Dang Ba Khac Trieu · Tsutomu Maruyama

Real-time color image segmentation based on Mean Shift Algorithm using an FPGA

Received: date / Revised: date

Abstract Image segmentation is one of the most important tasks in the image processing, and mean shift algorithm is often used for color image segmentation because of its high quality. The computational cost of the mean shift algorithm, however, is high, and it is difficult to realize its real time processing on microprocessors, though many techniques for reducing the cost have been researched. In this paper, we describe an FPGA system for the image segmentation based on the mean shift algorithm. In the image segmentation based on the mean shift algorithm, the image is once over-segmented, and then the small regions are merged considering the similarity between the over-segmented regions in order to obtain better segmentation. In our system, the mean shift filter is accelerated using a cache memory which can access to all pixels in a $w_s \times w_s$ pixel window at arbitrary position. This cache memory allows us to process $w_s \times w_s$ pixels in parallel every clock cycle. The region merging is also accelerated by not strictly managing the list structures used for the merging. This loose management introduces the redundant and out-of-date data into the list structures, but it makes the pointer dereferences unnecessary, and the overhead by those data can be hidden by pipeline processing. The performance for 768 × 512 pixel images is fast enough for real-time applications.

Keywords Mean shift algorithm · Segmentation · Region merging · Real time · FPGA

1 Introduction

Image segmentation is the process of partitioning an image into multiple regions of interest, and it is generally the first task in many automated image understanding applications. Many algorithms for the image segmentation have been proposed to date, such as [9][10][11][**?**][**?**]. Several hardware systems have been proposed to accelerate those algorithms, such as [14][15][12][13].

Mean shift algorithm is a procedure which is often used for color image segmentation because of its high quality. In the segmentation based on mean shift algorithm, first, the gradient of each pixel in a given image is calculated using a window centered by the pixel, and then, the pixel is moved along the gradient. This procedure is repeated until no movement will happen, and pixels which reach to the same bottom form a region. In general, many small regions are generated in this step (called over-segmentation). Then, the small regions are merged considering the similarity between the regions in order to obtain better segmentation (if we try to reduce the number of the regions by only the mean-shift, the results are far from what we expected in most cases). This over-segmentation and merging method is also used in other segmentation techniques such as watershed, K-mean clustering and so on.

The computational cost of a naive implementation of a mean shift algorithm is very high ($O(X \times Y \times w_s \times w_s \times N_r)$, where $X \times Y$ is the image size, $w_s \times w_s$ is the window size and N_r is the average number of the repetitions). Therefore, many technique for reducing the computational complexity have been researched (for example, [4][5][6]). However, it is still difficult to realize real-time processing on microprocessors. Nevertheless, only few papers about the acceleration of the mean shift algorithm by FPGA and GPU have reported to date because of the irregular memory access sequences required for tracing the movement of pixels along their gradients. In [7][8], the mean shift algorithm was used for object tracking, but the algorithm was applied only to the regions of interest. As for the region merging, to the best of our knowledge, no hardware systems have been proposed (in [17], a method for the labeling was proposed) probably because of the inherent sequentially and complex data management (this means that we can not ex-

D. B. K. Trieu · T. Maruyama
Systems and Information Engineering, University of Tsukuba,
1-1-1 Ten-nou-dai, Tsukuba, Ibaraki 305-8573, Japan
E-mail: maruyama@darwin.esys.tsukuba.ac.jp

D. B. K. Trieu
E-mail: dangtrieu@darwin.esys.tsukuba.ac.jp

pect high performance gain). However, this merging step should also be executed on the same platform in order to achieve higher performance in total.

In this paper, we describe FPGA implementation of both steps, over-segmentation using a mean shift algorithm and the region merging. The main difficulties in the segmentation are (1) how to read $w_s \times w_s$ pixels at arbitrary position in the image in parallel for tracing the movement of the pixels, and (2) how to fulfill the pipeline circuit (the pipeline depth for calculating the gradient is deep, and we need to interleave the calculation of several pixels), and the difficulties in the region merging are (3) the dynamic management of the lists which hold the contiguous regions (those regions are gradually changed as the merging progresses), and (4) the sorting of the pairs of two contiguous regions which are used to choose the two regions to be merged.

For (1) and (2), we have designed a special cache memory which can access to $w_s \times w_s$ pixels at arbitrary position in parallel. Typically, w_s is 15 to 31. Thus, this cache memory reads out 225 to 961 pixels of different position every clock cycle. In our implementation, the given image is scanned from top to bottom, and the pixels on L lines are cached into this cache memory. According to our experiments, most images can be processed during one scan (when a pixel moves out of the cached region toward the already processed area, another scan from bottom to top is executed). As for (3) and (4), we have chosen not to manage the data in the list structures strictly. By this loose management, redundant and out-of-date data may continue to stay in the data structures, and increases the amount of the computation. However, this relaxation makes it possible to manage those data in block without pointer dereferences (all data in the block can be accessed continuously), and the increase of the computation time by those data can be hidden by pipeline processing.

This paper is organized as follows. The mean shift algorithms are introduced in Section 2, and its FPGA implementation is described in Section 3. In Section 4, an algorithm for the region merging is introduced and its implementation is described in Section 5. Experimental results are shown in Section 6, and the conclusions are given in Section 7.

2 Mean Shift Algorithm

Mean shift analysis is a novel and powerful clustering approach originally reported in [1]. In spite of its excellent performance, it had been nearly forgotten until [2] extended it and introduced it to the image analysis community. In recent years, comprehensive analysis and successful applications of the mean shift occurred in the fields of tracking, image segmentation, information fusion, edge detection, clustering and classification, and video processing [4].

2.1 Original Mean Shift Algorithm

First, we briefly review the mean shift algorithm introduced in [3] mathematically. Given n data points $\mathbf{x}_i, i = 1, 2, ..., n$ in the d-dimensional space R^d, the kernel density estimator with kernel function $K(\mathbf{x})$ and a symmetric fixed bandwidth h can be written as

$$\hat{f}_h, K(\mathbf{x}) = \frac{c_{k,d}}{nh^d} \sum_{i=1}^{n} k(\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{h}) \tag{1}$$

where $k(\mathbf{x})$ is the *profile* of kernel K such that $K(\mathbf{x}) = c_{k,d}k(\|\mathbf{x}\|^2)$, and $c_{k,d}$ is a normalization constant. When the derivative of $k(\mathbf{x})$ exits, $g(\mathbf{x}) = -k'(\mathbf{x})$ can be used as a *profile* to define a new kernel $G(\mathbf{x}) = c_{g,d}g(\|\mathbf{x}\|^2)$ with normalization constant $c_{g,d}$. Take the gradient of (1), we can obtain

$$m_{h,G}(\mathbf{x}) = C\frac{\hat{\nabla} f_{h,K}(\mathbf{x})}{\hat{f}_{h,G}(\mathbf{x})} \tag{2}$$

