 http://dx.doi.org/10.5573/JSTS.2013.13.2.157

A novel hardware design for SIFT generation with reduced memory requirement

Eung Sup Kim and Hyuk-Jae Lee

Abstract—**Scale Invariant Feature Transform (SIFT) generates image features widely used to match objects in different images. Previous work on hardware-based SIFT implementation requires excessive internal memory and hardware logic [1]. In this paper, a new hardware organization is proposed to implement SIFT with less memory and hardware cost than the previous work. To this end, a parallel Gaussian filter bank is adopted to eliminate the buffers that store intermediate results because parallel operations allow all intermediate results available at the same time. Furthermore, the processing order is changed from the raster-scan order to the block-by-block order so that the line buffer size storing the source image is also reduced. These techniques trade the reduction of memory size with a slight increase of the execution time and external memory bandwidth. As a result, the memory size is reduced by 94.4%. The proposed hardware for SIFT implementation includes the Descriptor generation block, which is omitted in the previous work [1]. The addition of the hardwired descriptor generation improves the computation speed by about 30 times when compared with the previous work.**

Index Terms—**SIFT, computer vision, hardware implementation, memory reduction, Gaussian filter bank**

Manuscript received Aug. 23, 2012; accepted Nov. 29, 2012.
Both authors are with Inter-university Semiconductor Research Center, Department of Electrical Engineering, Seoul National University, Korea.
E-mail : eskim@capp.snu.ac.kr

I. INTRODUCTION

SIFT (Scale-Invariant Feature Transform) generates one of the popular local image features widely used to match objects in different images [2]. Because of its outstanding performance, it is used for various applications such as object recognition, image stitching, and robot navigation. However, complex computation and excessive memory access make it difficult to process SIFT operation for a large size video in real time. To speed up the SIFT operation, a number of previous research efforts have been made [3-5]. Among them, one presents a hardware-based implementation that achieves a real-time SIFT operation of QVGA-sized (320x240) video at the rate of 30 frames per second [1]. Although this work in [1] enables a real-time SIFT operation, the hardware cost is very large because intermediate results are stored in internal memory inside a chip. In a PC environment, it is an efficient approach to speed up the computation with an increased memory requirement because a PC has sufficient memory but a limited computing power for SIFT computation. However, in a customized ASIC or SoC (System-on-Chip), the use of a large internal memory significantly increases the cost of the chip.

This paper proposes a new hardware architecture and organization for the SIFT algorithm. In order to reduce the hardware cost, the proposed design attempts to reduce the internal buffer. To this end, the new design adopts a parallel Gaussian filter bank, which performs Gaussian filtering operations in parallel with various scales. The use of parallel Gaussian filters reduces the number of line buffers that temporarily store the intermediate results. For an additional reduction of the

internal memory size, the input image is partitioned to be stored in a buffer. As a result, the buffer size to store the input image is also reduced because only the partitioned sub-image needs to be stored in the buffer. The entire procedure for the SIFT computation is implemented in hardware and the computation speed is increased by about 30 times when compared with the previous work [1].

The rest part of this paper is organized as follows. Section II briefly introduces the SIFT algorithm and Section III presents a previous hardware implementation for the SIFT algorithm with analysis on the computation speed and memory requirement. Section IV proposes a new hardware architecture that attempts to reduce the hardware cost and to speed up computation time and Section V describes a hardware organization for further reductions of the hardware cost and memory access. In Section VI, the efficiency of the proposed hardware design is evaluated and conclusions are presented in Section VII.

II. SCALE-INVARIANT FEATURE TRANSFORM (SIFT)

This section briefly introduces the SIFT algorithm. Fig. 1 shows the computation flow of the SIFT algorithm. The procedure is composed of two main steps: keypoint detection and descriptor generation. In the first step, the input image is scanned to find the locations of pixels with special characteristics, called keypoints. In the second step, a feature is created for characterizing each keypoint found in the first step. This feature consists of the histograms of gradients around the keypoint. The keypoint detection step is composed of three substeps: scale-space image generation, local extrema detection, and keypoint detection. The descriptor generation step consists of orientation assignment and descriptor generation substeps. For self-containment, these substeps are briefly explained next and further details are available in [2].

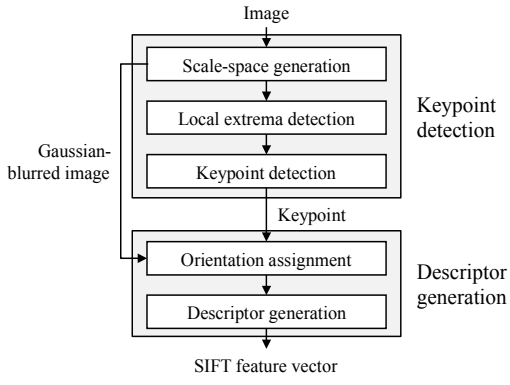


Fig. 1. SIFT algorithm.

1. Scale-space Image Generation

In this step, Gaussian-blurred images are generated by filtering an input image with Gaussian filters. The convolution operation described below produces a Gaussian-blurred image, $L_i(x,y)$, from an input image $I(x,y)$.

$$L_i(x,y) = G(x,y,\sigma_i) * I(x,y) \quad for\ i = 0,1,...S+2 \tag{1}$$

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

$$\sigma_i = \sqrt{\left(\sigma_0 \cdot 2^{\frac{i}{S}}\right)^2 - \sigma_{in}^2} \tag{3}$$

where σ_i is called the *scale of the Gaussian filter* and i is called the *scale index*. S is the number of scaled images to be generated and 3 is chosen for S in this paper as well as in the previous work in [1]. In the later substeps, these scaled images are used to generate keypoints. The Gaussian kernel used in the operation above, $G(x, y, \sigma_i)$, depends on σ_i while σ_0 is a given parameter which represents the scale of the first Gaussian-blurred image. Once σ_0 is given, the scales of the other images are determined from (3) where it is assumed that the input image is Gaussian-blurred with σ_{in} [1].

