) *SRAM-based implementation:* Although the shift registers can provide the needed data flow for sliding window operation, the area overhead is extremely high—considering that each register is realized by flip-flops while a flip-flop may have up to 30+ transistors. Therefore, we need to convert the above scheme to a structure realizable by SRAMs. One of the major contributions we make in this paper is such an architecture using rotating SRAM banks. For simplicity, we take three rotating SRAM banks for a 3 × 3 sliding window operation as an example in the following.

As shown in Fig. 15, it consists of three main components: 1) the 3 × 3 widow buffer made of registers at the left; 2) a switching network in the middle; and 3) the original 640 × 3 segment buffer made of three independent SRAM banks to

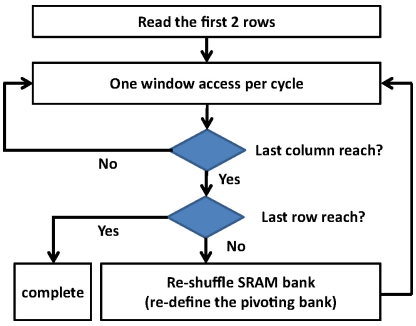

Fig. 16. Operation flow of the multi-access SRAM banks for 3×3 windows.

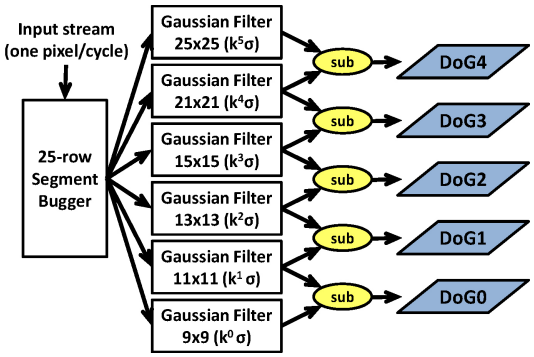

Fig. 17. Block diagram of parallel Gaussian filters and difference-of-Gaussians computation.

the right. It is notable that we can use a 6T SRAM to realize the segment buffer now. Since the cost for one-bit storage is reduced from 30 transistors (in a flip-flop) to 6 transistors (in an SRAM cell), this scheme enables about 80% area reduction for the segment buffer part.

Fig. 15(a)–(d) expresses the corresponding operations with Fig. 13(a)–(d). The number of SRAM banks is equal to the height of window, and the number of data in each bank is equal to the image width. With the data streaming in (one data per clock cycle), this multiple-bank SRAM can provide three data in one column simultaneously to the window buffer per clock cycle. The switching network is used for reordering the data coming out of the multiple-bank SRAM when needed in order to enable row shuffling (to be described in detail later). We need an initialization process so that this segment buffer can allow our system to achieve the throughput of one-window-access per clock cycle. During the initialization, we need to fill out the first two rows of data into the first two SRAM banks and the first two data in the third SRAM bank. After that, we only need to write data into the third bank gradually, one data at a time.

When one row of data is exhausted, and we are about to scan the next row, an operation—called row-shuffling—associated with the switching network is needed. We define a term called "pivoting bank" below so as to explain this concept more easily. Basically the three SRAM banks are ordered, each corresponding to a row in the objective image (which could be source image, or Gaussian-filtered image, or DoG image, and so on). This is to fix the problem when the logical order (i.e., the order in the image) of the SRAM banks mismatches that of their physical order.

Definition 3 (Pivoting Bank): The pivoting bank in the SRAM-based segment buffer is the bank that holds the row of data with the highest logic order.

Fig. 16 shows the flow chart of the row-shuffling operation. We need to constantly check if the last data in a column has been reached. If yes, we move the pivoting bank down by one position. By changing the control bits to the switching network, we can reflect such a situation. The mission of the switching network is to connect the output port of the pivoting bank to the first row of the window buffer—in which the physical order matches their logic order at all times. Such a row-shuffling situation occurs in Fig. 15(b) and (c) when we move the pivoting back from bank 1 of the SRAM to bank 2. It is also notable that such a scheme can support parallel processing easily as shown in Fig. 17, where the six Gaussian filtering blocks in a octave are processed concurrently while sharing the same input 25-row segment buffer. In this case, the window sizes vary from the smallest 9×9 to the largest 25×25.