$C(= \frac{1}{2}h^2c)$ is a constant. $m_{h,G}(\mathbf{x})$ can be rewritten

$$m_{h,G}(\mathbf{x}) = \frac{\Sigma_{i=1}^{n} \mathbf{x}_i g(\|\frac{\mathbf{x}-\mathbf{x}_i}{h}\|^2)}{\Sigma_{i=1}^{n} g(\|\frac{\mathbf{x}-\mathbf{x}_i}{h}\|^2)} - \mathbf{x} \tag{3}$$

and is called the *mean shift vector*. The expression (2) shows that, at location $\mathbf{x}$, the mean shift vector computed with kernel G is proportional to the normalized density gradient estimate obtained with kernel K. The mean shift vector thus always points toward the direction of maximum increase in the density. As the result, the mean shift iteration

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + m_{h,G}(\mathbf{x}^{(k)}), k = 1, 2, ... \tag{4}$$

is a hill climbing process to the nearest maximum of $\hat{f}_{h,K}(\mathbf{x})$.

2.2 Mean Shift Algorithm Implemented in Our System

An improved mean shift algorithm for the joint spatial-range domain is proposed in [3]. We implemented this algorithm in our system, because of its high performance, and the simplicity. Let r_c be $r(x, y)$ ($r(x, y)$ is a pixel in a given image), $x_c = x$, $y_c = y$ and $w_s = 2w+1$. Then, r_n, dx_n and dy_n are calculated as follows. In the equations below, LUV color space is used, and h_r is a constant which gives the threshold for choosing pixels close to r_c.

$$m = \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} h(r(x_c + dx, y_c + dy), r_c)$$

$$h(p, q) = \begin{cases} 1 & \text{if } |p_L - q_L| < h_r \,\&\, |p_U - q_U| < h_r \,\&\, |p_V - q_V| < h_r \\ 0 & \text{otherwise} \end{cases}$$

$$r_n = \frac{1}{m} \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} (r(x_c + dx, y_c + dy) - r_c) \times h(r(x_c + dx, y_c + dy), r_c)$$

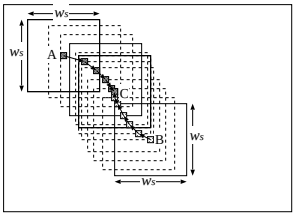

Fig. 1 Moves of the pixels by mean-shift

$$dx_n = \frac{1}{m} \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} dx \times h(r(x_c + dx, y_c + dy), r_c)$$

$$dy_n = \frac{1}{m} \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} dy \times h(r(x_c + dx, y_c + dy), r_c)$$

If $|dx_n| < \delta$ and $|dy_n| < \delta$, $p(x, y)$ is considered to converge to (x_c, y_c). Otherwise,

$r_c \leftarrow r_n$ & $x_c \leftarrow x_c + dx_n$ & $y_c \leftarrow y_c + dy_n$

and the steps above are repeated from the beginning. In the steps above, first, the number of pixels (m) which are similar to r_c is counted. Then, the average of those pixels (r_n) and the movement of the coordinate (dx_n and dy_n) are calculated. If the movement is very small, it is considered that the pixel converges to (x_c, y_c), and otherwise, the movement is repeatedly calculated by assigning r_n, x_c+dx_n, y_c+dy_n to r_c, x_c, y_c respectively.

By applying this procedure, pixels in the image are moved along to their gradients, and converged to one of the bottoms of the gradients. Fig.1 shows an example how the pixels are moved. In Fig.1, pixel (A) and (B) are moved to the same bottom (C), and are considered to belong the same region.

The computational cost of this algorithm is $O(X \times Y \times w_s \times w_s \times N_r)$ (N_r is the average number of the repetitions).

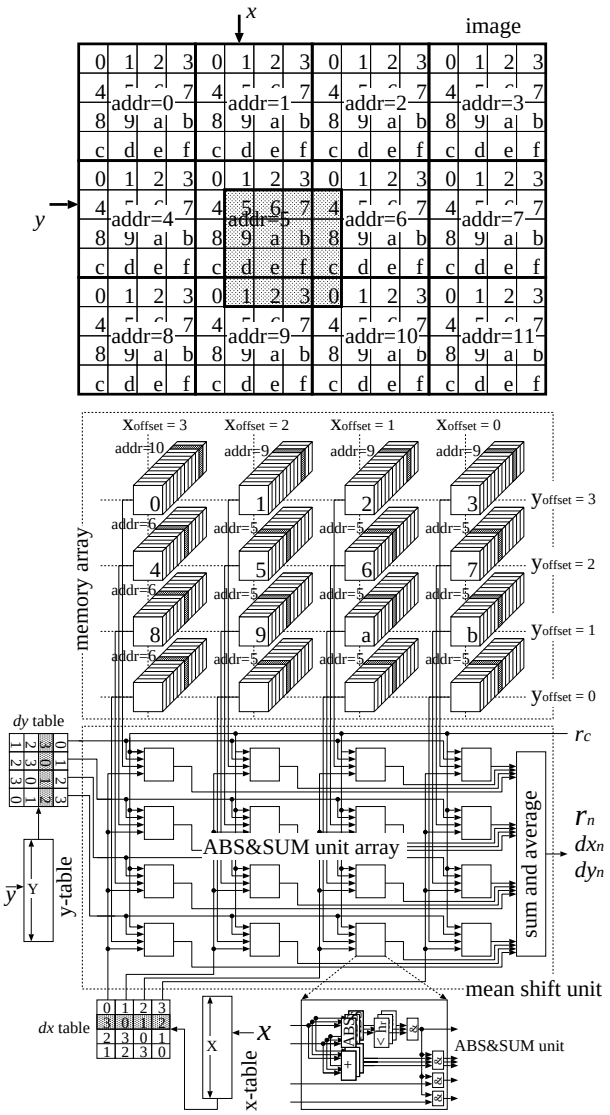

Fig. 2 The computation method

3 Implementation Method of Mean Shift Algorithm

The computation of the mean shift algorithm described in Section 2.2 can be summarized as follows.

1. $(x_c, y_c) \leftarrow (x, y)$ and $r_c \leftarrow r(x, y)$.
2. For each pixel at (x_c, y_c), read out $w_s \times w_s$ pixels centered by (x_c, y_c) in parallel, and count the number of pixels similar to r_c.
3. At the same time, calculate
   (a) the sum of LUV of those pixels, and
   (b) the sum of their coordinates
   and calculate the average of them (r_n and (dx_n, dy_n)).
4. check the convergence, and repeat the steps above if not converged by assigning r_n and (x_c+dx_n, y_c+y_n) to r_c and (x_c, y_c).

