IMPLEMENTATION OF A PARALLEL AND PIPELINED WATERSHED ALGORITHM ON FPGA

Dang Ba Khac Trieu and Tsutomu Maruyama

Systems and Information Engineering, University of Tsukuba
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 JAPAN
dangtrieu@darwin.esys.tsukuba.ac.jp

ABSTRACT

This paper describes an implementation of a parallel and pipelined watershed algorithm on FPGA. In the algorithm, pixels in a given image are repeatedly scanned from top-left to bottom-right, and then from bottom-right to top-left. Because of these simplified memory accesses, N pixels in a given image can be processed in parallel by reading N lines at the same time. However, N is limited by the number of external memory banks that store image data. In our implementation, in order to achieve high performance using an FPGA with limited number of external memory banks, (1) a given image is divided to K regions, (2) several of them are cached on the FPGA, (3) the watershed algorithm is applied on those regions, and (4) the next (or previous) region is loaded to the FPGA during the computation to hide the loading time. In our current implementation on XC2V6000, up to 32 pixels can be processed in parallel. The performance for 512×512 pixel images is about 3 - 4 msec, which is fast enough for real-time applications.

1. INTRODUCTION

The watershed transformation[1][2] is a popular image segmentation technique for grey scale images, and many algorithms have been proposed. Among them, [3] and [4] show very good performance. In these algorithms, four raster scans of all pixels in a given image are major memory accesses, but extra memory accesses for FIFO and stack are required. These extra memory accesses make it difficult to achieve high performance using dedicated hardware architectures for these algorithms. By modifying the algorithms for pipeline processing, and simplifying the memory accesses (scanning pixels in a given image repeatedly from top-left to bottom-right, and then from bottom-right to top-left), it becomes possible to accelerate the performance using hardware systems. This approach also makes it possible to process N pixels in parallel by reading N lines at the same time. However, N is limited by the number of external memory banks that store image data, and we can not expect high performance on off-the-shelf FPGA boards with limited number of external memory banks.

In this paper, we describe an implementation technique of the watershed algorithm on FPGA. In the implementation, (1) a given image is divided to K regions, (2) several of them are cached on the FPGA, (3) the watershed algorithm is applied on those regions, and (4) the next (or previous) region is loaded to the FPGA during the computation to hide the loading time.

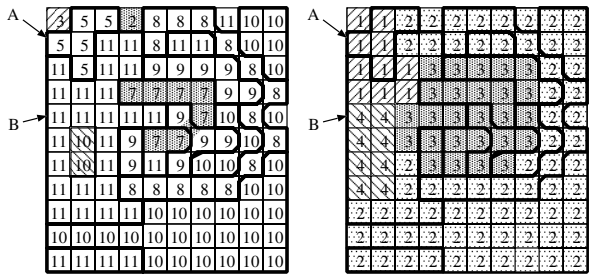
This paper is organized as follows. In Section 2, sequential watershed algorithms are introduced, and the parallel and the pipelined algorithm is described in Section 3. The implementation technique of the algorithm is discussed in Section 4, and its experimental results are shown in Section 5. In Section 6, conclusions and future works are given.

2. WATERSHED ALGORITHMS

In the traditional implementation of the watershed segmentation algorithm which simulates the flooding process, regional minima in a given image are detected, and unique labels are given to the minima first. Then, other pixels in the image are sorted according to their grey levels using a hierarchical queue[1][2]. The pixels in the queue are scanned from lower grey levels, and if labels have already been assigned to their neighbor pixels, the labels are copied to the pixels. The complexity of this algorithm is a little high because of sorting and managing the hierarchical queue, which makes efficient implementation on hardware difficult. In the watershed algorithm based on connected components[3] (called *BM*), each component (pixel) in the image is connected to its lowest neighbor component, and all components which leads to the same lowest component (local minima) forms a segment. This algorithm requires only a simple FIFO, and a stack (for recursive calls), and is much faster than the traditional implementation. In [4], another algorithm based on chain code was proposed (called *SYR*), and it was shown that the algorithm is a bit faster than *BM*. Steps in these two algorithms can be summarized as follows.