D. Feature Generation Process

For each identified KP-region, the co-processor will generate a 132-element vector as its feature descriptor—containing information like: 1) the x-coordinate of the key-point; 2) the y-coordinate of the key-point; 3) the Gaussian-scale number the key-point is related to; 4) the principle orientation of the KP-region; and 5) 128-element vector describing the gradient histograms of 16 subregions (as illustrated previously in Fig. 4). For each subregion, there are eight values to be derived, each representing the accumulated gradient magnitude in a 45° orientation bin. It is notable that the size of the KP-region (and the size of each subregion) is variable depending on the Gaussian-scale the key-point is related to. In our co-processor, we can adapt to one of several KP-region sizes.

At the very beginning, the gradient information of each pixel (including the gradient magnitude and the gradient orientation) has been recorded in some segment buffer produced by the key-point identification part. Our design converts the raw information into the feature descriptor in three substages: 1) principle orientation computation; 2) the gradient histogram generation for each subregion; and 3) normalization.

In the first substage, the computation of the principle orientation involves several operations, e.g., assigning the gradient magnitude of each pixel to one of 36 bins—each representing a 10° orientation bin. The gradient magnitudes of all pixels belonging to the same bin are accumulated with a Gaussian weight based on the distance of that pixel away from the center of the KP-region. At the end, we elect an orientation bin with the largest accumulated gradient magnitude. Its corresponding orientation will be the final principle orientation. As for the Gaussian weighting function, since it is static we can thus use a precomputed look-up table and a multiplier to simplify the task. We use a finite state machine to regulate the above procedure in hardware.

In the second substage, the statistical gradient histogram of each subregion is computed in a similar way as the first

substage—by operations such as deciding the orientation bin of a pixel, accumulating the gradient magnitude in a bin, and so on. But the difference is that we have two more operations to perform. First, in order to achieve orientation invariance, it is required in the SIFT algorithm to rotate the coordinates of all pixels in the KP region with respect to the principle orientation derived in the first substage. In hardware, this can be done by applying a rotating matrix to the (x, y)-coordinate vector of a pixel, and then subtract the gradient orientation of the pixel by the principle orientation. Again, we use a finite state machine to regulate the procedure that iterates through each subregion. For each subregion, we identify its boundary and then process each pixel within it. Finally, we will produce eight accumulated magnitude values for each subregion, (one for each 45° orientation bin). There are 16 subregions, so we have 16*8 = 128 magnitude values in total. In the third normalization substage, we perform a process that converts each of the 128 magnitude values into an integer in the range of [0, 255]. This normalization process requires some root-mean-square hardware and some divisor in hardware.

IV. Experimental Results

A. Accuracy

The accuracy is assessed by comparing the results produced by our SIFT accelerator to those reported by a SIFT software. Since a SIFT software often uses the floating-point number while our hardware uses the fixed-point number, certain mismatch between them exists, which is approximated by

$$\text{Error}\% = \frac{\text{Software Value} - \text{Hardware Value}}{\text{Software Value}} \times 100\%. \quad (5)$$

1) *Word-Length for Exact Key-Point Detection:* In general, our hardware detects exactly the same set of key points as the reference SIFT software. In other words, our hardware does not have under-detection or over-detection, based on the results of three test images. For example, we reported 151 key points for a *beaver image* in the VGA-format. This is achieved at the cost of increasing the word-lengths of the pixels representing the DoG images. For example, if only 13 bits are used to represent the fractional part of a DoG image pixel, then in this case our hardware will detect only 137 key points (with five of them false-positive and seven false-negative). We gradually increase the word-length of the fractional part from 13 bits to 16 bits to resolve the misdetection completely. For a scenery photo taken at National Tsing Hua University, Hsinchu, Taiwan, there are 785 key points as shown in Fig. 18(b), and 882 key points for a photo featuring a male person as shown in Fig. 18(c). This word-length can achieve exact detection for these two images as well.