For processing these steps efficiently, we need a data mapping method which enables the parallel access to $w_s \times w_s$ pixels centered by any (x, y).

3.1 Data mapping method

Fig.2 shows a data mapping method used in our system. In Fig.2, w_s=4 and all pixels in the image are mapped on a memory array which consists of $w_s \times w_s$ memory banks. Pixels labeled 'k' (k = '0' − 'f') in the image are stored in the memory banks labeled 'k' ('0'−'f'), and $w_s \times w_s$ pixels (each being labeled one of '0'−'f') can be accessed in parallel. In this mapping method, the first $w_s \times w_s$ pixels on the upper-left corner of the image are placed on the same plain ($addr$=0) of the memory array, the next $w_s \times w_s$ pixels are on the next plain ($addr$=1) and

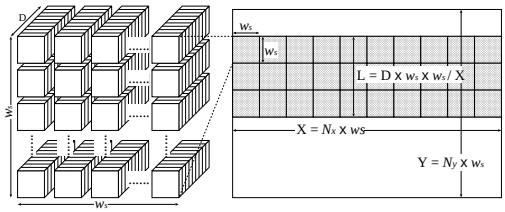

Fig. 3 Actual mapping of the image on the memory array

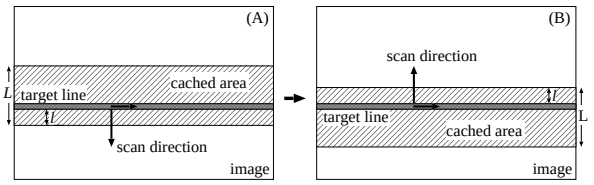

Fig. 4 scanning method

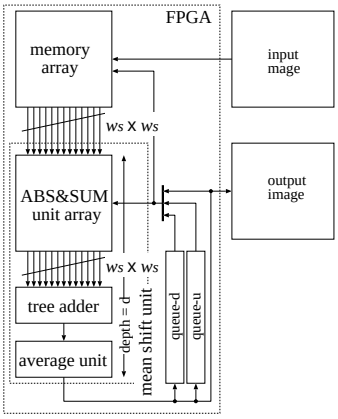

Fig. 5 A block diagram of our system

so on. Suppose that we are going to read $w_s \times w_s$ pixels from (x, y) (the gray box in the image) (here, $w_s \times w_s$ pixels whose upper left corner is (x, y) are used in order to simplify the explanation). This box lies down on four plains ($addr$=5, 6, 9, 10). To each of the $w_s \times w_s$ memory banks, different addresses are given as follows.

$addr = \lfloor (y + y_{\text{offset}})/w_s \rfloor \times w_s + \lfloor (x + x_{\text{offset}})/w_s \rfloor$

x_{offset} and y_{offset} are constants given to each column and row in advance. Then, the pixels in the gray box are read out from the memory banks using the different addresses.

r_n, dx_n and dy_n are calculated on the mean shift unit. In this computation, the positions of the pixels that appear on the memory array are different from those in the gray box in the image. In order to calculate dx_n and dy_n correctly, we need to give the true dx and dy to the mean shift unit. For this purpose, two kinds of tables are used (dx and dy table, and x and y table). There are only w_s patterns of dx and dy (the range of dx and dy is $[0, w_s-1]$). Therefore, the patterns are stored in dx and dy table (their depth is w_s), and these tables are accessed via x and y table whose depth is X and Y.

In an actual implementation, w_s is an odd number (w_s= 2w+1), and the coordinate of the center pixel is given. In this case, $x-w$ and $y-w$ are given to the circuit instead of x and y, and the range of dx and dy is changed to $[-w, +w]$. Then, dx=$-w-1$ and dy=$-w-1$ are used to mask the data on the memory array. With this data mapping method, we can apply the mean shift to any pixel in the image with a simple circuit.

3.2 Actual data mapping on FPGA

The size of on-chip memory banks is, however, not so large in practice, and we can not store the pixels of whole image. Therefore, pixels of only L lines can be store in the memory array (Fig.3). When the depth of each memory banks of the memory array is D, L becomes $D \times w_s \times w_s/X$. In this case, the processing order of the pixels in the image becomes very important. In our implementation, the image is scanned twice (or once if all pixels converged during the first scan) as shown in Fig.4. In Fig.4(A), the pixels are scanned from top to bottom. When, the pixels of first l+1 lines are read into FPGA, the mean shift filter is applied to the pixels on y=0. Then, the pixels on the next lines are read from the image, and the filter is applied to the pixels on y=1. This sequence is repeated, and pixels of up to L lines are cached on the FPGA. The mean shift filter continues to be applied on the pixel on lth line in the L lines (Fig.4(A)).

Three cases happen during the computation.

1. the pixel converges to the bottom in the L lines,
2. the pixel moves upward, and goes out of the L lines,
3. the pixel moves downward, and goes out of the L lines.

When the second case happens, the pixel (r_n, (x_c+dx_n, y_c+dy_n) and its original coordinate) is put in *queue-u*, and the processing of the pixel is suspended. After scanning the image to the bottom, the scan is restarted from the bottom. When the target line of this scan reaches to y_c+dy_c, the processing of the pixel is resumed. This is because the pixel will continue to move upward with high probability.

When the third case happens, the pixel is put in *queue-d*, and the processing is suspended. The processing of the pixel is resumed when the target line reaches to y_c+dy_n+$L/2$. This is because the pixel will continue to move downward, and if we resume the processing of the pixel immediately, the pixel will move out of the L lines again.

3.3 System architecture

Another problem of this implementation is the long delay caused by the long feedback in the mean shift unit. In

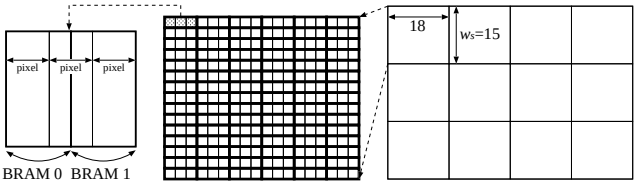

Fig. 6 A mapping to reduce block RAMs

the mean shift filter, we need to sum up values of $w_s \times w_s$ pixels to calculate r_n, dx_n and dy_n, and the depth of the mean shift unit becomes deeper than $\log(w_s \times w_s)$.