After the Gaussian Blurred image is generated, the next operation is to derive the DoG (Difference of Gaussians) image, $D_i(x,y)$, which is computed by subtracting L_i from L_{i+1} as described in (4). From $S+3$ $L_i(x,y)$ images, $S+2$ DoG images are produced.

$$D_i(x,y) = L_{i+1}(x,y) - L_i(x,y) \quad for\ i = 0,1,...,S+1 \tag{4}$$

This set of the $S+2$ DoG images generated as described above is called the first octave of the DoG images. The second octave is generated as follows. The

$L_0(x,y)$ for the second octave is derived from the S-th Gaussian-blurred image, $L_S(x,y)$, by down-sampling it by every other pixels along both horizontal and vertical directions. Then, $L_i(x,y)$, for $i=1,...S+2$ is generated using (5) to (7). Note that the scale σ_i in (3) is replaced by σ'_i, in (7) [6].

$$L_i(x,y) = G(x,y,\sigma'_i) * L_0(x,y) \quad for\ i = 1,2,...S+2 \tag{5}$$

$$G(x,y,\sigma'_i) = \frac{1}{2\pi\sigma'^2_i} e^{-(x^2+y^2)/2\sigma'^2_i} \tag{6}$$

$$\sigma'_i = \sigma_0 \cdot \sqrt{\left(2^{\frac{i}{S}}\right)^2 - 1} \tag{7}$$

The next octaves can also be produced in the same manner as the second octave which is generated from (5), (6) and (7) with $L_0(x,y)$ derived by down-sampling the $L_S(x,y)$ of the previous octave. Note that each octave consists of $S+3$ Gaussian-blurred images and $S+2$ DoG images.

2. Local Extrema Detection

A DoG pixel at the location (x,y) of the i-th scale, $D_i(x,y)$, is compared with the 8 DoG pixels around the location in the $(3x3)$ window, and also the DoG pixels in the $3x3$ windows of the $(i+1)$-th and $(i-1)$-th scales. The $D_i(x,y)$, is marked as a keypoint candidate if it has extreme value among the 27 DoG pixels in the $3x3x3$ window. This local extrema detection is performed for every pixel in the DoG images and every extremum point is recorded as a keypoint candidate.

3. Keypoint Detection

In order to select the final keypoints from all keypoint candidates derived in the previous substep, the next substep carries out two tests: contrast check and eliminating edge responses. For contrast check, $D_i(x,y)$ of every keypoint candidate is compared with a predefined threshold. If the value is less than the threshold, the point is discarded from the keypoint candidate. This paper uses 0.03 as the predefined threshold, which is the same valued used in [2]. In eliminating edge response, Inequality (8) is tested. Only the keypoint candidate that satisfies (8) is finally chosen as the keypoint.

$$\frac{Tr(\mathbf{H})^2}{Det(\mathbf{H})} < \frac{(r+1)^2}{r} \tag{8}$$

$$\begin{aligned} \mathbf{H} &= \begin{bmatrix} \Delta_{xx} & \Delta_{xy} \\ \Delta_{xy} & \Delta_{yy} \end{bmatrix} \\ \Delta_{xx} &= D_i(x+1,y) + D_i(x-1,y) - 2D_i(x,y) \\ \Delta_{yy} &= D_i(x,y+1) + D_i(x,y-1) - 2D_i(x,y) \\ \Delta_{xy} &= (D_i(x+1,y+1) - D_i(x-1,y+1) \\ &\quad - D_i(x+1,y-1) + D_i(x-1,y-1))/4 \end{aligned} \tag{9}$$

where r is a predefined threshold chosen as 10 [2] and $Tr(\mathbf{H})$ and $Det(\mathbf{H})$ are computed as follows:

$$\begin{aligned} Tr(\mathbf{H}) &= \Delta_{xx} + \Delta_{yy} \\ Det(\mathbf{H}) &= \Delta_{xx}\Delta_{yy} - \Delta_{xy}^2 \end{aligned} \tag{10}$$

4. Orientation Assignment

For a window of size $(2 \cdot Round(\sqrt{2} \cdot 3\sigma \cdot \frac{N+1}{2}) + 1)$ x $(2 \cdot Round(\sqrt{2} \cdot 3\sigma \cdot \frac{N+1}{2}) + 1)$ around a keypoint, gradients are computed for all pixels in this area. Note that σ denotes the scale of a keypoint, N is chosen as 4 [2], and $Round()$ represents the rounding function [7]. Gradient is computed in the horizontal and vertical directions as follows:

$$\begin{aligned} \Delta_x &= (L_i(x+1,y) - L_i(x-1,y))/2 \\ \Delta_y &= (L_i(x,y+1) - L_i(x,y-1))/2 \end{aligned} \tag{11}$$

Then, the gradient magnitude, $m(x,y)$ and the gradient orientation, $\theta(x,y)$, are obtained from (12).

$$m(x,y) = \sqrt{\Delta_x^2 + \Delta_y^2}, \quad \theta(x,y) = \tan^{-1}\left(\frac{\Delta_y}{\Delta_x}\right) \tag{12}$$

5. Descriptor Generation

Within the window of size $(2 \cdot Round(\sqrt{2} \cdot 3\sigma \cdot \frac{N+1}{2}) + 1)$ x $(2 \cdot Round(\sqrt{2} \cdot 3\sigma \cdot \frac{N+1}{2}) + 1)$, a sub-region of size $(2 \cdot Round(4.5\sigma) + 1)$ x $(2 \cdot Round(4.5\sigma) + 1)$ centered at the same pixel as the window is defined and all locations in this sub-region are used to build a gradient

orientation histogram. The gradient orientation at each location within the sub-region is mapped to one of the 36 bins. Each bin covers 10 degree and total 36 bins cover 360 degree of orientations. The gradient magnitude at each location within the sub-region is weighted by a Gaussian window with a scale σ that is 1.5 times that of the scale of the keypoint.

The value of the histogram bin is made by summing all of the weighted magnitudes with the same gradient orientation. After the orientation histogram is generated, the bin with the largest value is picked as the dominant orientation of the keypoint. If there are bins with larger than 80% of the largest value, new keypoints with that orientation are created. In other words, more than one descriptor can be generated for the same location with different dominant orientations.