2) *Error of Gradient and Feature Descriptors:* The gradient magnitude and orientation are calculated from the Gaussian-filtered image. We represent the gradient of a Gaussian-filtered pixel as two parts—the horizontal part and the vertical part—each having 20 bits (with one sign bit, 8 integer bits and 13 fractional bits).

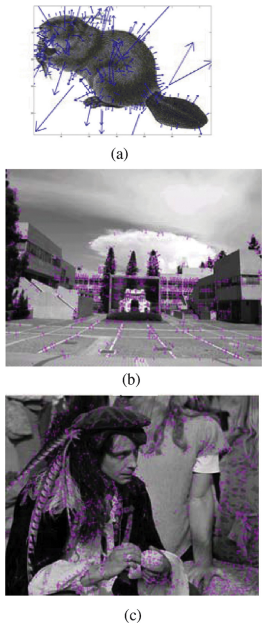

Fig. 18. SIFT features of a few VGA images. The arrows express the detected key points. (a) Beaver image with 151 key points. (b) Scenery photo taken at National Tsing Hua University with 785 detected key points. (c) Image of a male person with 882 key points.

TABLE IV
Error Versus the Data Word-Length (as Comparing the Results Produced by Our Hardware to Those Produced by a SIFT Software)

	Word Length (Bits)		Average Error%
	Integer	Fraction	
Source image	8	0	None
DoG pixel	8	16	0.034%
Gradient magnitude	11	13	0.017%
Gradient orientation	9	0	0.435%

Then, the CORDIC algorithm [26] is used to compute the gradient magnitude and the gradient orientation iteratively. For gradient magnitude computation, we finally select a fixed-point number format with 11 integer bits and 13 fraction bits which can reduce the average error to about 0.017% as compared to the software result. For gradient orientation computation, a fixed-point number format with nine integer bits can reduce the average error to about 0.435% as summarized in Table IV.

Each of the feature descriptors our SIFT hardware reports for a key point consists of five parts: 1) the x-coordinate of the key-point (10 integer bits); 2) the y-coordinate of the key-point (10 integer bits); 3) the associated value of mapped scaling-factor in the Gaussian filtering (4 integer bits plus 6 fractional bits); 4) the principle orientation of the key-point (3 integer bits plus 6 fractional bits); and 5) a 128-D vector (with each element represented in 10 bits).

The errors of these terms are all summarized in Table V. In general, we achieved zero-error for the coordinates of the key points (for achieving exact key point detection), and low errors for the scaling factor (0.22%) and the principle orientation (0.39%). As for the overall 128-D feature descriptor, its accuracy is not as demanding as the previous four parts in

TABLE V
WORD LENGTHS OF THE VALUES REPORTED IN THE FINAL FEATURE DESCRIPTORS AND THEIR AVERAGE ERRORS

	Word Length (Bits)			Type	Average Error%
	Sign	Integer	Fraction		
KP location (y)	0	10	0	Integer	0
KP location (x)	0	10	0	Integer	0
Scale number	0	4	6	Fixed-point	0.22%
KP orientation (rad)	1	3	6	Fixed-point	0.39%
Each element of the 128-D descriptor	0	10	0	Integer	9.49%

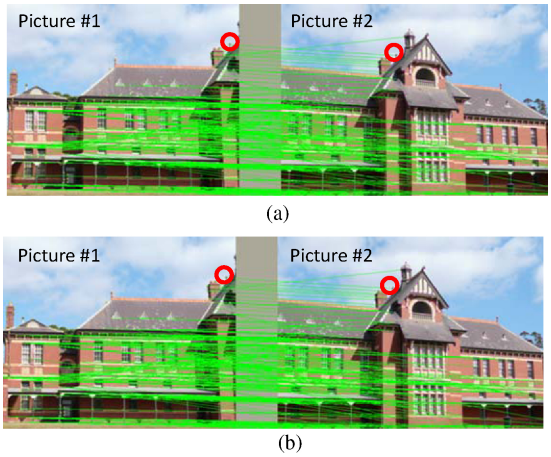

Fig. 19. SIFT-based key-point matching between two overlapping pictures. (a) Key-point matching based on SIFT software. (b) Key-point matching based on our SIFT hardware.

most applications and thus we allow a relatively larger error (9.49%) to save hardware cost. However, if needed, it can also be made more accurate by increasing the word lengths of related data.