Fig.5 shows a block diagram of our circuit. Suppose that pixels of l+1 lines have been buffered in the memory array. Then, the pixel at $(0, 0)$ is processed first by the mean shift unit. $w_s \times w_s$ pixels centered by $(0, 0)$ are given to the ABS&SUM unit array (pixels out of the image are masked in the ABS&SUM unit array), and r_n, dx_n and dy_n for the pixel at $(0, 0)$ are calculated. In this case, r_c (namely, the pixel at $(0, 0)$) is given from the memory array to the mean shift unit. It takes d ($d > \log(w_s \times w_s)$) clock cycles to calculate them. In order to hide this long latency, the computation of the next pixel (the pixel at $(0, 1)$) is started immediately. Then, pixels to $(0, d-1)$ are processed continuously. At clock cycle d, r_n, dx_n and dy_n for the pixel at $(0, 0)$ come out of the average unit, and they are given to the ABS&SUM unit array and the memory array (to read out $w_s \times w_s$ pixels centered by $(0+dx_n, 0+dy_n)$) to trace the movement of the pixel at $(0, 0)$. At the next clock cycle, r_n, dx_n and dy_n for the pixel at $(0, 1)$ come out of the average unit, and is processed in the same way. In this way, d pixels are continuously processed in the mean shift unit. When one of them converges, or moves out of the L lines (it is put in one of the queues), the pixel at $(0, d)$ is newly processed. By repeating this sequence, the mean shift unit can always be filled by d pixels. The pixels in the queues are processed before starting the calculation of the pixels on the target line.

3.4 An method to reduce the memory size

The problem of the implementation described in the previous section is the large number of the on-chip memory banks used for the memory array. When we consider to implement the circuit on Xilinx FPGAs, 15×15 block RAMs are necessary when w_s=15, and 961 when w_s=31.

Block RAMs in Xilinx FPGAs can be configured as $512 \times 36b$. Therefore, by using 2 block RAMs, we can store pixels of three columns (the data width of a pixel is $24b$). In this case, however, three pixels on the two banks have to be read as one set. As shown in Fig.6, w_s is expanded to a multiple of three (18 when w_s=15, and 36 when w_s=31).

Fig.7 shows how to access $w_s \times w_s$ pixels with this implementation method. Suppose that we are going to read 15×15 pixels centered by (x, y). These pixels lie on four

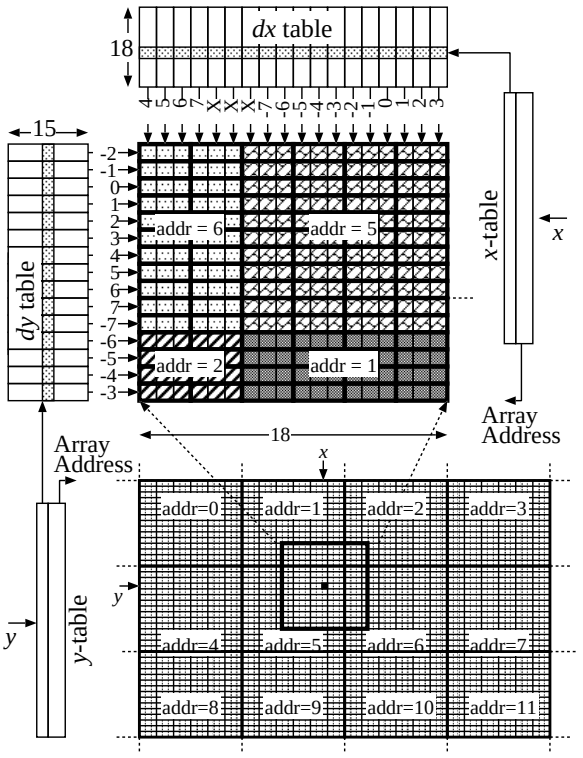

Fig. 7 A mapping to reduce block RAMs (1)

memory plains of the memory array ($addr$=1, 2, 5, 6), and three pixels in the array have to be read as one set. Therefore, 18×15 pixels are read out from the memory as shown in Fig.7, and three columns are masked using dx table (in Fig.7, X shows the mask, and other values show dx). In order to give the addresses to the memory array, we need to divide x by 18 and y by 15. In order to avoid this divide operation, the addresses are stored in x and y tables as well as the addresses to dx and dy tables.

Fig.8 shows the actual data mapping on the memory array. The memory array consists of 6×15 block RAMs when w_s=15. The first 6 column of each memory plain are stored in the first 2×15 block RAMs (the first three columns are stored in the first half of the block RAMs, and the next three column (gray parts in Fig.8) are stored in the second half), and these 6 columns are read out in parallel using the dual read of block RAMs. Other columns are also grouped by 6 columns, and stored in the block RAMs in the same way. With this implementation, we can not update the data in the memory array, while the pixels on the target line are processed accessing the memory array. In order to reduce the time for updating data in the memory array, shallow buffers are provided, and the pixels of the next line are downloaded to these buffers, while the pixels on the target line are processed. Then, the mean shift unit is stopped (it is still filled by d pixels while it is stopped), and the data in the

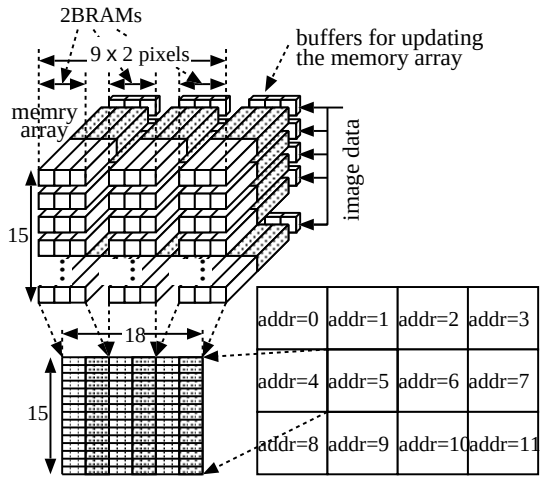

Fig. 8 A mapping to reduce block RAMs (2)

buffers are moved to the memory array in parallel. The time for updating the memory array is very small (only $X/(90 \times 1.5)$ clock cycles per line when $w_s = 15$).

4 Region merging

In the image segmentation based on mean-shift algorithm, the image is once divided into many small regions (over-segmentation), and then, the small regions are gradually merged to obtain better segmentation. Many region merging algorithms have been proposed[18][19][20]. The merging algorithms can be summarized as follows.

1. For each region labeled 'u', make a list $l_c('u')$ which holds the regions contiguous to 'u'.
2. List up all pairs of regions ('u', 'v') which are contiguous each other.
3. Calculate the distance between the two regions in the pairs using a given function f.
4. Choose the pair which gives the minimum distance, and merge the two regions.
5. Update l_c of the merged regions.
6. Repeat step 2 to 5 while the minimum distance is smaller than a given threshold.