1. *step-A*: For each pixel in a given image, its lowest neighborhood is detected, and the pixel points the lowest (using its address in *BM* and chain code in *SYR*). If the pixel is on a plateau, no lower neighbors are found.

1-4244-0 312-X/06/$20.00 ©2006 IEEE.



(a) input image
minima and non-minima plateaus

(b) output image of labels

Fig. 1. A given image and its segmentation

2. *step-B*: For each pixel on plateaus, if one of its neighbors points somewhere (this means that this neighbor is on the edge between lower plateau, and points the lower plateau), the pixel points the neighbor. By repeating this procedure using a FIFO, pixels on plateaus point one of their lower plateaus. The FIFO is used to divide plateaus evenly according to the distances from their lower plateaus.
3. *step-C*: For each pixel which points nowhere (this means that the pixel is on local minima plateaus which are not labeled yet), a unique label is given. Then, its neighbor pixels on the same plateau point the pixel. By repeating this procedure using recursive calls (*BM*) or a stack (*SYR*), all pixels on the plateau points the pixel (namely the label) directly or indirectly.
4. *step-D*: Pointers for all pixels are dereferenced, and labels are given to all pixels.

Figure 1(a) shows an input image, and plateaus (surrounded by bold lines) and local minima (grey boxes or boxes with slanting lines) in the image. There are four local minima, thus four segments in the image. Figure 1(b) shows the four segments obtained by *BM* and our algorithm. In Figure 1(b), pixels on non-local minima plateaus (for example, plateaus A and B) are assigned to different segments according to the distance from their adjacent lower plateaus.

Watershed transformation is a powerful image segmentation technique. With dedicated hardware architectures, the potential of its real-time application can be realized. However, little work has been reported so far on hardware realization of watershed transformation. In [5][6][7], hardware implementations of flooding-based watershed algorithms were proposed, but their performance is much slower than the hardware implementation based on our algorithm.

3. THE PARALLEL AND PIPELINED ALGORITHM

Figure 2 shows the details of our algorithm for parallel and pipeline processing. In Figure 2, p means a pixel in a given image, and $f[p]$ is its grey level (after filtered by gradient filters and so on). $N(p)$ means neighbor pixels of p.

In our algorithm, two arrays ($l[p]$ and $v[p]$) are used. $l[p]$ is an array to store labels, while $v[p]$ is an array to store dis-

```
Input: f
Output: l
  v[p] ← 0, l[p] ← 0 // initialization
  current_label ← 1
for (;;) {
  Scan from top-left to bottom-right (p) {      // downward-scan
    step1(p) | step2(p) | step3(p)              // in pipeline
  }
  if (v[p] or l[p] is not modified) exit
  Scan from bottom-right to top-left (p) {      // upward-scan
    step1(p) | step2(p) | step3(p)              // in pipeline
  }
  if (v[p] or l[p] is not modified) exit
}
step1(p){          // edges of non-local minima plateaus
 if (v[p] ≠ 1)
   for each (p' ∈ N(p))
     if (f[p'] < f[p]) v[p] ← 1
}
step2(p){          // the inside of non-local minima plateaus
 if (v[p] ≠ 1) {
   min ← VMAX;
   for each (p' ∈ N(p) and f[p'] = f[p] and v[p'] > 0)
     if (v[p'] < min) min ← v[p']
   if (min ≠ VMAX and v[p] ≠ min + 1) v[p] ← min + 1
 }
}
step3(p){
 if (v[p] = 0) {          // local minima plateaus
   min ← LMAX;
   for each (p' ∈ N(p) and f[p'] = f[p] and l[p'] > 0)
     if (l[p'] < min) { min ← l[p']; q ← p; }
   if (l[p] = 0)
     l[p] ← (min ≠ LMAX)? l[q]: current_label++
   else if (min ≠ LMAX and l[q] < l[p])
     l[p] ← l[q]
 } else {
   if (v[p] = 1) {     // edges of non-local minima plateaus
     min ← FMAX; lmin ← LMAX;
     for each (p' ∈ N(p) and f[p'] < f[p])
       if (f[p'] < min or
           (f[p'] = min and l[p'] > 0 and l[p'] < lmin))
             { min ← f[p']; lmin ← l[p']; q ← p'; }
   } else {          // the inside of non-local minima plateaus
     min ← VMAX; lmin ← LMAX;
     for each (p' ∈ N(p) and f[p'] = f[p] and v[p'] > 0)
       if (v[p'] < min or
           (v[p'] = min and l[p'] > 0 and l[p'] < lmin))
             { min ← v[p']; lmin ← l[p']; q ← p'; }
   }
   if (lmin ≠ LMAX and l[q] ≠ l[p]) l[p] ← l[q] }
 }
}
```