To investigate if the accuracy of our SIFT hardware is adequate, we implemented a panoramic image stitching program in which SIFT is used for key-point matching between two overlapping pictures [6]. On one hand, we take the feature descriptors reported by a software SIFT program (using floating-point numbers internally) as the input. On the other hand, we take the feature descriptors reported by our SIFT hardware (using fixed-point numbers internally) as the input. Fig. 19 shows the results. For both cases, the matching results are the same. This implies that the accuracy of our SIFT hardware is adequate for this application even though the SIFT hardware has introduced 9.49% error in the feature descriptor on the average. It is worth mentioning that in this paper we wish to focus more on the architecture and hardware design needed for achieving high-performance feature extraction. As for the accuracy and its dependence on the word-lengths of the internal number system may need more involved analysis and that analysis is beyond the scope of this paper.

B. Performance Evaluation

The processing time of one VGA frame is divided into two parts: 1) the key-point identification part, and 2) feature generation part. At the operation frequency of 100 MHz, the processing time of a VGA frame is about 3.4 ms, which is directly proportional to the size of input image. The processing time of the second part is proportional to the number of detected key points. The total feature generation time of one image frame can be expressed roughly as the number of features times the average generation time for each feature (which is approximately 0.0331 ms). Expression (6) summarizes the overall processing time for one VGA image frame

$$\begin{aligned}&\text{SIFT processing time of one VGA frame}\\ &\cong \text{Keypoint indetification time} + \text{features generation time}\\ &\cong 3.4\,\text{ms} + \text{numbers of features} \times 0.0331\,\text{ms}.\end{aligned} \tag{6}$$

The above expression implies that we can process one VGA image frame within 33 ms when the number of features is under 890 roughly. For example, to processing the VGA beaver image which has 151 features, the processing time is only 8.397 ms.

TABLE VI
CHARACTERISTICS OF OUR SIFT HARDWARE DESIGN

Technology	TSMC 0.18 μm CMOS
Function	SIFT hardware
Operation frequency	100 MHz
Gate count	1320K
Memory usage	5.729 Mb

TABLE VII
MEMORY USAGE (AS COMPARED TO SOME RECENT WORKS)

	Stage	Bonato [19]	Yao [20]	Ours	
Frame size		QVGA	VGA	VGA	
Memory usage (M bits)	DoG	1.35 Mb	3.24 Mb	0.224 Mb	0.896 Mb
	Key point detection			0.404 Mb	
	Gradient magnitude and orientation			0.241 Mb	
	Feature generation	16 Mb	8.192 Mb	4.86 Mb	
	Total	17.35 Mb	11.432 Mb	5.729 Mb	

C. Design Characteristics

The proposed SIFT hardware is implemented using a TSMC 0.18 μm standard CMOS process technology. The hardware is completely designed as a synthesizable Verilog code, synthesized to a cell-based netlist by a tool called Design Compiler. The equivalent gate count (in terms of two-input NAND gates) is about 1320K, and it can support up to 100 MHz. The on-chip memory required consists of two parts: 1) the buffers within the key-point identification component, which is 0.89 Mb, and 2) the input buffer for feature generation component (i.e., the buffer between the key-point identification component and feature generation component), which is 4.86 Mb. Table VI summarizes the major characteristics of our SIFT hardware design.

Relatively, our design uses much less memory than the recent related works as shown in Table VII. Due to the

proposed segment buffer scheme, we achieve 67% and 50% reductions on the overall memory requirement than the works proposed in [19] and [20], respectively.

