FPGA Implementation of Parallel Histogram Computation

Asadollah Shahbahrami[1, 2], Jae Young Hur[1], Ben Juurlink[1], and Stephan Wong[1]

[1]Computer Engineering Laboratory
Delft University of Technology
2628 CD Delft, The Netherlands

[2]Department of Computer Engineering
Faculty of Engineering
The University of Guilan, Rasht, Iran

E-mail:{shahbahrami,j.y.hur,benj,stephan}@ce.et.tudelft.nl

Abstract. Parallelization of histogram functions is a challenging problem due to memory collisions. We propose a hardware technique to avoid memory collisions. It is called Parallel Histogram Computation (PHC). The hardware implementation of the PHC uses a dual-ported memory. This hardware technique needs two phases to perform the histogram function. First, two histogram arrays are calculated for pixels of even- and odd-numbered addresses, simultaneously. This means that the values that are stored at even- and odd-numbered addresses are used as indices to histogram arrays. In half of a cycle, these pixel values are read from both input ports. In the other half of the cycle, updated values of histogram elements are stored in different histograms. Thereafter, the two histograms are added and finally at the end, the calculated results are stored in the histogram array. The performance of the FPGA implementation of our proposed technique is compared to the performance of FPGA implementation of the fastest sequential algorithm. Our implementation reduces the number of cycles by a factor of 2 and improves performance by a factor of 1.92.

1 Introduction

A histogram is a diagram that depicts how many pixels of an image or a video frame have a certain intensity. It has many applications in image and video processing [1]. This is because extracting histogram features is simple and its features are invariant to image rotation. Furthermore, it has low storage requirements compared to size of the image. A histogram of an image of size $N \times M$ is calculated by the code below:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    Histogram[image[i][j]] +=1;
```

This implementation is referred to as the Scalar Algorithm (SA) in this paper. The size of the histogram array depends on the number of bits per pixel (bpp). If $bpp = n$, the histogram array has 2^n elements.

This research was supported in part by the Netherlands Organization for Scientific Research (NWO).

Parallelization of the histogram calculation is a challenging problem [2, 3]. This is because if there are several occurrences of a pixel value, there are several writes to the same memory location. Such a situation is referred to as a memory collision and is illustrated in Figure 1. In image and video processing collisions are common because there are many occurrences of a pixel value.

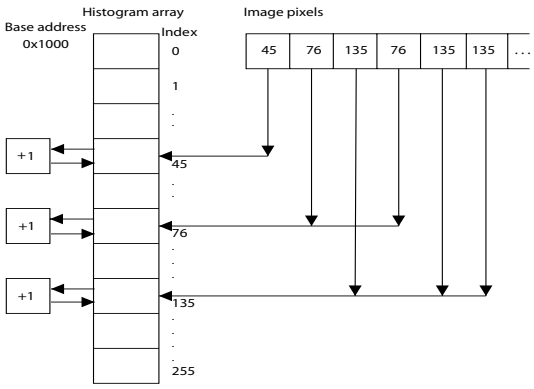

Fig. 1. Parallelizing the histogram calculation potentially leads to memory collision.

We propose a hardware technique for Parallel Histogram Calculation (PHC). Our proposed scheme leverages a dual-ported on-chip memory in order to avoid memory collisions. Considering sequentially processing hardware and software implementation as a reference, a comparative study is conducted. The presented hardware module is implemented in an FPGA. The results show that the PHC technique performs nearly 2× better with moderate area overheads, compared to the sequential hardware. We found that the memory capacity prevents implementing images larger than 525×525 for our target device and also extending the PHC to 4- and 8-way parallelism. Therefore, we use the stream image pixels as input to our hardware technique to overcome on this problem.

This paper is organized as follows. Related work is discussed in Section 2. Section 3 discusses the FPGA implementation of the scalar algorithm that is used as the reference implementation. Section 4 explains our proposed parallelized scheme. Section 5 presents the experimental results. Section 6 discusses the memory limitation and presents a solution. Finally, conclusions are drawn in Section 7.

2 Related Work

There are many software and hardware techniques to avoid memory collisions. Ahn et al. [2] have discussed three software methods for calculating the histogram function: sorting, privatization, and coloring. In the sorting method, the data is first sorted so that identical values are stored consecutively. Thereafter, the numbers of identical values are calculated before writing them to memory. In

the privatization scheme there are 2^{bpp} iterations and each iteration computes the sum for a particular pixel value. In the coloring scheme each color contains non-colliding elements. Then each iteration updates the sums in memory for a particular color.