Along the derived dominant orientation, all gradients obtained from (12) are rotated. To this end, the gradient orientation, *(x,y)* obtained as in (12), is subtracted from the dominant orientation. The gradient magnitude is weighted by a Gaussian weighting function with its standard deviation σ_{window} which is equal to a half of the width of the descriptor window. Within the rotated gradient window, another sub-region, the descriptor window, is defined as the square of size *(W x W)* around a keypoint where *W* is defined as follows:

$$\mathrm{W} = (2 \cdot Round(3\sigma \cdot \frac{N+1}{2}) + 1) \qquad (13)$$

The descriptor window is partitioned into *(N+1)x(N+1)* subregions. There are *NxN* points at which the edges of four adjacent subregions cross. Each point of them has a gradient histogram with 8 orientation bins. The value of each gradient sample within the window is distributed into adjacent histogram bins. For each subregion, there are at most eight adjacent histogram bins along *x*, *y*, and orientation bins each of which makes two bins independently. Suppose that the distance of the gradient sample from the center of the bin is *[dx, dy, do]*. Then, each entry of the gradient sample is multi1plied by weight *(1-dx)(1-dy)(1-do)*. With the histograms of 16 sub-regions, 16x8 bins are concatenated to form a vector, **f**. It is normalized such that its Euclidean norm [8] becomes a unit length, and then each component of the normalized vector is clipped not to exceed the predefined threshold, 0.2 [2]. The vector, which is re-normalized once again after the clipping operation, becomes the final image features. Further details about the SIFT algorithm are presented in [2].

III. Analysis of the Memory Requirement by the previous SIFT Implementation

As SIFT requires pixel-by-pixel operations, the computational complexity increases as the image size increases. Furthermore, the amount of computation required for processing each pixel is also very large. Thus, previous research efforts have been made on the effective implementation to speed up SIFT generation [1, 3-5]. However, the speed-up is achieved at the cost of increased hardware cost, especially increased memory requirement. This section briefly introduces these previous research efforts and explains the computation speed and memory requirement of the previous implementations.

In order to reduce the number of computations, Lowe in [2] modifies the Gaussian filtering operations of (1) and (5) by using a Gaussian filter bank in a cascade form. In the cascade filter bank, the result of a Gaussian filter is fed to the input of the next Gaussian filter as described in (14).

$$L_i(x,y) = G(x,y,\sigma_{cascade,i}) * L_{i-1}(x,y) \qquad (14)$$

As the cascade Gaussian filter must perform the same operations as (1) or (5), $\sigma_{cascade,i}$ is chosen to make (14) equal to (1) and (5) [6].

$$\begin{aligned} \sigma_{cascade,i} &= \sqrt{\sigma_i^2 - \sigma_{i-1}^2} \\ &= \sigma_0 \sqrt{(2^{\frac{i}{S}})^2 - (2^{\frac{i-1}{S}})^2} \\ &\quad for\ \ i = 1,2,...,S+2 \end{aligned} \qquad (15)$$

The reason why the cascade Gaussian filtering reduces computational complexity is because the filter length of the cascade form is smaller than that of the original form. Note that $\sigma_{cascade,i}$ defined by (15) is smaller than σ_i in (3) or σ'_i in (7). A Gaussian filter is, in general, approximated as an FIR filter of which coefficient

decreases rapidly as the order increases. As the coefficient of a large order is approximated as 0, the width of a Gaussian filter with non-zero coefficients is proportional to σ, the standard deviation of Gaussian function. Let W_{Gauss} denote the width of a Gaussian filter, then W_{Gauss} is given as follows [7, 9]:

$$W_{Gauss} = 2 \cdot Round(3\sigma) + 1 \tag{16}$$

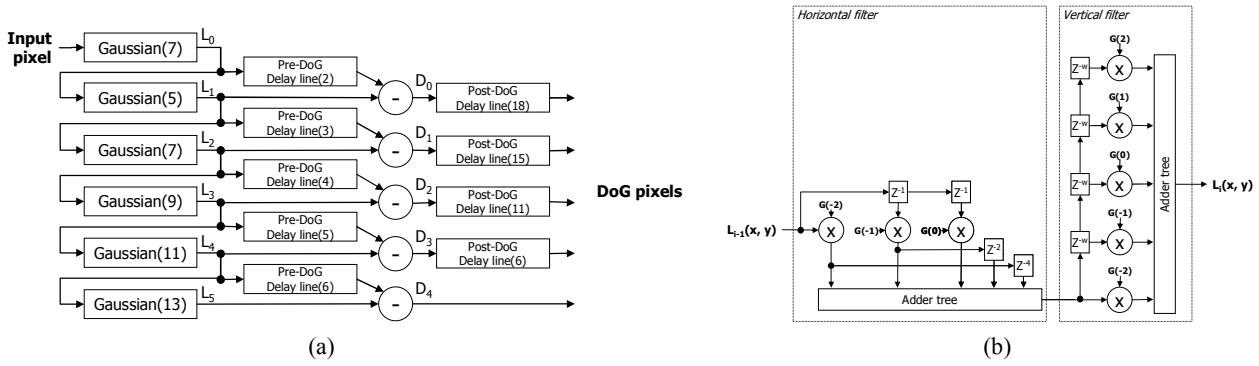
For example, given $\sigma=0.87$, W_{Gauss} becomes *7*, and the Gaussian kernel is *[0.0012, 0.0326, 0.2369, 0.4586, 0.2369, 0.0326, 0.0012]*. Given $\sigma=0.87$, W_{Gauss} becomes *9*, and the Gaussian kernel is *[0.0056, 0.0309, 0.1050, 0.2188, 0.2794, 0.2188, 0.1050, 0.0309, 0.0056]*.

Bonato in [1] proposes a hardware implementation that includes cascade Gaussian filters as in [2]. Although the organization may reduce computational complexity, it may increase the cost of the hardware because it requires a number of delay buffers, which are necessary to store the intermediate results of the Gaussian filtering operation. Fig. 2(a) represents the Gaussian filter bank in the implementation presented in [1]. The block labeled "Gaussian" represents the hardware resource for Gaussian filtering operation. An input image is processed by the uppermost Gaussian filter, and then L_0 is generated. The numbers in parentheses in the right of the label "Gaussian" represents the filter width when $\sigma_0=1$. The filter width of the first Gaussian filter is equal to 7. The output of the first filter, L_0, is forwarded to the input of the second filter. Then, the second filter produces L_1. With the cascaded operations, all Gaussian-blurred images from L_0 to L_5 are generated.