In most software algorithms, a global functions is defined, and the distances between two regions are calculated considering the global balance among the pairs. In some cases, this distance calculation may become the bottle-neck of the merging, and not be able to implement on hardware systems easily. In this paper, in order to give a general framework, we focus on the other steps than the distance calculation, and we consider a simple function f which calculates the distances using only local information.

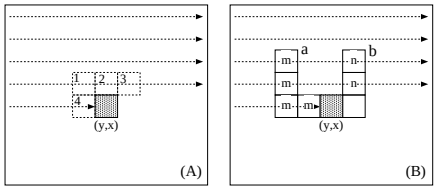

Fig. 9 A method for giving unique labels to the regions

5 Our approach of region merging

In our implementation described in Section 3, the outputs by the mean shift unit are color images in which the number of colors are reduced from the original image. Therefore, first, we need to give a unique label to the pixels which are contiguous each other and have same color.

Our approach consists of the four steps below.

1. The image is scanned, and the unique labels are given to the regions (a region consists of the contiguous pixels of the same color). The regions which consist of only one pixel are put aside, and are processed in the last step.
2. The image is scanned again, and for each region 'u', the list which holds their neighbor regions $l_c('u')$ is generated. At the same time, the distances between the two contiguous regions are calculated, and the pairs of the two regions are sorted according to their distances.
3. Two regions are repeatedly merged according to their distances. $l_c('u')$ is updated when 'u' and 'v' ('u' < 'v') are merged.
4. Regions which consist of only one pixel are merged to their larger neighbor regions.

5.1 The first scan

First, we need to give a unique label to each region. In our approach, the image is scanned from $(0, 0)$ to $(Y–1, X–1)$. In this scanning, the color of the pixel at (y, x) $(I(y, x))$ is compared with the color of its four neighbors as shown in Fig.9(A). If the color of $I(y, x)$ is different from its all four neighbors, a new region label is given to $I(y, x)$. If the same as one of its neighbors, its region label is copied to $I(y, x)$ (in Fig.9(A), if $I(y, x)$ has the same color as the pixel 2, the region label of the pixel 2 is copied to $I(y, x)$). Then, the region label of $I(y, x)$ is output to the off-chip memory bank. During this first scan, the regions which consists of only one pixels are also detected, and their addresses $((y, x))$ are stored in the queue in the off-chip memory bank. According to our experiments, the number of those one-pixel regions is 1% to 13% of the total pixels in the image when w_s=15 or 31 for all tested benchmark images. In our current implementation, these regions are left unprocessed in the queue, and after other

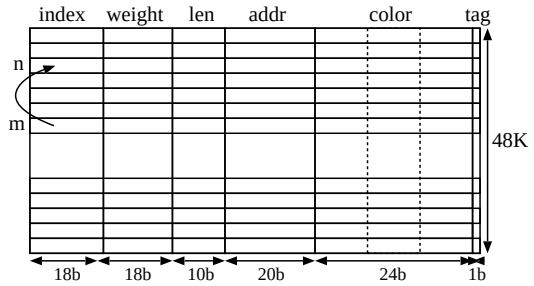

Fig. 10 The region table

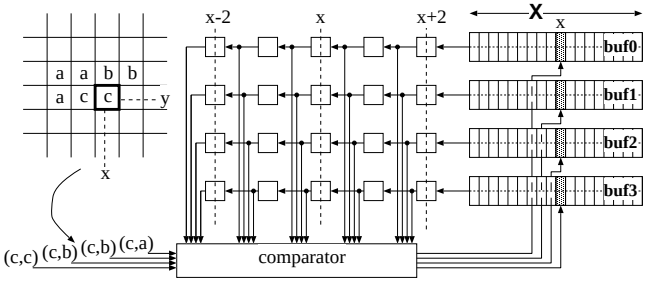

Fig. 11 The label pair comparator

regions are merged, the one-pixel regions are merged to them, in order to improve the performance of the system. It may seem that 1-13% of the image is not negligible, and those one-pixel regions should be merged from the beginning. However, those regions usually exist among larger regions (namely, around the edges of the objects in the image), and do not have the significant affect on the final result.

Suppose that there is a u-shaped region in the image as shown in Fig.9(B). Then, the region label 'm' is give to the pixel 'a', and 'n' is given to 'b', because their color is not same as their four neighbors, though 'a' and 'b' are parts of the same region. When comparing the color of $I(y, x)$ with its four neighbors, the color of the pixel at '1' and '3', and '4' and '3' (see Fig.9(A)) are also compared, and if they are equal, they are recognized to belong to the same region. In Fig.9(B), when comparing the color of $I(y, x)$, it is found that the label 'm' and 'n' were given to the same region. The region table shown in Fig.10 is used to record the equality among the region labels. This table has seven fields, and is used to store other in formations about the regions.

weight & *color* The *weight* field shows the number of the pixels which belong to the region. When a new region label is assigned to a pixel, the *weight* is set to one, and the color of the pixel is copied to the *color* field.

index When we notice that two labels ('m' > 'n') were given to the same region, a pointer is set in the *index* field of 'm' in order to record that 'm' and 'n' are the same region as shown in Fig.10. The weight of 'm' are added to the weight of 'n'.

addr & *len* The *addr* field is used to hold the address of a block which stores the labels of the regions which are contiguous to this region. In the first scan, the *len* field is used to estimate the maximum number of the regions which are contiguous to the region (namely, the length of $l_c('u')$). Then, in the second scan, it shows how many labels has been stored in the block pointed by *addr*.

tag The *tag* is used to remove the redundancy in l_c.

When the first image scan is finished, the *index* in the region table is scanned from the top, and the pointers are dereferenced. Suppose that a pointer from 'c' to 'b' ('c' > 'b') was linked first, and then another pointer from 'b' to 'a' ('b' > 'a') during the image scan. This means that 'c', 'b' and 'a' are given to the same region. During the scan of the region table, 'a' is tested first, because the scan started from the top, and 'a' is written into the *index* of 'a', because 'a' did not point anywhere. Then, when testing 'b' (or 'c'),

1. the *index* of 'b' (or 'c') is read out (in this case 'a' (or 'b')),
2. using the read-out value ('a' (or 'b')), the *index* of 'a' (or 'b') is read out ('a' (or 'a')), and
3. the read-out value ('a' (or 'a')) are set into the *index* of 'b' (or 'c').

With this dereference, every *index* directly points to the true region label.