We have proposed two techniques for SIMD vectorization of histogram functions in [4]. The first technique is hierarchical structure to provide $n-$way parallelism. This technique has three levels. In the first and second levels, auxiliary arrays consisting of n and $n/2$ subarrays are used, respectively. A histogram array is used for the last level. The elements of each two subarrays are added together and the results are stored in a subarray of the next level. The second technique is parallel comparators. In this technique the similar pixel values are pre-counted before doing a memory update.

A hardware scatter-add operation for data parallel SIMD architectures has been proposed in [2]. The scatter-add unit consists of a controller with multiplexers, a functional unit to perform the necessary operations, and a combining store. The combining store is used to guarantee that the scatter-add operations are performed atomically. The scatter-add unit avoids memory collisions by comparing the memory addresses with each other. This means that before reading data from memory, its address is compared to the addresses already stored in the combining store unit using a content addressable memory. The hardware of the scatter-add unit should be placed either in front of the memory controller or with each cache bank.

Muller [5] has discussed two basic architectures for histogram calculation. The first one consists of an array of counters, with one counter for each gray scale value. The number of counters scales with the number of gray scales. In addition, only one counter will be active in each cycle. The second one is a read-modify-write architecture. In this architecture an incrementer reads, modifies and writes the result of the addressed histogram element. One read-modify-write operation takes two system cycles using a single-ported memory.

Jamro et al. [6] have implemented the SA shown in Section 1 on the MicroBlaze soft-processor. They observed that 12 assembly instructions are necessary for this algorithm and that it takes about 30 cycles to process a single pixel value. They also investigated the read-modify-write architecture. Jamro et al. [6, 7] as well as Garcia [8] have used a dual-ported memory in order to reduce the number of cycles required for a read-modify-write operation. Both ports are addressed by the input pixel values. In [6, 7] the address on the second port is delayed by one cycle. This technique allows to fetch a new input pixel every cycle but it needs 2 cycles per pixel. In [8] each cycle is divided into two sub-cycles: a sub-cycle for fetching the current value and a sub-cycle for adding the memory content. This means that to calculate the histogram for an image of size $N \times M$, NM cycles are needed. This implementation is the fastest. Our proposed hardware technique processes two pixels in each cycle.

Compared to other works, we make the following contributions. First, we focus on the use of dual-ported memory technology. Second, we split the input image data into even and odd addresses. The histogram arrays of pixels stored

at even and odd addresses are calculated separately and simultaneously. Third, we implemented our proposed technique on an FPGA platform. It is almost 2 times faster than the previous fastest implementation.

3 Reference Implementation

We consider the hardware implementation of the scalar algorithm (HWSA) as a reference to compare with our implementation. Figure 2 depicts the HWSA module, where the HWSA is implemented with a dual-ported memory combined with an up-counter and an incrementer. The on-chip memory is the basic building block and the read/write signals, data input/output lines, and address pins are depicted in Figure 2. The behavior of the HWSA is close to the behavior of [8] in that we utilize a dual-ported memory and multiple clocks with different frequencies. Similar to [8], the cycle is divided in two sub-cycles, namely a *Read* cycle for getting the value, and a *Write* cycle for updating (+1) the memory content.

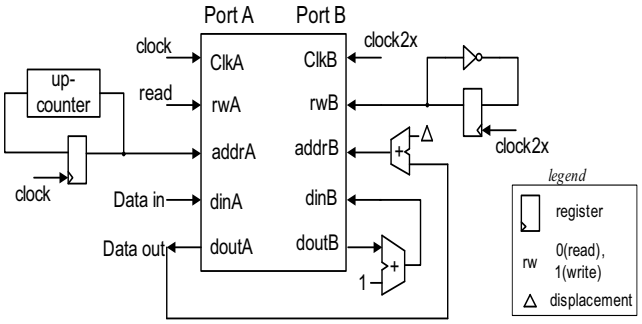

Fig. 2. Hardware implementation for the scalar algorithm.

In every cycle, the data is read from the on-chip memory port A when rwB is '0'. The clock frequency of port B is twice the clock frequency of port A. Data is read from port B in the first half of each cycle and written in the second half. The performance of HWSA is comparable to [8] and it requires N^2 cycles for an image of size $N \times N$. Since histogram elements are sequentially read and written, memory collisions do not occur. The HWSA requires $(N^2 + 2^{bpp})$ memory space.