A DoG image is generated by computing the differences between the Gaussian-blurred images. To obtain one DoG image, two Gaussian blurred images are necessary. In this cascade filter organization, the two blurred images are not produced at the same time. Thus, a buffer is necessary to store the blurred image that is produced ahead of the other blurred image for the derivation of a DoG image. In Fig. 2, the block labeled "Pre-DoG Delay line" represents such a buffer. The numbers labeled within parentheses in the right of the label represents the number of the lines to be stored in the buffer when $\sigma_0=1$. For instance, when the size of an input image is *1280 x 720*, 2 lines mean that the buffer needs to store *1280 x 2* pixels. In Fig. 2(a), total 20 lines are required for the Pre-DoG delay buffers.

In Fig. 2(a), DoG images are stored in the Post-DoG delay line buffers [1]. These buffers are necessary to have all DoG images available for a given pixel as each DoG image is produced in different time. Thus, delay line memories are placed except for the last DoG image. The size of the memories is determined by the difference of the time between the generation of the current DoG image and the generation of the last DoG image [1].

In the implementation shown in Fig. 2(a), additional buffers are necessary for implementation of Gaussian filters. Fig. 2(b) shows the internal organization of the Gaussian filter of width 5 for example. For two-dimensional Gaussian filtering operations, one-dimensional Gaussian filtering is performed twice in the horizontal and vertical directions. Each pixel of the input image is the input to the horizontal filter in the raster scan order. The result of the horizontal filter is fed into the vertical filter. In order to feed the pixels necessary for the vertical filtering operations, it is necessary to store the results of the horizontal filter. In Fig. 2(b), Z^{-W} indicates one line memory and in total, 4 line memories



(a) (b)

Fig. 2. Cascade Gaussian filer bank (a) Filter bank structure, (b) Gaussian filter structure.

are required as the filter width is 5.

IV. Proposed Hardware Architecture with Reduced Memory Requirement

In this section, two new techniques to reduce the number of the line buffers storing the input image and the intermediate results are proposed. The first technique is parallel computation of Gaussian filtering operations, which reduces the internal memory at the cost of increasing computational complexity. The second technique is a change of the processing order by partitioning an input image into subblocks and processing the block-by-block order. This technique reduces internal memory at the cost of increasing external memory bandwidth. Details of these two techniques are explained next in this section.

1. Parallel Gaussian Filter Bank

Fig. 3(a) shows the architecture for the Parallel Gaussian filter bank. The input image is stored in the source line buffer, and the stored image is input to all Gaussian filters, which operate in parallel. As the results of all Gaussian filters are available simultaneously for the next operations, pre-DoG delay line buffers are not necessary. The size of the source line memory is the same as the width of the widest Gaussian filter because the shared source line buffer provides input to all Gaussian filters.

Fig. 3(b) shows the internal organization of the two-dimensional Gaussian filters. In this organization, the vertical filtering operation is performed ahead of the horizontal filtering operation. The vertical filter receives multiple pixels at a time and generates the result pixel-by-pixel, which is fed into the horizontal filter. With the interchange of the processing order between the vertical operation and horizontal operation, the internal line buffer to store intermediate results of the filter is also no longer necessary.

Comparison between Fig. 2(a) and Fig. 3(a) shows that the parallel Gaussian filter bank significantly reduces the number of line memories. The memory sizes shown in Fig. 2 and 3 are obtained for the case of $\sigma_0=1$. For the organization in Fig. 2(a), 20 lines in the Pre-DoG delay buffers, 50 lines in the Post-DoG delay buffers, and 46 lines in the buffer storing the horizontal filtering results are required. On the other hand, for the organization in Fig. 3(a), only 18 lines within the common source line buffer is needed, so that 98 line memories are eliminated by the parallel Gaussian filter. Note that the reduction of the line memory is achieved at the cost of increasing computational complexity. This is because the filter width of the Parallel Gaussian filter is wider than that of the cascade one.

2. Block-based Gaussian Filter

In addition to the parallel Gaussian filter presented in the previous subsection, further memory reduction is achieved by partitioning an input image into multiple blocks and changing the processing order from the raster-

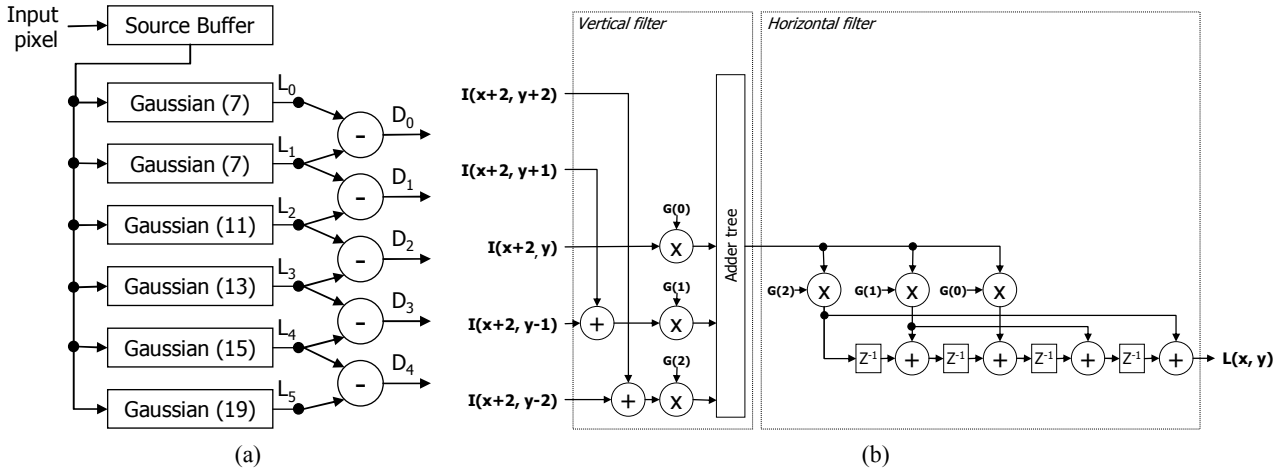


Fig. 3. Parallel Gaussian filer bank (a) Filter bank structure, (b) Gaussian filter structure.

scan order to the block-by-block order. Details are explained next.

Fig. 4 shows the new processing order of pixels in an input image. In this figure, an input image is partitioned into 15 blocks. Pixels within each block are processed in the vertical direction, which is represented by the dotted arrows in a block. After a column is processed, the right column is processed next. After all pixels in a block are processed, the next block is processed in the raster scan order as indicated by the thick arrows. After the first octave are completed, $L_S(x,y)$ is loaded for the next octave. This is repeated until all octaves are processed.