V. Conclusion

The SIFT algorithm has been a known effective means of extracting features from a given image. Due to its high computational complexity, hardware acceleration is often needed in many computer vision applications. In this paper, we presented a SIFT hardware accelerator which is the fastest so far to our knowledge. As compared to previous works, it features several distinctive characteristics. First, it is the first work that attempts to perform the feature descriptor generation component in hardware that could form another performance bottleneck in addition to the key point identification component. Second, a 3-stage pipeline design along with fine-grain parallel processing has been demonstrated to speed up the operation by several times. Third, our design enables a streaming type of data flow that uses a 50% less memory, and the memory part can be realized by SRAM macros to reduce the area overhead. In the 0.18 TSMC CMOS process, our design uses 1320K equivalent 2-input NAND-gates plus 5.729 Mb of SRAM, and can support operation running at a clock rate of up to about 100 MHz. For VGA images, it can process the SIFT operation in real-time (i.e., roughly 33 ms per image frame) when the number of key points in the image frame is less than 890.

Acknowledgment

The authors would like to thank to the Chip Implementation Center of Taiwan, Hsinchu, Taiwan, for its assistance in providing some needed electronic design automation tools.

References

[1] H. P. Moravec, "Toward automatic visual obstacle avoidance," in *Proc. 5th Int. Joint Conf. Artif. Intell.*, Aug. 1977, p. 584.

[2] H. P. Moravec, "Obstacle avoidance and navigation in the real world by a seeing robot rover," Ph.D. dissertation, Comput. Sci. Dept., Stanford Univ., Stanford, CA, 1980.

[3] C. Harris and M. J. Stephens, "A combined corner and edge detector," in *Proc. Avley Vis. Conf.*, 1988, pp. 147–152.

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

[5] S. Se, D. G. Lowe, and J. J. Little, "Vision-based global localization and mapping for mobile robots," *IEEE Trans. Robot.*, vol. 21, no. 3, pp. 364–375, Jun. 2005.

[6] M. Brown and D. G. Lowe, "Recognizing panoramas," in *Proc. 9th IEEE Int. Conf. Comput. Vis.*, Oct. 2003, pp. 1218–1225.

[7] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. 7th IEEE Int. Conf. Comput. Vis.*, Sep. 1999, pp. 1150–1157.

[8] D. G. Lowe, "Local feature view clustering for 3-D object recognition," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. I. Dec. 2001, pp. 682–688.

[9] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded up robust features," in *Proc. 9th Eur. Conf. Comput. Vis.*, 2006, pp. 404–417.

[10] M. Grabner, H. Grabner, and H. Bischof, "Fast approximated SIFT," presented at the Asian Conf. on Computer Vision, Hyderabad, India, 2006.

[11] G.-S. Hsu, C.-Y. Lin, and J.-S. Wu, "Real-time 3-D object recognition using scale invariant feature transform and stereo vision," in *Proc. 4th Int. Conf. Autonomous Robots Agents*, 2009, pp. 239–244.

[12] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau, "Real-time eye blink detection with GPU-based SIFT tracking," in *Proc. 4th Can. Conf. Comput. Robot Vis.*, 2007, pp. 481–487.

[13] N. Pettersson and L. Petersson, "Online stereo calibration using FPGAs," in *Proc. IEEE Intell. Vehicles Symp.*, Jun. 2005, pp. 55–60.

[14] J. Qiu, T. Huang, and T. Ikenaga, "A FPGA-based dual-pixel processing pipelined hardware accelerator for feature point detection part in SIFT," in *Proc. 5th Int. Joint Conf. INC IMS IDC*, 2009, pp. 1668–1674.

[15] H. D. Chati, F. Muhlbauer, T. Braun, C. Bobda, and K. Berns, "SoPC architecture for a key point detector," in *Proc. Int. Conf. Field Programmable Logic Applicat.*, 2007, pp. 710–713.

[16] H. D. Chati, F. Muhlbauer, T. Braun, C. Bobda, and K. Berns, "Hardware/software co-design of a key point detector on FPGA," in *Proc. 15th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, Apr. 2007, pp. 355–356.