Fig. 2. The proposed algorithm

tance from its lower pixels or plateaus. $v[p]$ is used instead of FIFO to divide non-minima plateaus evenly.

Memory accesses in our algorithm are simplified for parallel and pipeline processing. The only memory access sequences are repetition of sequential scanning of pixels from top-left to bottom-right (called *downward-scan*) and from bottom-right to top-left (called *upward-scan*). This repetition is finished when no $l[p]$ and $v[p]$ are modified in the current scanning. Values of $l[p]$ and $v[p]$ are delivered to right and downward in *downward-scan*, and to left and upward in *upward-scan*. Therefore, we need to repeat these two scanning in turn. Another feature is that only values on p are modified (values on $N(p)$ are not modified). These features make it easy to design a circuit for the algorithm. By preparing 3×3 units which hold $f[]$, $v[]$ and $l[]$, and compare them, values on p can be efficiently calculated.

$step1(p)$ in Figure 2 is almost same as *step-A* in the previous section, but $v[p]$ is set instead of linking a pointer. In *step2(p)*, $v[p]$ for pixels on non-minima plateaus is calculated. In this step, suppose that d is given to $v[p]$ in *downward-scan* (d is the distance from its lower plateau which is located upward). But, it may be overwritten in *upward-scan* if the distance from its lower plateau which is located downward is smaller than d. In $step3(p)$, labels are given to pixels on local minima plateaus, and those labels ($l[p]$) are delivered to its neighbors. In this step, a new label is assigned to each pixel whose $v[p]$ is zero if all its neighbor pixels with same grey level have no labels. This may cause

1. assignment of new labels to pixels on non-minima plateaus (if $v[p]$ is not calculated yet), and
2. assignment of different labels to pixels on same local minima plateau.

However, those labels are overwritten by correct labels in the subsequent scans. Another discussion point in this step is that $l[p]$ is calculated using values of different generations. For example, when calculating $l[p]$ in *downward-scan*, upward and left values of $v[p]$ are already updated in $step2(p)$, but downward and right values are not updated yet. By storing new $v[p]$ of three lines, and then calculating $l[p]$ using them, we can avoid this problem, but it makes no significant differences on performance according to our experiments.

Figure 3 shows how the segmentation of the image in Figure 1 progresses in our algorithm. In *scan1*, some $v[p]$ are not fixed yet, but all $v[p]$ are correctly fixed during *scan2*. In *scan1*, seven labels are used, but they are gradually replaced by correct labels (only four segments in this image) as the segmentation progresses. In *scan6*, no $v[p]$ and $l[p]$ are modified, and the segmentation steps are finished. Thus, for this image, we need 6 scans in total (*non-optimized scanning*). However, in *scan3*, only data on four rows (*r1, r7, r9* and *r11*) are modified (grey boxes). Therefore, in *scan4 (upward-scan)*, we need not to read data on *r4* (*row-optimized scanning*) unless data on previous lines are modified and delivered *upward*. Furthermore, as for the other rows, we need not to read data in the area with slanting lines (*column-optimized scanning*) unless data on those rows are modified and delivered to the area. In *scan4*, only three data on *r10* (at *c1, c2* and *c3*) are modified. Therefore, in *scan5*, we need not to read data on *r1*-*r7* (*row-optimized scanning*), and data with slanting lines on other rows (*column-optimized scanning*) (unless values on those rows are modified and delivered to *c5* and so on). As shown in this example, we can drastically reduce the amount of data which have to be scanned by memorizing which data (for which pixel) are modified in the previous scan.