The processing order of pixels is determined according to the first filtering direction of 1D Gaussian filter and the structure of the source block memory, which is explained later. The first filtering direction of 1D Gaussian filter should be perpendicular to the processing order of pixels not to have a temporal storage between the first and second 1D Gaussian filters. Thus, if pixels are processed in the vertical direction, the direction of the first 1D Gaussian should be horizontal.

Fig. 5 shows the architecture of the source pixel block buffer. The width of a block is equal to $(W_{Gauss}-1)/2$ where W_{Gauss} represents the width of the widest Gaussian filter. The vertical size of a block is determined based on the available memory bandwidth, which is explained later in this subsection. Recall that W_{Gauss} is

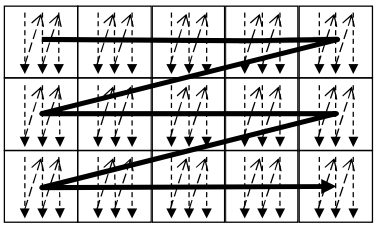


Fig. 4. Block-by-block processing order.

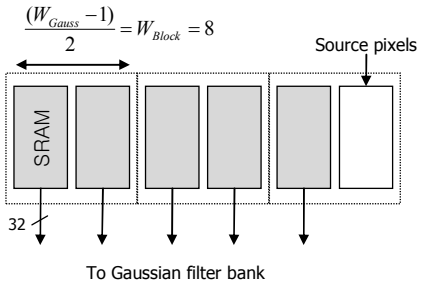


Fig. 5. The organization of the source block memory.

19 when σ_0 is 1. Then, the width of the block becomes 9 pixels. To access 9 pixels at a time, three SRAMs with a 32-bit data bus are necessary. Note that 12 pixels can be accessed at a time with three 32-bit SRAMs, the memory space for 3 pixels is not used. In order to avoid this underutilization of memory space, the memory organization is modified to reduce the number of pixels for Gaussian filter to 17 pixels. In this case, the width of a block becomes 8 pixels, which can be accessed with two 32-bit SRAMs. Therefore, no space is wasted, and consequently, memory utilization is improved.

Each block memory is composed of two 32bit SRAMs, each of which can output 4 pixels at a time within a row. The source pixel buffer consists of three blocks of memory, composed of 6 SRAMs in total. After five SRAMs are loaded among the 6 SRAMs, Gaussian filtering is started. During these SRAMs are being read for filtering operations, the last one SRAM is used for load from the external memory. For these buffers, data are loaded, read for filtering, and discarded in a circular manner.

When a block from an input image is loaded, all pixels within the block are processed by the Gaussian filter. Since the Gaussian kernel is a separable 2D kernel, 1D filtering is performed in the horizontal and vertical directions, in sequence. The horizontal filtering is performed first and then its results are input to the vertical filter. Note that no internal memory is necessary to store the result of the horizontal filtering because the pixels are scanned along the vertical direction as shown in Fig. 4.

After Gaussian filtering operation is completed for the current location, a DoG pixel is generated by subtracting the result of the adjacent scale images. If the DoG is a maximum or minimum compared with the DoGs within a *3x3x3* window, the location is considered as a keypoint candidate. Thus, the DoG needs to be derived not only for all pixels in the block but also one pixel outside the block as shown in Fig. 6(a). Therefore, the Gaussian filtering operation as well as the DoG operation is carried out for all pixels in a region, which is bigger than the block by one additional row above the top and another one row below the bottom. In fact, these additional rows are processed twice for Gaussian filtering and DoG operations for the adjacent blocks along the vertical direction.

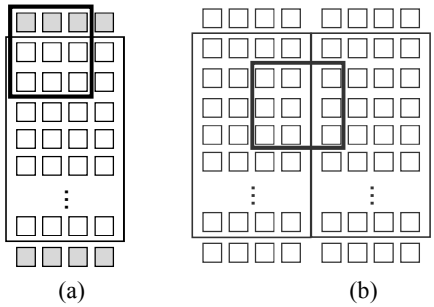


Fig. 6. DoG pixel reuse (a) Redundantly filtered DoG pixels, (b) Region of pixels stored in the DoG buffer.

Recall that pixels are filtered along the vertical direction. Thus, two DoG buffers with *(Block height+2)* length in each scale are used to temporarily store DoG values. Then, these values are used to find an extremum in a *3x3x3* window. These temporary buffers can also be used to store the DoG values, which are reused by the next right block. In Fig. 6(b), the shaded squares represent the pixels to be stored in these buffers.

For a Gaussian filter with W_{Gauss} length, W_{Gauss} X W_{Gauss} window is used for filtering operations. For filtering operations at the boundary of a block, the window includes pixels outside the block. Therefore, pixels in $(W_{Gauss}+1)/2$ lines are loaded twice for two adjacent blocks along the vertical direction. Let H_{Block} denote the height of a block. Then, the ratio of the lines accessed twice is given by

$$Bandwidth \quad Increase \quad Ratio = \frac{W_{Gauss}+1}{H_{Block}} \qquad (17)$$

This ratio decreases as the block height H_{Block} increases. The internal memory size for the block storage also increases in proportion to H_{Block}. Thus, the buffer size needs to be determined by considering a trade-off between the buffer size and the bandwidth requirement. In this paper, H_{Block} is chosen as 46, so that an SRAM for the source pixel buffer stores 64 words among which 9 words are above the block and the other 9 words are below the block.