[17] D. Kim, K. Kim, J.-Y. Kim, S. Lee, S.-J. Lee, and H.-J. Yoo, "81.6 GOPS object recognition processor based on a memory-centric NoC," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 3, pp. 370–383, Mar. 2009.

[18] S. Se, D. G. Lowe, J. J. Little, H. K. Ng, P. Jasiobedzki, and T. J. Moyung, "Vision based modeling and localization for planetary exploration rovers," in *Proc. Int. Astronautical Congr.*, 2004, p. 11.

[19] V. Bonato, E. Marques, and G. A. Constantinides, "A parallel hardware architecture for scale and rotation invariant feature detection," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 12, pp. 1703–1712, Dec. 2008.

[20] L. Yao, H. Feng, Y. Zhu, Z. Jiang, D. Zhao, and W. Feng, "An architecture of optimized SIFT feature detection for an FPGA implementation of an image matcher," in *Proc. Int. Conf. Field-Programmable Technol.*, 2009, pp. 30–37.

[21] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-D convolvers for fast digital signal processing," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 7, no. 3, pp. 299–308, Mar. 1999.

[22] P.-Y. Hsiao, C.-H. Chen, H. Wen, and S.-J. Chen, "Real-time realization of noise-immune gradient-based edge detector," *IEE Proc. Comput. Digital Tech.*, vol. 153, no. 4, pp. 261–269, Jul. 2006.

[23] F. M. Alzahrani and T. Chen, "A real-time edge detector: Algorithm and VLSI architecture," *Real-Time Imaging*, vol. 3, no. 5, pp. 363–378, 1997.

[24] E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*. Englewood Cliffs, NJ: Prentice-Hall, 1998.

[25] P.-Y. Hsiao, C.-H. Chen, S.-S. Chou, L.-T. Li, and S.-J. Chen, "A parameterizable digital-approximated 2-D Gaussian smoothing filter for edge detection in noisy image," in *Proc. IEEE Int. Symp. Circuits Syst.*, Sep. 2006, pp. 3189–3192.

[26] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proc. ACM/SIGDA 6th Int. Symp. Field Programmable Gate Arrays*, 1998, pp. 191–200.

Feng-Cheng Huang received the B.S. degree in electrical engineering from National Kaohsiung University, Kaohsiung, Taiwan, in 2008, and the M.S. degree in electrical engineering from National Tsing Hua University, Hsinchu, Taiwan, in 2011.

His current research interests include very large scale integration design for intelligent image processing.

Shi-Yu Huang (M'97) received the B.S. and M.S. degrees in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1988 and 1992, respectively, and the Ph.D. degree in electrical and computer engineering from the University of California, Santa Barbara, in 1997.

He joined the faculty of the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, in 1999. His current research interests include mainly very large scale integration (VLSI) design, automation, and testing, with an emphasis on power estimation, fault diagnosis, all-digital phase-locked loop design and its application to delay fault testing in VLSI, nanometer static random-access memory design, and image processing application-specific integrated circuits for image stitching.

Dr. Huang served in the IEEE Asian Test Symposium as the Program Co-Chair in 2004, and as the General Co-Chair in 2009. He also served as the Program Chair of the IEEE International Workshop on Memory Technology, Design, and Testing in 2005 and 2006, respectively.

Ji-Wei Ker received the B.S. and M.S. degrees in electronic engineering from National Sun Yat-Sen University, Kaohsiung, Taiwan, and National Tsing Hua University, Hsinchu, Taiwan, in 2009 and 2011, respectively.

His current research interests include very large scale integration design, including delay-locked loops, duty-cycle correctors, and panoramic image synthesis.

Yung-Chang Chen (F'05) received the B.S. and M.S. degrees in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1968 and 1970, respectively, and the Ph.D. degree in engineering from Technische Universität Berlin, Berlin, Germany, in 1978.

He joined the faculty of the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, in 1978. He is currently a Distinguished Professor with the university. His current research interests include mainly image and video processing.