4 Proposed Parallelized Scheme

As mentioned earlier, our objective is to design a hardware module for the Parallel Histogram Computation (PHC). The implemented hardware must avoid the memory conflict problem. Our proposed idea utilizes a dual-ported memory and is separated into two phases that are graphically depicted in Figure 3 for $bpp = 8$. In the first phase, two histogram arrays are calculated in parallel, one for pixels with even-numbered addresses and one for pixels with odd-numbered addresses.

Following the histogram functions, the values that are stored in even- and odd-numbered addresses used as indices to histogram arrays. In the first half of the cycle, these pixel values are read from both ports dout1A and dout1B as depicted in Figure 3. In the second half of the cycle, updated values (incremented) of the histogram elements are stored. In the second phase, the two histograms arrays, containing the histogram values of even- and odd-numbered addresses, are merged. The final results are stored in one histogram array according to Equation (1), where $m = 2$ and $bpp = 8$ in our example.

$$Histogram[i] = \sum_{i=0}^{2^{bpp}} \sum_{j=1}^{m} Histogram_j[i] \tag{1}$$

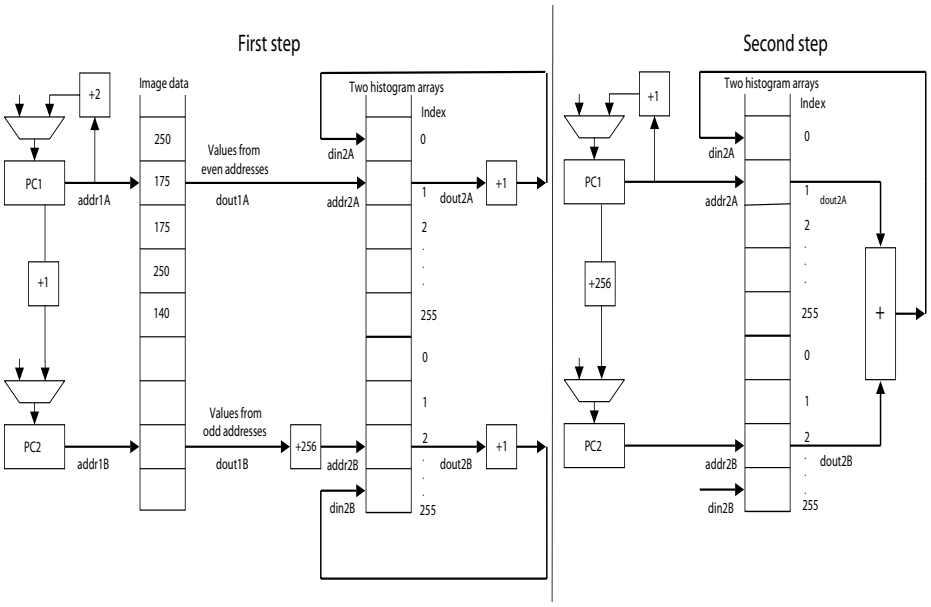

Fig. 3. The graphical representation of proposed technique for parallel histogram computation using dual-ported memory.

Figure 4 depicts the hardware module of the PHC technique. The module contains two memory blocks. The image data is stored in the first memory block, and two histogram arrays are stored in the second memory block. In addition, Figure 5 depicts the cycle-accurate functional simulation of the PHC. As depicted in Figure 5(a), two scalar functions are utilized in parallel to calculate the histogram contents and computed histogram elements are assumed to be stored from addresses 000 and 100, respectively. In the first phase, the data is read from the on-chip memory block 1, when rw (rw1A and rw1B) is '0'. The clock frequency of memory block 2 runs twice as fast as the clock frequency of memory block 1. The data is read from (when the rw is '0') and written to (when the rw is '1') memory 2, as depicted in Figure 5(b). Consequently, $\frac{4^2}{2}$

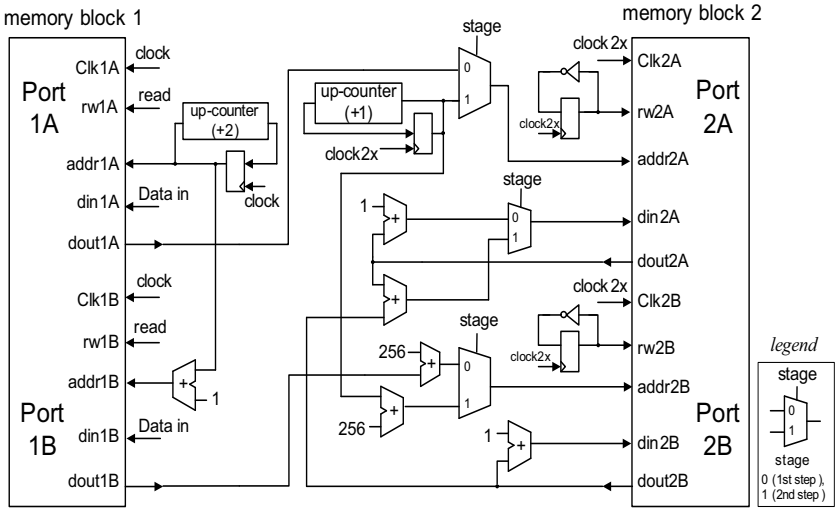

Fig. 4. Hardware module for 2-way parallel histogram computation.