V. Hardware Organization for Cost and Bandwidth Reductions

Bonato in [1] simplifies the SIFT algorithm by finding certain parts of computation which can be eliminated without significant drop of feature quality. 16 versions of simplified algorithms are proposed in [1]. This paper adopts the simplified version which eliminates 'Duplicated image' and 'Location refinement'. Three octaves are implemented as in [1]. Fig. 7 shows the hardware implementation, which is composed of four main function units: Gaussian filter bank, Keypoint

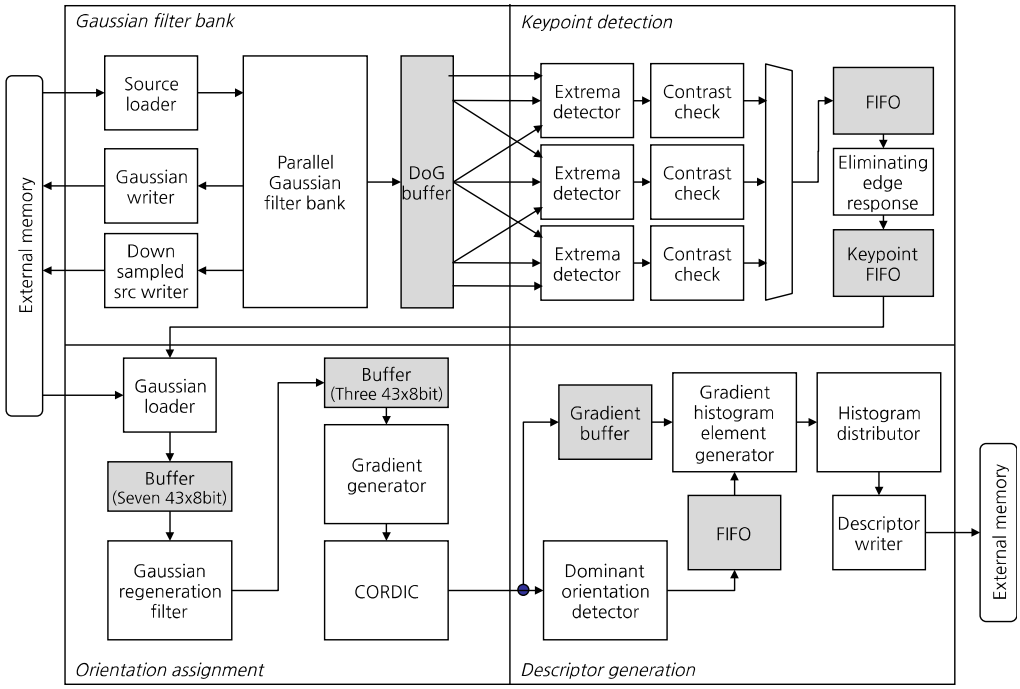


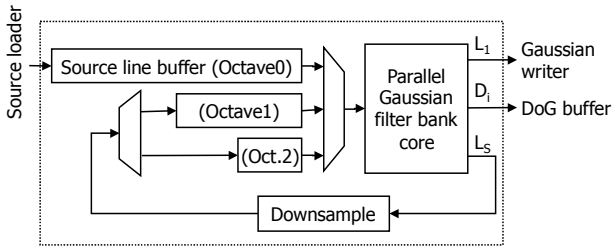
Fig. 7. The block diagram of the SIFT hardware implementation.

detection, Orientation assignment, and Descriptor generation blocks. The Keypoint detection block includes the local extrema detection explained in Section 2.2. Implementation issues of these blocks are described in this section.

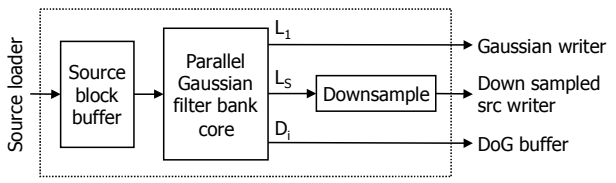
1. Hardware Cost Reduction with a Shared Gaussian Filter Bank Core

Fig. 8 shows the organization of the Parallel Gaussian filter bank. Fig. 8(a) represents the parallel Gaussian filter bank using three sets of the line memories to store the source images of the three octaves, respectively. Note that the filter bank core is shared by all three octaves. As a result, the hardware size for the filter bank is reduced by the shared filter core in the proposed organization when compared with the filter bank core in [1] which duplicates the filter bank core for every octave. The hardware share increases the computation for Gaussian filtering by about 1.3125 *(=1 + 1/4 + 1/16)* times. Nonetheless, the proposed implementation achieves the target performance (see Section 6).

(S+1)-th Gaussian-blurred image, $L_S(x,y)$, is down-sampled and then stored in the line memory. This image is used as the input image for the next octave. After all operations on a line for an octave are completed, operations on a line for the next octave are started as soon as the source line buffer becomes full. In this manner, the minimum number of the lines of source pixels are stored in internal memories without storage of down-sampled $L_S(x,y)$ into the external memory. The second Gaussian-blurred image, $L_1(x,y)$, is stored in the external memory to be used later by the orientation assignment block (see Section 5.3 for the reason of using this image).



(a) Parallel Gaussian filter bank with line memories



(b) Parallel Gaussian filter bank with block memories

Fig. 8. Parallel Gaussian filter bank organizations.

Fig. 8(b) shows the organization of the parallel Gaussian filter bank using block memories as the internal buffers. The *(S+1)*-th Gaussian-blurred image, $L_S(x,y)$, is down-sampled and then is stored in the external memory. The second Gaussian-blurred image, $L_1(x,y)$, is also stored to the external memory. As explained in Section 4.2, the DoG buffer is needed to store DoG pixels of two columns. This buffer is included inside the parallel Gaussian filter bank.

2. Balanced Keypoint Detection Block

In Fig. 7, the upper right part represents the organization of the Keypoint detection block. Three Extrema detection modules are followed by Contrast check modules run in parallel. The detected extrema are stored in a FIFO. As the number of pixels determined as an extrema is small, the Eliminating edge response module does not require large computing power. Thus, a single Eliminating edge response module is used to compute all detected extrema stored the FIFO. Finally, detected keypoints are stored in the keypoint FIFO and used by the Orientation assignment block.

3. Orientation Assignment Block with Reduced Memory Access

In this stage, gradient magnitudes and orientations are generated from the Gaussian-blurred image. As the gradients are generated from the same scale image as a keypoint, all scales of the Gaussian-blurred images need to be stored in an internal memory [2]. Nevertheless, the implementation in [1] stores only a single Gaussian-blurred image in an internal memory and produces gradient magnitude and orientation from the stored image of which scale can be different from a keypoint. As a result, the quality of the generated feature may be degraded although the internal memory is reduced.

This paper proposes a new organization that stores only a single Gaussian-blurred image without quality loss. For the case when the scale of the stored image is

different from that of a keypoint, the Gaussian-blurred image of the necessary scale is generated using the stored image by additional Gaussian filtering operation. To this end, the Gaussian-blurred image with scale index, *i=1*, is stored in the external memory, and then reloaded at the beginning of the Orientation assignment step.