During the image scan, if the label given to $I(y, x)$ is different from the labels of its four neighbors, the *len* of the label is counted up in order to estimate the length of $l_c('u')$. Suppose that the labels of its four neighbors are 'a', 'a, 'b' and 'c', and the label of $I(y, x)$ is 'c', the *len* is counted up by 2 ('c' is contiguous to 'a' and 'b'). However, if we repeat this counting up on every pixel, the *len* is counted up many times by the same pairs of the labels. For example, if $I(y, x)$ and $I(y, x + 1)$ belong to the same region labeled 'c', and they are contiguous to 'a', then, the *len* of 'c' is counted up twice. In order to prevent this redundant counting up strictly, we need to manage $l_c('u')$ strictly using list structures. However, this strict management of l_c is not easy on hardware systems. In stead, we have used the unit shown in Fig.11 to prevent the redundant counting up as much as possible. In Fig.11, suppose that the label of $I(y, x)$ was set to 'c', because its color is the same as $I(y, x{-}1)$. Then, $I(y, x)$ is contiguous to 'a' and 'b', and the *len* of 'c' has to be counted up by 2. However, $I(y, x{-}1)$ is also contiguous to 'a' and 'b', and the *len* of 'c' has already counted up for 'a' and 'b'. In Fig.11, in order to prevent this redundant counting up, four buffers and the comparator unit is used. Here, we denote a pair of the two contiguous regions as ('a','b'). When the *len* of 'c' is counted up by ('c','a') at (y–1, x), it is recorded at x–1 of one of the four buffers. When ('c','a') is given from the upper-left pixel, it is written into buf0. In the same way, buf1,2,3 are used

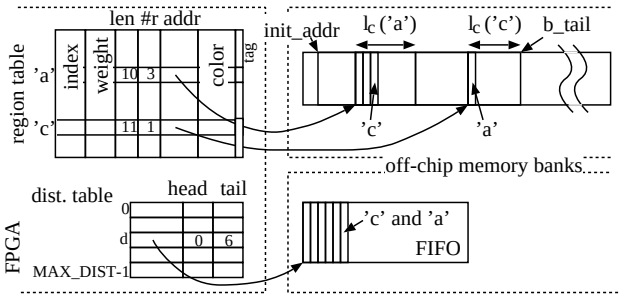

Fig. 12 Generating the list l_c

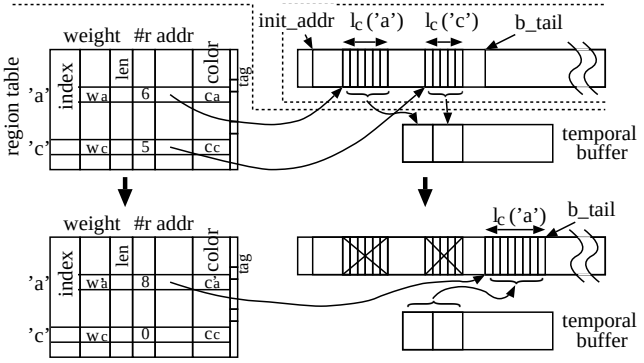

Fig. 13 Merging regions

for the upper, upper-right and left pixels. These pairs are kept in the buffers until they are overwritten. The pairs in the four buffers are read out onto the register array (its size is 5×4, and the pairs which were detected at $x-2$ to $x+2$ are held on it), and are compared with the four new pairs in the comparator unit in parallel. In Fig.11, the four pairs ('c','a'),('c',b'),('c',b') and ('c','c') are compared with the 20 pairs on the shift registers (('c','c') is discarded immediately). If a new pair is equal to one of the 20 pairs on the registers, the new pair is discarded. The four new pairs are also compared each other, and the redundant pairs are discarded. According to our experiments, the average length of l_c is 7-10, and about 20% longer than when strictly managed. This is short enough for our purpose. When two regions ('a' < 'b') are merged, the *len* of 'b' is added to that of 'a' as well as the *weight*. Note that the length of l_c is evaluated in this phase, but l_c itself is not constructed.

5.2 The second scan

In the first scan, the region label of (y, x) are stored in the off-chip memory bank. In the second scan, those labels are read back, and dereferenced using the region table in order to obtain the true labels. Then, the unit shown in Fig.11 is used again to detect which region is contiguous to which region. Suppose that

1. the pair of two contiguous regions ('c','a') is detected,
2. this is the first pair for 'c', and the third for 'a', and
3. *b_tail* is an address to the other off-chip memory bank, which is initialized to init_addr > 0.

Then, the following operations are applied to 'c' and 'a' (Fig.12 shows what happens by the operations).

1. Because this is the first pair for 'c' (this can be checked if its *addr* is zero or not), its *addr* is set to *b_tail*, and *b_tail* is incremented by its *len*+δ (δ is a margin for the estimated length of l_c, because in some cases, it can not be estimated correctly in the first scan). By allocating a block of *len*+δ words in advance, we can store all contiguous regions to 'c' in this block. Then, 'a' is stored in the off-chip memory bank using its *addr* as the address of the off-chip memory bank, and its *len* is reset to one.
2. As for 'a', 'c' is stored in the off-chip memory bank using its *addr*+*len* as the address, and *len* is incremented.
3. At the same time, the distance d between 'c' and 'a' is calculated, and if the distance is smaller than a given threshold MAX_DIST, the pair ('c' and 'a') is sorted in the queue pointed by d. The queue are managed using their heads and tails in the dist.table.

As described in the previous subsection, the pairs of the contiguous regions are not managed strictly. Therefore, some same pairs may be stored in the queue of the same distance.

5.3 Region merging

In this step, first, the dist.table is scanned from $d_{\min}$ (the current minimum distance), and a pair with the minimum distance is read out from its queue. Suppose that the pair is ('c','a') and 'a' < 'c'. Then, the following operations are applied to 'a' and 'c' (Fig.13 show what happens by the operations).

1. The weight of 'c' (w_c) is updated to $w_a^{'} = w_a + w_c$
2. The color of 'a' is updated to $c_a^{'} = (c_a \times w_a + c_c \times w_c)/(w_a + w_c)$
3. The weight of 'c' is changed to zero, which means that this region is discarded (out-of-date).
4. Each region label in $l_c(\text{'a'})$ and $l_c(\text{'c'})$ is read out sequentially from the off-chip memory bank, and written into the temporal buffer,
   (a) if the region label is not 'a', not 'c', and not out-of-date (its weight is not zero), and
   (b) if the *tag* of the region is zero.
   Otherwise, the region label is discarded. When the region label is copied into the buffer, its *tag* is set to one. The *tag* is used to prevent to copy the same region more than once.
5. When the region label 'u' is copied into the buffer, the distance d between 'a' (its color has been updated)

and 'u' is recalculated. In this case, the pair ('a', 'u') may already be in one of the queues according to the distance which were calculated using the previous color of 'a'. Normally, the pair ('a','u') should be removed from the queue, and put in queue[d] which corresponds to the new distance d. However, in our approach, the old ('a','u') is not removed, and the new ('a','u') is just put into the queue[d]. This means that two ('a','u') exist in two different queues (or in the same queue if the color of 'a' is not changed by this merging).