4. IMPLEMENTATION OF THE ALGORITHM

As described in the previous section, we need $f[]$, $v[]$, $l[]$ to process each pixel on each line. Our target image size is not smaller than 512×512 or 640×480. Therefore, total data width for one pixel becomes 32b (8b for $f[]$, 8b for $v[]$, and 16b for $l[]$).

4.1. Without Cache Memory on FPGA

In our algorithm, we can process N pixels in parallel, by reading N lines at the same time. Figure 4(A) shows how to use the external memory banks in this parallel processing. All data ($f[]$, $v[]$ and $l[]$) are stored in external memory banks. First, $f[]$ in $m0$, $v[]$ in $m1$ and $l[]$ in $m2$ and $m3$ are read out, and processed by the pipelined circuit on the FPGA. The results (new $v[]$ and $l[]$ generated by the circuit) are stored into $m4$, $m5$ and $m6$ in parallel with the computation. Then, the role of the memory banks ($m1$,$m2$,$m3$ and $m4$,$m5$,$m6$) are exchanged. In this case, we need seven memory banks for processing four pixels in parallel. If the number of memory banks is limited as shown in Figure 4(B) (the number of memory banks is four), the performance becomes worse because we can not load/store $v[]$ and $l[]$ at the same time.

Figure 5 shows how each line is processed on the parallel and pipelined circuit. Four lines are processed in parallel in Figure 5. In this parallel processing, we need to wait 3 clock cycles before starting the computation of the next line in order to transfer correct values of $v[]$ and $l[]$ to the next lines. The total clock cycles to processes N lines becomes

$L + (N-1) \times 3$ where L is the pipeline depth.

4.2. With Cache Memory on FPGA

Figure 6 shows an approach to process more pixels in parallel under limited number of external memory banks. In this approach, a given image is divided to K regions along x (or y) axis, and several regions (r_i) are cached in cache memory banks c_i (block RAMs) on the FPGA (four cache memory banks in Figure 6). Then, the algorithm is applied to the regions cached in some of these cache memory banks (r_k and r_{k+1} on $c1$ and $c2$ in Figure 6), and the data for next region (r_{k+2}) are downloaded to $c3$ during this computation.