In Fig. 7, the lower left part shows the structure of the Orientation assignment block. For the computation of a single keypoint, *43x43* pixels of a Gaussian-blurred image are loaded, and then stored into a buffer which consists of seven *43x8* bit memories. This buffer supplies a series of seven pixels in the vertical direction to the Gaussian filter at a time. In the case of the scale index, *i =1*, the filtering operation is not carried out, but in the case of *i=2* or *i=3*, the Gaussian-blurred image with the same scale of the keypoint is regenerated by the Gaussian filter. The pixels generated by the Gaussian filter are stored into a buffer with three 43x8 bit memories. This buffer provides the Gradient generation block with three pixels in the vertical direction. The Gradient generation block conducts the operation denoted by Eq. (11) and the CORDIC block computes Eq. (12) using the CORDIC algorithm [1].

4. Descriptor Generation Block

In [1], descriptor generation is implemented in software whereas the remaining parts of SIFT operations are implemented in hardware. With this software implementation of descriptor generation, real-time SIFT is possible only when the number of keypoints in an image is small. For example, assuming that 0.05% of pixels are detected as keypoints, the computation performance of 91,000 cycles/(descriptor generation) is required to process a QVGA (320x240) size image at the speed of 30 frames per second with the operating clock frequency of 100 MHz [1]. The required computing power increases as the number of keypoints increases. It is observed that the ratio of keypoints ranges from 0.1 to 0.5% for ordinary images. In this case, the computation speed needs to be improved by ten times for the partitioned implementation in [1] to process real-time SIFT operation. Therefore, in this paper, the descriptor generation is also implemented in hardware.

In Fig. 7, the lower right part shows the organization of Descriptor generation block. Gradient samples, which consist of the gradient orientations and magnitudes generated by the Orientation assignment block, are fed to the Dominant orientation detector as well as the Gradient histogram element generator. The Dominant orientation detector produces a gradient orientation histogram, and derives the dominant orientations, which are stored into a FIFO. Recall that multiple dominant orientations can be found with respect to a single keypoint.

The Gradient histogram element generator rotates the local patch around the keypoint along the dominant orientation, and computes up to 8 histogram elements from a gradient sample in the patch according to its location, orientation and magnitude. The elements are fed to the Histogram distributor, which generates gradient histograms. All the completed histograms are concatenated to become the final descriptor, which is stored in the external memory.

VI. EVALUATION

1. Number of Multiplications and Memory size

Fig. 9 compares the cascade Gaussian filter and the parallel Gaussian filter in terms of the number of multiplications and the line memory size. It is assumed that the Gaussian filtering operations are conducted with $\sigma_{in}=0.5$ and the line memories are composed of dual port SRAMs. Various values of σ_o are used for comparison. Fig. 9(a) shows that the parallel filter reduces the size of the line memory by more than 80%. Fig. 9(b) compares the number of multiplications for the computation for octave 0. As shown in the figure, the number of multiplications is increased by 48.5% for σ_o=1.4, which is the largest increase among all values of σ_o. 9(c) shows the number of multipliers for three octaves. In the proposed design, this number is the same as that for octave 0 only because the Gaussian filter bank core is shared by three octaves. On the other hand, the previous design duplicates the filter bank core so that the number of multipliers is increased by more than 170%. As a result, the proposed design requires about 50% of multipliers for all three octaves when compared with the previous design.

2. Logic Gate Count

The SIFT algorithm is implemented in hardware as

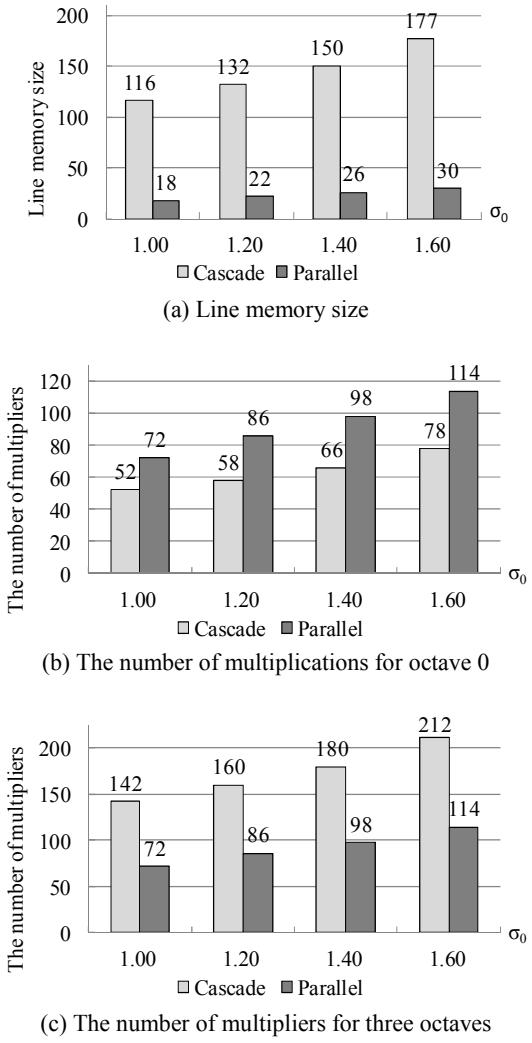


Fig. 9. Comparison of the hardware costs of parallel and cascade Gaussian filter banks.

shown in Fig. 7. The Verilog HDL is used for the hardware model, which is synthesized targeted for an FPGA (Altera STRATIX II, EP2S60F672C3). Table 1 shows the synthesis results of the four hardware blocks, Gaussian filter bank, Keypoint detector, Orientation assignment, and Descriptor generation. Each block is synthesized independently and the results are compared with those presented in [1].

The Gaussian filter bank does not include any DSP blocks. For the ESB bits, which refer to the size of the internal memory, the proposed design reduces them by 98.0% for the implementation of the Gaussian filter bank. This is caused by the proposed technique for memory size reduction. The number of registers and LUTs are also decreased by 58.6% and 36.6%, respectively, because of the shared filter core for 3 octaves. The hardware cost of the Keypoint detector in the proposed design is also decreased because it omits Location refinement operation, which is not essential in the SIFT algorithm. The hardware cost of the orientation assignment is increased because it includes an additional Gaussian filter.