6. The *len* of 'a' is set to the number of the regions which are copied into the temporal buffer (8 in Fig.13).
7. A new block whose size is the *len* of 'a' is newly allocated in the off-chip memory bank, and the *addr* of 'a' points the block. Then, the region labels in the temporal buffer is written back into the block. During this copying, the *tag* of the regions are reset to zero.
8. Then, the dist.table is scanned from $d_{\min}$ again, and a pair with the minimum distance is read out from one of the queues. As described above, the out-of-data pairs are not removed from the queues in our approach. Suppose that the pair ('a','b') is given from queue[d]. In order to verify that the pair is valid one,
   (a) the *weight* of 'a' and 'b' is checked if they are zero or not, and
   (b) the distance between 'a' and 'b' is recalculated, and checked if it becomes d.

   If the pair is not valid, it is discarded, and the next candidate is read out from the queues. In our implementation, several candidates are read out, and they are checked if they are valid or not on the pipelined unit. Then, the merging procedure is repeated from the step 1.

The two blocks which were used to hold $l_c('a')$ and $l_c('c')$ become unnecessary when they are merged, but they are not garbage-collected in our approach. According to our experiments, their total size is less than 512K words, and can be easily stored in an off-chip memory bank.

The merging procedures above are repeated while pairs whose distance is less than MAX_DIST exist.

5.4 Region merging of the small regions

After merging regions whose size is greater than one, the small regions which consists of only one pixel are merged to their neighbor larger regions. First, the address of a small region is read out from the buffer in the off-chip memory bank. Then, its color is read out from the image data in another off-chip memory bank. At the same time, the region label of its eight neighbors are read out from the region label array (label '0' is given to the small regions during the first scan) sequentially, and the distance to it is calculated on the pipelined unit. The colors of the neighbors are given from the region table. Then, the closest neighbor is chosen, and the small label is merged to the neighbor.

6 Experimental results

We have implemented the circuits for the mean-shift algorithm and the region merging on Xilinx XC4VLX160 on RC2000-4 FPGA board. The circuit for the mean-shift algorithm uses 23.3 KLUTs and 96 block RAMs when w_s=15 (90 as the memory array). When w_s=31, the circuit size becomes almost fourfold, and the number of block RAMs required for the memory array becomes 372. The circuit for the region merging uses about 9.5K LUTs. 92, 183, and 274 block RAMs are used when the size of the region table (N_{rt}) is $16K \times 1$, $16K \times 2$, and $16K \times 3$ (one block RAM is used for dist.table). Because of the limitation of the number of block RAMs, the combination of the circuits which can be implemented on one XC4VLX160 is limited. The possible combinations are $w_s = 15$ and $N_{rt} \leq 32K$. The performance of the filter circuit for $w_s = 31$ was evaluated using a software simulator, and that of the merging circuit for N_{rt}=$48K$ was evaluated separately from the filter circuit. The operational frequency of the two circuits is 138.8MHz and 166.6 MHz respectively, and 138.8MHz was used for the performance evaluation.

Table 1 shows the performance of the circuits for four bench mark images[21]. 'average' and 'max' show how long each pixel is moved by the mean shift filter until it stops, and N_r shows how many times the mean shift is applied to each pixel in average (the performance is almost proportional to N_r). 'in' and 'out' show the number of the regions before and after the region merging, and 'execution time' shows the ratio spent in the four steps described in Section 5. 'fps_1' and 'fps_2' show the processing speed of the mean shift filter and the region merging, and 'fps_3' shows the total performance when the two circuits are applied sequentially to the images.

In all tested cases, all pixels converge during the first scan of the mean shift filter. Table 2 shows the number of the lines (L) which can be stored in the memory array. L becomes smaller as X becomes larger, because the number of the block RAMs is kept constant. When X = 768 and w_s=15, L is 75 (l is 5), and this is larger than the maximum move along the y axis (40). When w_s=31, the maximum becomes 80, but L=372 in this case.

'fps_2' becomes almost half by enlarging w_s, because N_r becomes larger, but the number of the regions generated by the filter ('in') becomes smaller, and 'fps_2' becomes faster. As the result, 'fps_3' is fast enough for real-time applications when the image size is not larger than 768×512.

Fig.14 shows the original, mean-shifted, and region-merged images ($w_s = 31$) for two benchmarks, *peppers* and *tulips*.

Table 1 Movement of the pixels and the performance of our circuit (when h_r=9)

	mean shift filter							region merging							total
		move_distance/pixel						#regions		execution time (%)					
	w_s	average		max		N_r	fps$_1$	in	out	1st	2nd	merge	small	fps$_2$	fps$_3$
		x	y	x	y										
peppers	15	1.46	1.37	31	28	2.71	193.9	26450	553	22.9	17.6	41.9	17.6	93.2	62.8
512 × 512	31	6.45	5.90	92	80	6.14	85.6	15773	289	25.3	21.5	34.2	19.0	113.6	48.8
tulips	15	2.17	2.14	30	40	3.49	100.5	42907	773	23.0	17.3	42.4	17.3	61.1	38.0
768 × 512	31	7.62	7.55	71	64	6.94	50.4	25744	489	25.1	21.0	34.5	19.4	74.0	30.0
monarch	15	1.42	1.50	32	35	2.70	129.6	34393	705	24.3	19.2	37.3	19.2	67.9	44.6
768 × 512	31	5.48	5.43	89	62	5.51	63.6	22517	430	26.9	23.0	33.2	16.9	81.1	35.6
serrano	15	2.00	1.98	39	34	3.30	83.6	21322	540	36.2	32.1	28.6	3.1	89.2	43.2
629 × 794	31	7.01	6.66	70	65	6.87	40.2	12840	444	42.1	39.1	17.3	1.5	108.6	29.3