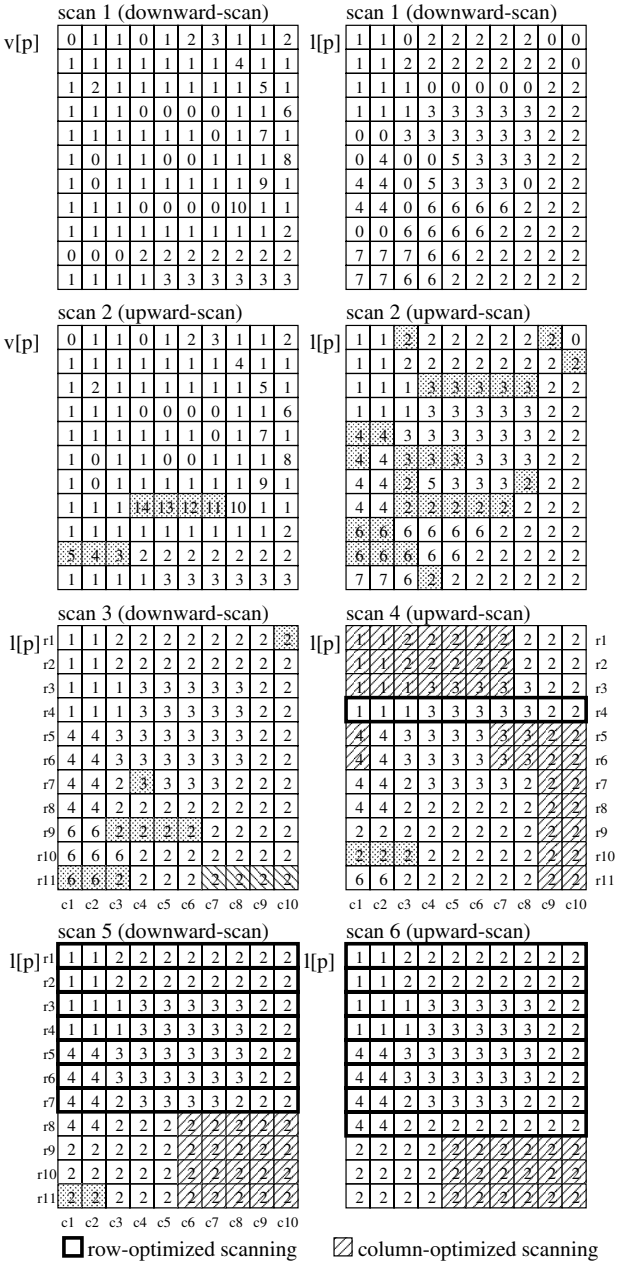


Fig. 3. The progress of the segmentation

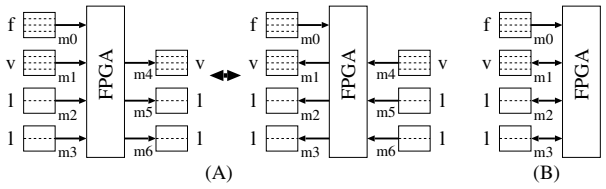


Fig. 4. Usage of external memory banks

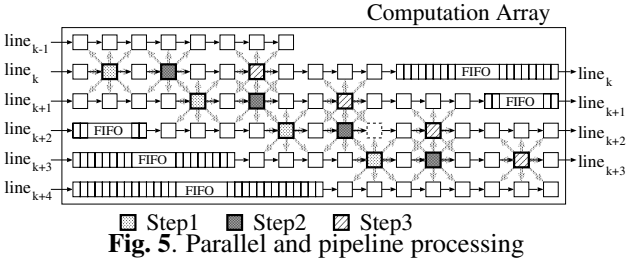


Fig. 5. Parallel and pipeline processing

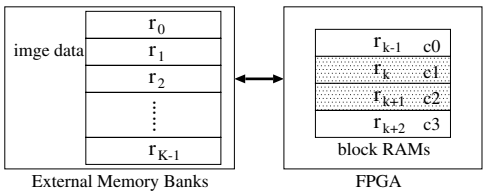


Fig. 6. Caching lines

Figure 7 shows how the computation progresses in this approach. First, data for two regions ($r0$ and $r1$) are downloaded to two cache memory banks ($c0$ and $c1$). Then, the watershed algorithm is applied to $r0$ and $r1$ (*Step1*). All lines in $r0$ and $r1$ can be processed in parallel. At the same time, data for $r2$ are downloaded to $c2$ (only $f[]$ is necessary, because $v[]$ and $l[]$ are not set yet, and initial values can be used). In *Step2*, the algorithm is applied to $r1$ and $r2$, and data for $r3$ are downloaded to $c3$. The algorithm is applied to $r1$ again in order to deliver $v[]$ and $l[]$ which are set in $r2$ to $r1$ efficiently. Data for $r0$ are kept in $c0$. In *Step3*, the algorithm is applied to $r2$ and $r3$, and data for $r4$ are downloaded to $c0$. In this step, data in $c0$, namely $v[]$ and $l[]$ for $r0$ are stored to external memory banks in parallel with this downloading. *Step4,5,6* are same as *Step3*. In the steps in Figure7(A), the input to FPGA is $f[]$, and the outputs are $v[]$ and $l[]$. Therefore, with four memory banks, we can continue the computation without stopping the pipelined circuit.