cycles are required for the first phase. In the second phase, the merge operation is performed and requires 2^2 cycles. Additionally, memory space of 2^2 addresses are required. The performance of the first phase in the PHC module is almost $2\times$ better than the HWSA module, since the PHC algorithm requires $\frac{N^2}{2}$ cycles. The second phase of the PHC requires 2^{bpp} cycles . Consequently, our PHC requires $\frac{N^2}{2} + 2^{bpp}$ cycles. Given a general n-ported memory, the following cycles are required for the n-way parallelized computation:

$$PHC = \frac{N^2}{n} + 2^{bpp} \tag{2}$$

In case $N >> bpp$, the required number of cycles in n-way PHC is n times less than the HWSA. Our proposed parallelized scheme requires 2^{bpp} of additional memory space (i.e., the additional histogram array) to avoid the conflict. Additionally, there is a performance overhead for the merge operation, requiring 2^{bpp} cycles. However, the memory space overhead is not significant, based on the observation that the *bpp* size is in most cases much smaller than the number of pixels. As an example, an image of size 512×512 with $bpp = 8$ requires the $512^2 + 2 \times 2^8$ of memory space and the area overhead is only 2^8 memory locations or 0.1% compared to the HWSA. The computation requires $\frac{512^2}{2} + 2^8$ cycles and the performance overhead is 2^8 cycles or 0.1%.

5 Performance Evaluation

In this section, first, we present our evaluation environment. Subsequently, we evaluate the performance of the proposed hardware technique and compare it with the performance of the reference implementation.

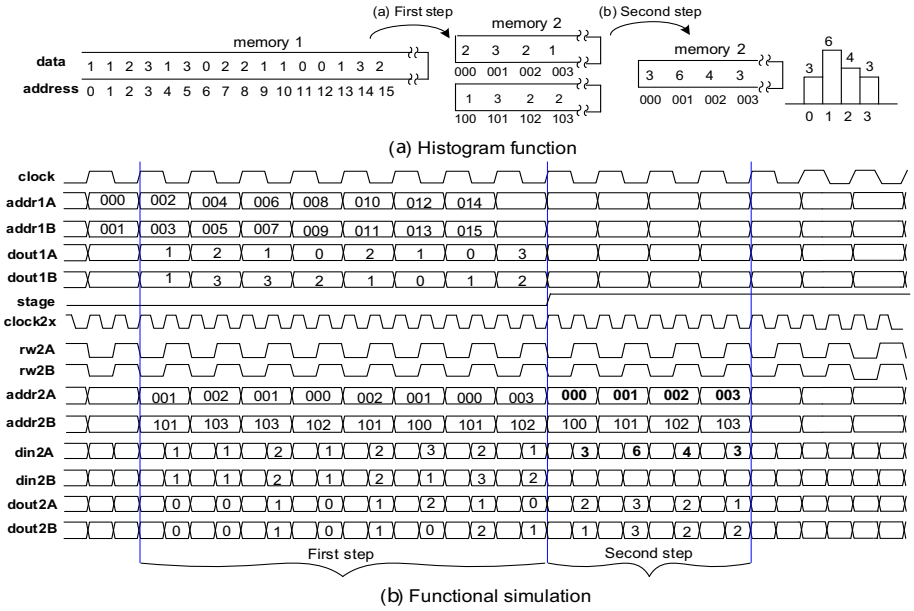

Fig. 5. Functional simulation for 2-way parallel histogram computation.