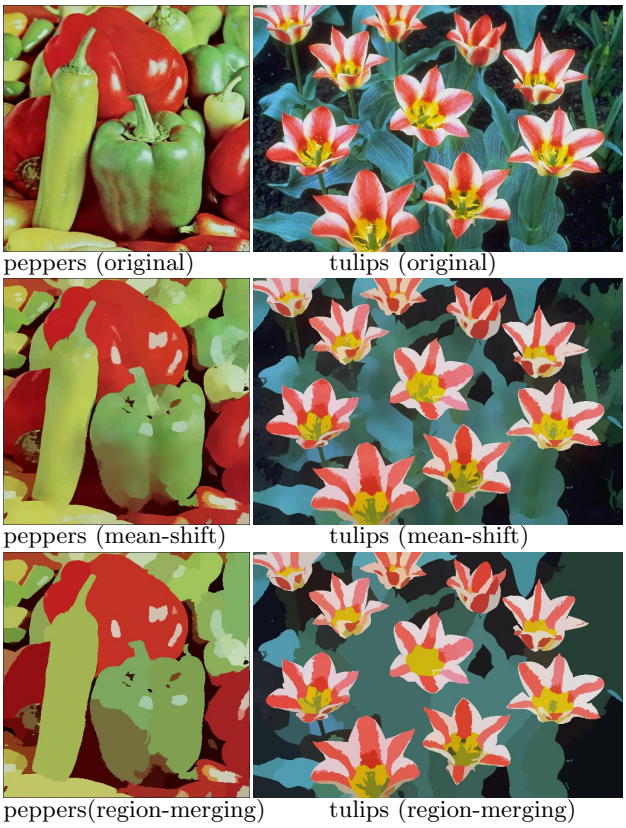

Fig. 14 the original, mean-shifted, and merged images

Table 2 The number of the lines cached on FPGA

X	640	720	768	1024	1920
w_s=15	105	90	75	60	30
w_s=31	465	372	372	279	124

We need an FPGA with 400 block RAMs (the block RAMs can be shared between the two circuits, though in this experiment, we did not share them for the design simplicity) for achieving all combination of w_s and N_{rt} on one FPGA. Recent FPGAs have more than 1000 block RAMs, and are large enough.

7 Conclusions

In this paper, we have described an image segmentation method on FPGA using the mean shift algorithm. In the image segmentation based on the mean shift algorithm, the image is once over-segmented, and then the small regions are merged. By caching L lines of the given image, and by processing the pixels in them in proper order, we can apply the mean shift filter to all pixels in the image efficiently. The region merging is an inherently sequential process, and the acceleration by hardware systems is not easy. We have shown that we can achieve real-time processing by relaxing the data management and enabling the pipeline processing of the data. Our implementation requires a number of block RAMs (about 400), but recent FPGAs support more number of block RAMs, and our implementation is feasible for those FPGAs.

In our current implementation, the distance between two regions is calculated using only local relation of the two regions. We need to calculate the distance by considering the global relation of the regions in order to obtain better segmentation. That is our main future work.

References

1. K. Fukunaga and L.D. Hostetler: The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition. IEEE Transactions on Information Theory, vol. 21, pp. 32-40 (1975)
2. Yizong Cheng: Mean Shift, Mode Seeking, and Clustering. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 17, no. 8 (1995)
3. Dorin Comaniciu and Peter Meer: Mean Shift: A Robust Approach Toward Feature Space Analysis. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, no. 5, pp. 603-619 (2002)
4. Huimin Guo, Ping Guo, and Hanqing Lu: Fast Mean Shift Procedure with New Iteration Strategy and Re-sampling. IEEE International Conference on Systems, Man and Cybernetics, pp.2385-2389 (2006)
5. Zhiming Qian, Changren Zhu and Runsheng Wang: An Improved Fast Mean Shift Algorithm for Segmentation. International Conference on Computer Application and System Modeling, pp.116-120 (2010)
6. K. Bitsakos, C. Fermuller and Y. Aloimonos: An Experimental Study of Color-Based Segmentation Algorithms

Based on the Mean-Shift Concept. European conference on Computer vision, pp.506-519 (2010)
7. Peihua Li and Lijuan Xiao: Mean Shift Parallel Tracking on GPU. Iberian Conference on Pattern Recognition and Image Analysis (2009)
8. U. Ali and M. B. Malik: Hardware/software co-design of a real-time kernel based tracking system. Journal of Systems Architecture, vol. 56, pp. 317–326 (2010)
9. W. Ma and B. Manjunath: Edge flow: A framework of boundary detection and image segmentation. Computer Vision and Pattern Recognition, pp. 744–749 (1997)
10. J. Shi and J. Malik: Normalized cuts and image segmentation. Computer Vision and Pattern Recognition, 1997, pp.731–737 (1997)
11. S. Zhu and A. Yuille: Region competition: unifying snakes,region growing, and bayes/mdl for multiband image segmentation. Pattern Analysis and Machine Intelligence, vol. 18, no. 9, pp. 884–900 (2002)
12. K. Appiah, A. Hunter, P. Dickinson, and H. Meng: Accelerated hardware video object segmentation: From foreground detection to connected components labeling. Computer Vision and Image Understanding, vol. 114:11, pp. 1282–1291 (2010)
13. P. Dillinger, J. Vogelbruch, J. Leinen, S. Suslov, R. Patzak,H. Winkler, and K. Schwan: Fpga-based real-time image segmentation for medical systems and data processing. IEEE Transactions on Nuclear Science, vol. 53, no. 4, pp.2097–2101 (2010)
14. T. Saegusa and T. Maruyama: An fpga implementation of real-time k-means clustering for color images. Journal of Real-Time Image Processing, vol. 2:4, pp. 309–318 (2007)
15. D. B. K. Trieu and T. Maruyama: Real-time image segmentation based on a parallel and pipelined watershed algorithm. Journal of Real-Time Image Processing, vol. 2:4, pp. 319–329 (2007)
16. D. B. K. Trieu and T. Maruyama: An implementation of the mean shift filter on fpga. International Conference on Field Programmable and Applications,pp. 219–224 (2010)
17. Christian Schmidt and Andreas Koch: Fast Region Labeling on the Reconfigurable Platform ACE-V. International conference on Field Programmable Logic and Applications, 2003.
18. L. G. Ugarriza, E. Saber, S. R. Vantaram, V. Amuso,M. Shaw, and R. Bhaskar: Automatic image segmentation by dynamic region growth and multiresolution merging. IEEE TRANSACTIONS ON IMAGE PROCESSING, vol. 18, no. 10,pp. 2275–2288 (2009)
19. Q. Luo and T. M. Khoshgoftaar: Efficient image segmentation by mean shift clustering and mdl-guided region merging. International Conference on Tools with Artificial Intelligence,pp. 337–343 (2004)
20. J. Stawiaski and E. Decenciere: Region merging via graphcuts. Image Analysis and Stereology, vol. 27, no. 1, pp. 39–45 (2008)
21. http://links.uwaterloo.ca/Repository.html