Suppose that $v[]$ or $l[]$ on upper edge of $r4$ are updated in Figure 7(B) *Step5*. This means that we need to deliver those values to $r3$. Therefore, $r3$ and $r4$ are processed again in *Step6a*. In this case, we don not need to download data for $r3$, because they are still held in $c3$. During this computation, data for $r2$ are downloaded to $c2$ discarding data in $c2$ ($f[]$ for $r6$). If $v[]$ or $l[]$ on upper edge of $r3$ are not modified, we do not need to deliver values in $r3$ to $r2$. Therefore, $r4$ and $r5$ are processed again using $r5$ in $c1$ (*Step7a*). In *Step8a*, $v[]$ or $l[]$ on upper edge of $r5$ are updated. Then, the algorithm is applied to $r4$ and $r5$ again in order to deliver those values to $r4$ (*Step9a*).

Suppose that $v[]$ or $l[]$ on upper edge of $r3$ are updated (Figure 7(C) *Step6a*) as well as *Step5*. Then, we need to trace back once more, and the algorithm is applied to $r2$ and $r3$ again to deliver those values to $r2$ (*Step7b*). In this step, we need to download data for $r1$ (not only $f[]$, but $v[]$ and $l[]$, because $v[]$ and $l[]$ for $r1$ are already calculated

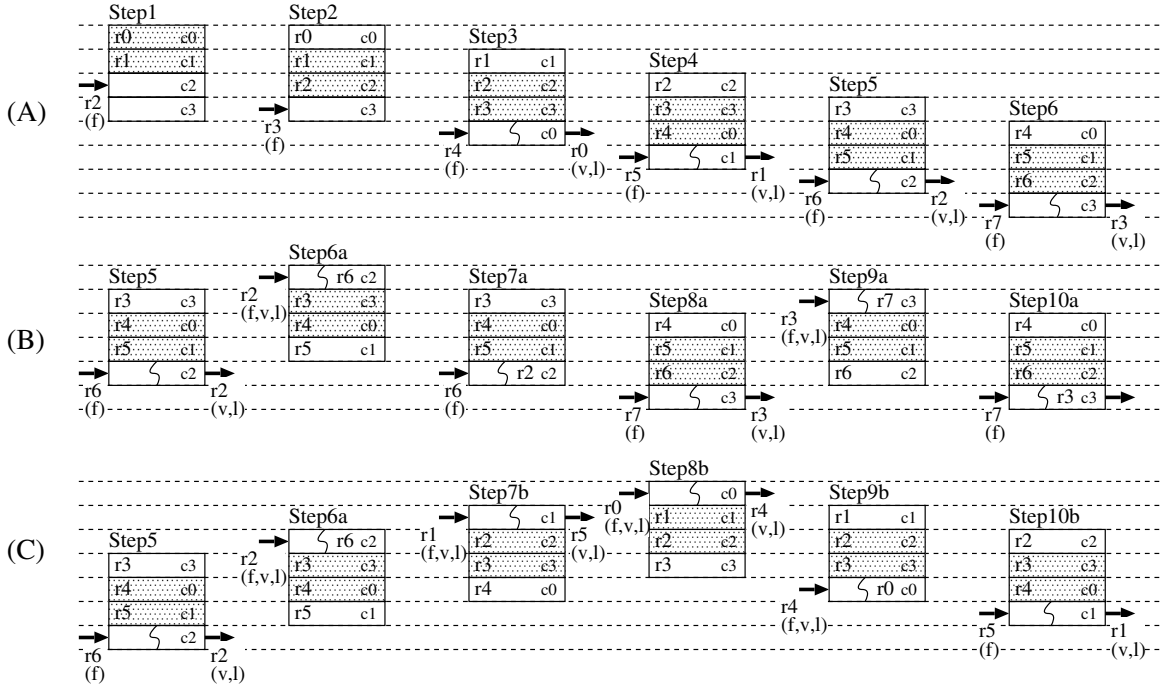


Fig. 7. Trace of the computation with cache memory on FPGA

and stored to external memory banks in *Step4*), and to store data for $r5$ ($v[]$ and $l[]$) at the same time. Under limited number of external memory banks, the circuit may become idle, because the download for $r1$ can not be started until data for $r5$ are stored. In *Step8b*, we need to load and store $v[]$ and $l[]$ at the same time as well as *Step7b*.