5.1 Evaluation Environment

The aforementioned hardware modules in Figure 2 and Figure 4 have been implemented in VHDL. We utilized Xilinx embedded dual-ported block memories (BRAMs) primitives. The BRAM width for image data corresponds to the size of the utilized *bpp* and the depth corresponds to the number of pixels in an image. In addition, the BRAM width for histogram array corresponds to the size of each element of the array and the depth corresponds to the 2×2^{bpp}. These parameters are changed for each particular *bpp* and image size. In other words, these width and depth, parameters are reconfigurable. We target the Virtex-II Pro xc2vp30-ff896-7 device that contains 136 BRAMs and 13696 slices. The capacity of each BRAM is 2K bytes and total available memory is 272K bytes. Considering 8-bit pixels, each BRAM primitive can accommodate 2048 pixels. Since a single BRAM primitive accommodates $\frac{16384}{bpp}$ pixels, $\lceil \frac{bpp \times N^2 + 2^{bpp} \times 32}{16384} \rceil$ BRAMs are required for HWSA and $(\lceil \frac{bpp \times N^2}{16384} \rceil + \lceil \frac{2 \times 2^{bpp} \times 32}{16384} \rceil)$ BRAMs are required for PHC technique to implement an image of size $N \times N$.

The hardware modules are synthesized, placed, and routed using the Xilinx ISE tool. The performance model in Equation 2 and the functional behaviors are verified by the post placement and route (PAR) simulations for HWSA and our 2-way PHC algorithms.

5.2 Performance Results

Figure 6 depicts the area utilization and the speedup of our 2-way parallel histogram computation over a 1-way scalar algorithm for different image sizes and

$bpp = 8$. The area column shows the number of utilized units and their percentage usage. The performance column presents the number of cycles, clock period, execution time, and speedup. The speedup has been obtained by the ratio of execution time of the 1-way implementation over the 2-way implementation.

As this figure shows for image size smaller than 32×32, the speedup is less than 1. For example, for image size 16×16 the speedup is 0.60. This is due to additional cycles that are needed to merge the histogram arrays. The amount of these additional cycles is 2^{bpp} (Equation 2). Therefore, the merge operation is not negligible when the image size is not significantly larger than bpp. On the other hand, for image sizes either equal or larger than 32×32, the speedup is larger than 1. For instance, the speedup for image size 256×256 is 1.92. This means that the merging overhead is negligible for large image sizes. Additionally, as both figures shows, the speedup of the 2-way implementation for image size 512×512 is slightly less than image size 256×256. This is because large image sizes almost utilize more slices which uniformly distributed in the chip. This means that logic complexity of the 2-way is slightly higher than the 1-way, which lightly incur longer interconnection delay. This causes a reduction in the clock frequency.

Image size		Area								Performance			
		Slices	%	LUTs	%	DCMs	%	BRAMs	%	Num. cycles	Clock period (ns)	Exec. Time (ns)	Speed up
16 x 16	1-way	31	0.2	53	0.2	1	6.3	1	0.7	256	9.98	2554	0.60
	2-way	122	0.9	217	0.8	1	6.3	2	1.5	384	11.11	4265	
32 x 32	1-way	35	0.3	57	0.2	1	6.3	2	1.5	1024	9.96	10203	1.20
	2-way	122	0.9	217	0.8	1	6.3	2	1.5	768	11.11	8529	
64 x 64	1-way	52	0.4	82	0.3	1	6.3	3	2.2	4096	9.96	40813	1.57
	2-way	133	1.0	225	0.8	1	6.3	3	2.2	2304	11.27	25961	
128 x 128	1-way	56	0.4	88	0.3	1	6.3	9	6.6	16384	9.99	163643	1.74
	2-way	148	1.1	255	0.9	1	6.3	9	6.6	8448	11.11	93874	
256 x 256	1-way	120	0.9	142	0.5	1	6.3	33	24.3	65536	11.43	749076	1.92
	2-way	244	1.8	359	1.3	1	6.3	33	24.3	33024	11.79	389485	
512 x 512	1-way	246	1.8	420	1.5	1	6.3	129	94.9	262144	14.16	3713008	1.80
	2-way	506	3.7	911	3.3	1	6.3	129	94.9	131328	15.74	2067628	

Fig. 6. The required area and the speedup of 2-way parallel histogram computation over 1-way scalar algorithm for different image sizes and $bpp = 8$ implemented on the Virtex-II Pro.