Table 2 presents the synthesis results for the entire hardware. Again, the results presented in [1] are given in this table for comparison. The number of DSP blocks is the same as that given in [1]. On the other hand, the ESB bits, registers, and LUTs are reduced by 94.4%, 70.0%, and 61.2%, respectively.

The maximum operating clock frequency of the proposed design is 70.3 MHz, which is given by the synthesis report. For the four blocks are synthesized independently, the maximum frequencies for Gaussian filter bank, Keypoint detector, and Orientation assignment blocks are 91.8 MHz, 130.0 MHz, 93.4 MHz, and 137.1 MHz, respectively. These frequencies are not compared with those in [1] which does not present the maximum operating clock frequency, but does the supplied clock frequency which varies from 2.3 MHz to 100 MHz.

3. Processing Time

The processing speed is estimated by simulation with the implementation in Verilog HDL and the estimated speed is compared with the previous implementation as shown Table 3.

Table 1. FPGA synthesis results of the three subblocks

	Previous design [1]				Proposed design				Comparison			
	DSP blocks	ESB bits	Register	LUT	DSP blocks	ESB bits	Register	LUT	DSP blocks	ESB bits	Register	LUT
Gaussian filter bank	0	910,000	7,256	15,137	0	17,784	3,003	9,594	-	-98.0%	-58.6%	-36.6%
Keypoint detector	6	200,000	2,094	14,357	1	49,152	1,033	2,536	-83.3%	-75.4%	-50.7%	-82.3%
Orientation assignment	0	30,000	670	1,863	0	6,000	1,505	5,294	-	-80.0%	124.6%	184.2%
Descriptor generation	-	-	-	-	0	49,544	4,130	11,648	-	-	-	-

Table 2. FPGA synthesis results of the proposed SIFT hardware

	DSP blocks	ESB bits	Registers	LUT
Previous design [1]	8	1,350,000	19,100	43,366
Proposed design	8	75,240	5,729	16,832
Comparison	0.0%	-94.4%	-70.0%	-61.2%

Table 3. Maximum keypoint ratio for real time operation

Frame size	FPS	Proposed (50 HMz)	Bonato (50 MHz)
320x240	30	0.719%	0.024%
640x480	30	0.181%	0.006%

On average, it takes about 3,017 cycles to generate a keypoint with its descriptor. Thus, the proposed implementation allows QVGA-size image with about 0.719% (=[50 Mcycles/sec)/[(320 x 240 pixels/frame) x (30 frames/sec) x (3,017 cycles/keypoints)]) ratio to be processed at the speed of 30 frames per second. On the other hand, the implementation in [1] allows only 0.024% keypoint ratio with the speed of 91,000 cycles/keypoints. For VGA-size image, 0.181% ratio is allowed by the proposed implementation whereas 0.006% is allowed by the previous work.

VII. CONCLUSIONS

In this paper, a novel hardware architecture and organization for the SIFT algorithm is proposed and its implementation in Verilog HDL is presented. The architecture adopts parallel Gaussian filter bank and processes pixels in the block-by-block order. As a consequence, the memory size is reduced by 94.4%. The parallel Gaussian filter bank requires more multipliers than the previous work increasing the computational complexity. Furthermore, the Gaussian regeneration filter also increases the computation complexity. However, the overall hardware logic size is decreased because only a single Gaussian filter bank is shared by all octaves.

ACKNOWLEDGMENTS

This work was supported by the Industrial Strategic technology development program funded by the Ministry of Knowledge Economy (MKE, Korea). (10039188, Development of multimedia convergence programmable platform for smart vehicles)

REFERENCES

[1] V. Bonato, E. Marques, and G.A. Constantinides, “A Parallel Hardware Architecture for Scale and Rotation Invariant Feature Detection,” *IEEE Trans. on Circuits and Syst. Video Technol*ogy, vol. 18, no. 12, pp. 1703-1712, Dec. 2008.

[2] D. Lowe, “Distinctive image features from scale-invariant keypoints,” *Int. Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Jan. 2004.

[3] M. Grabner, H. Grabner, and H. Bischof, "Fast approximated SIFT," *ACCV 2006, LNCS 3851*, pp. 918–927, 2006.

[4] Y. Ke and R. Sukthankar, “PCA-SIFT: A more distinctive representation for local image descriptors,” *in Proc. IEEE CVPR*, pp. 506-513, Washington, USA, 2004.

[5] K.G. Derpanis, E.T.H. Leung, and M. Sizintsev, “Fast Scale-Space Feature Representations by Generalized Integral Images,” in *Proc. IEEE ICIP*, San Antonio, USA, 2007, pp. 521-524

[6] T. Lindeberg, "Scale-space for discrete signals," *IEEE Trans. On Pattern Analysis and Machine Intelligence,* vol. 12, no. 3, pp. 234-254, Apr. 1990.

[7] R. Hess. SIFT Feature Detector (Source Code). Available: http://blogs.oregonstate.edu/hess/code/sift/

[8] J.R. Magnus and H. Neudecker, *Matrix Differential Calculus with Applications in Statistics and Econometrics, 2nd edition*, Wiley, 1999.

[9] D.B. Williams and V. Madisetti, *Digital Signal Processing Handbook, 1st edition*, CRC Press, 1997.

Eung Sup Kim received the B.S. and M.S. degrees in Electronic Engineering from Soongsil University, Seoul, Korea, in 2004 and 2006, respectively, and the Ph.D. degree in Electronics Engineering from Seoul National University, Korea, 2012.

His research interests are in the area of SoC design for multimedia applications including computing for video compression and image analysis.

Hyuk-Jae Lee received the B.S. and M.S. degrees in Electronics Engineering from Seoul National University, Korea, in 1987 and 1989, respectively, and the Ph.D. degree in Electrical and Computer Engineering from Purdue University at West Lafayette, Indiana, in 1996. From 1998 to 2001, he worked at the Sever and Workstation Chipset Division of Intel Corporation in Hillsboro, Oregon as a senior component design engineer. From 1996 to 1998, he was on the faculty of the Department of Computer Science of Louisiana Tech University at Ruston, Louisiana. In 2001, he joined the School of Electrical Engineering and Computer Science at Seoul National University, Korea, where he is currently working as a professor. He is a founder of Mamurian Design, Inc., a fabless SoC design house for mobile multimedia applications. His research interests are in the areas of computer architecture and SoC design for multimedia applications.