As described above, when we need to trace back more than once, we have to load and store $v[]$ and $f[]$ in same step, which degrades the performance because of the idle time caused by memory access conflicts. In order to reduce the number of trace back, we need to enlarge the number of lines which are processed in parallel. This is a trade-off between the performance and the circuit size.

5. EXPERIMENTAL RESULTS

We have implemented the algorithm on ADM-XRC-II (by Alpha Data) with one XC2V6000. Data width of $f[]$, $v[]$ and $l[]$ are 8b, 8b and 16b respectively (according to our experiments $v[p]$ is smaller than 100). Therefore, one line of the image can be cached with one block RAM (configured as 512 $\times$ 36b). The length of the line can not be longer than 512, which means that X or Y of the given image can not be larger than 512. XC2V6000 has 144 block RAMs, so we can cache up to 144 lines. In our current implementation, four cache memory banks with 32 line width are implemented on the FPGA (128 BRAMs are used), and 64 lines in two of them are processed by 32 units which run in parallel. Suppose that 64 lines in two regions (r_k and r_{k+1} on c_i and c_{i+1}) are being processed now. In *downward-scan*, 32 lines in r_k is processed in parallel first, and then, 32 lines in k_{k+1} is processed in parallel by the 32 units. In *upward-scan*, r_{k+1} is processed first, and then r_k. The circuit runs at 66 MHz (though the maximum frequency reported by ISE7.1i is 56.5 MHz), and occupies 92% of the hardware resources (slices), and 95% of BRAMs.

Taking the gradient image thresholded with an arbitrary value, we can control the number of segments obtained by the algorithm. Table 1 shows the performance of our algorithm under two different thresholds. In Table 1, *kZ* means k external memory banks of ZBT SRAMs, while *kD* means k external memory banks of DDR SRAMs. *#seg* is the number of segments, and all other figures show the computation time in msec. *none*, *r&c* and *c* mean *non-optimized*, *row&column-optimized* and *row-optimized scanning* respectively. In the caching algorithm, 32 lines in each cache memory bank are processed in parallel. So we can not use *row-optimized* method. As shown in Table 1, the performance of our circuit is faster than 30 frame per second with one unit. With four units (without cache memory banks), the performance can be improved about 3.7 - 3.8 times (we need to read four lines if data on one of the four lines are modified, and the computation of subsequent lines are delayed by 3 clock cycles as shown in Figure 5). With 32 units with cache memory banks (*non-optimized*), the improvement over four units with no cache memory banks is about 2.0 - 2.4 for *threshold-1*, and 1.2 - 2.3 for *threshold-2*. The reasons are follows.

1. In the approach with cache memory banks, the algorithm is applied at least twice to each region. Therefore, the performance gain with 32 units over four

Table 1. Performance of the algorithm

memory banks	threshold-1							threshold-2						
		non-caching		caching					non-caching		caching			
		2Z	7Z/4D	4Z		4D			2Z	7Z/4D	4Z		4D	
	#seg	1unit	4 units	32 units		32 units		#seg	1 unit	4 units	32 units		32 units	
		r&c	r&c	none	c	none	c		r&c	r&c	none	c	none	c
boat(512×512)	11477	22.8	6.18	2.76	2.38	2.64	2.25	2576	17.6	4.78	3.14	2.75	2.72	2.18
goldhill(512×512)	3717	21.6	5.86	2.43	2.17	2.31	2.05	3382	24.5	6.62	2.88	2.40	2.75	2.26
lena(512×512)	12633	22.7	6.12	2.61	2.31	2.49	2.19	1640	17.1	4.75	3.94	3.47	3.22	2.55
mountain(640×480)	13013	27.4	6.35	3.19	2.73	3.04	2.57	7769	25.1	5.85	2.93	2.44	3.47	2.29
peppers(512×512)	12883	22.4	6.04	2.56	2.23	2.44	2.10	1708	19.5	5.31	3.41	2.84	3.08	2.32

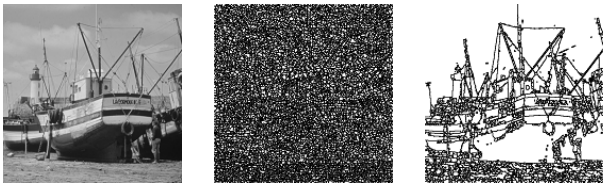


Fig. 8. Boat (org, threshold-1, threshold-2)

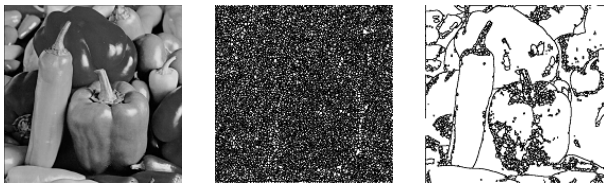


Fig. 9. Peppers (org, threshold-1, threshold-2)

units is at most four.

2. Because of the back-trace described in Section 4.2, each region may be processed more than twice. The size of segments under *threshold-2* is much larger than *threshold-1*, and we need more back-traces.

3. By processing 32 lines in parallel, the pipeline overhead described in Section 4.1 becomes larger.

By using DDR SRAMs and *column-optimized scanning* (which are not evaluated on hardware yet), we can improve the performance gain to 2.3 - 2.9 and 1.9 - 2.9 for *threshold-1* and *threshold-2* respectively. This performance is more than 20 times of software implementation on Pentium4 2.4GHz reported in [4].

Figure 8 and 9 show the results of *boat* and *peppers* in Table 1. Segments under *threshold-1* are so many, but the segmentation is the first step of various kinds of image processing, and the threshold is decided by the requirement of subsequent stages of the image processing. The results are almost same with the software algorithm *BM*, but there are some small differences on the way to divide non local-minima plateaus. Our algorithm divide them more evenly than *BM* because distances from lower plateaus are strictly calculated.

6. CONCLUSIONS

In this paper, we proposed an implementation method of a parallel and pipelined watershed algorithm on FPGA, and demonstrated that the implementation is fast enough for real-time applications. The execution time is 2.9 - 3.9 msec for images of 512 × 512 pixels, which is about 20 times faster than software programs. In our current implementation, the performance gain by 32 units over four units is limited. By strictly skipping pixels which will not be updated any more in *column-optimized scanning*, and by improving the algorithm so that the pipeline overhead (three clock cycles to start next lines) can be reduced, we will be able to achieve higher performance gain. We are now improving the algorithm and the circuit to achieve higher performance.

7. REFERENCES

[1] S. Beucher, F. Meyer, "The morphological approach to segmentation: The watershed transformation", Mathematical Morphology in Image Processing, Marcel Dekker Inc, New York, 1993, pp.433-481.

[2] F. Meyer, "Topographic distance and watershed lines", Signal Processing 38(1) (1994) 113–125.

[3] A. Bieniek, A. Moga, "An efficient watershed algorithm based on connected components", Pattern Recognition, 33 (2000), 907–916.

[4] H. Sun, J. Yang, M. Ren, "A fast watershed algorithm based on chain code and its application in image segmentation", Pattern Recognition Letters 26 (2005) 1266–1274.

[5] Zahirazami, Shahram Akil, Mohamed, "Implementation of a watershed algorithm on FPGAs", Proc. SPIE Applications of Digital Image Processing, Vol. 3460, p. 98-105, 1999.

[6] Kumud P. Gupta, S. Srinivasan, "Reduced Memory Implementation of Modified Serial Watershed Algorithm Based Ordered Queue", International Conference on Information Technology: Computers and Communications (ITCC) 2003

[7] Rambabu, C. Chakrabarti, I. Mahanta, A., "Flooding-based watershed algorithm and its prototype hardware architecture", Vision, Image and Signal Processing, IEE, Vol. 151 (3), 2004