As previously mentioned, Figure 6 also shows area utilization for both implementations. This figure shows that the number of utilized BRAMs in a particular image size is the same in both hardware implementations except for image size 16×16. For this image size, the number of BRAM in the 1-way and 2-way is 1 and 2, respectively. For image size 512×512, both implementations need 129 BRAMs consuming 94.9% of the total BRAMs in the device. On the other hand, the number of utilized slices is not the same in the HWSA and PHC implemen-

tation. For example, for image size 256×256 the number of utilized slices is 142 and 359, respectively. This means that the 2-way implementation needs almost 2.50 times more area in terms of utilized slices than the 1-way implementation. Our target device has 13696 slices, while the percentage usage of slices is significantly less compared to the number of slices. For example, the percentage usage of slices in the 2-way implementation of the image size 256×256 is 1.8. This means that the area utilization in terms of slices is not a bottleneck for parallel histogram computation. In general, the PHC occupies more area in terms of slices than the HWSA, since additional multiplexers and adders are required, while there is no area overhead in terms of BRAMs in the former approach.

In general, for large image sizes, the 2-way implementation needs half of a cycle to process a single pixel, while the 1-way implementation needs one cycle. This means that the number of cycles in the 2-way implementation is two times less than 1-way implementation. In addition, the clock frequency of these implementations is not the same as shown in Figure 6. Consequently, the speedup for large image is slightly less than 2.

The proposed idea can be extended to 4- and 8-way parallelism as well. This means that we can reduce the number of cycles by a factor of 4 and 8. However, the BRAMs units are a bottleneck to implement larger image sizes than 525×525 and to implement higher parallelism. In the next section, this bottleneck is discussed in more detail.

6 Memory Considerations

As already mentioned, the total capacity of the BRAMs in the Virtex-II Pro is $136 \times 2Kbytes = 278528$ bytes. This capacity is not sufficient to implement large image sizes. Therefore, the memory capacity of the FPGA device is a limitation. The simplest way to overcome on this limitation is, using new devices which have much more BRAMs units. However, the number of BRAMs units in these devices is still limited.

To overcome on aforementioned problem, we use stream image and video data for parallel histogram calculation. In the stream data, a sequences of image pixels passing through the computational units. Using stream data and 2-way implementation, the hardware can receive two pixels and process them simultaneously without storing them. This implementation just needs two BRAMs units. That means that the percentage usage of the BRAMs units is independent from the image sizes.

Figure 7 depicts the speedup of the 2-way PHC over the HWSA using stream data for different image sizes and $bpp = 6, 8$, and 12. As this figure shows that we can implement larger image sizes using stream data. With increasing the image sizes the speedup is also increased. In other words, the largest speedup is obtained for the largest image size. For example, for $bpp = 8$ the speedup is increased from 0.6 for image size 16×16 to 1.8 for image size 1024×1024.

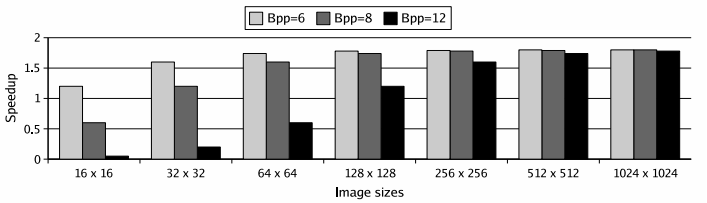

Fig. 7. Speedup of the 2-way parallel histogram computation over scalar algorithm for different image sizes and bpps using stream image pixels.

7 Conclusions

In this paper, a hardware technique for parallel histogram computation has been proposed. The novel hardware unit avoids memory collisions in the histogram functions by using a dual-ported memory. The proposed hardware calculates the histogram of an image in two phases. First, image pixels are divided into even- and odd-numbered addresses. The pixel values that are stored in even- and odd-numbered addresses used as indices to histogram arrays. Therefore, two histogram arrays are calculated for both pixels in the even- and odd-numbered addresses, simultaneously. In half of a cycle, these pixel values are read from dual-ported memory and in other half of the cycle updated values of histogram elements are stored. Second, after that the histograms of even- and odd-numbered addresses calculated, those histograms arrays are merged and finally at the end, the calculated results are stored in a histogram array. Experimental results obtained by FPGA implementation have shown that the proposed technique improve the performance compared to the fastest scalar version by a factor of 1.92.

References

1. S. Deb, "Video Data Management and Information Retrieval," IRM Press, 2005.
2. J. H. Ahn, M. Erez, and W. J. Dally, "Scatter-Add in Data Parallel Architectures," In *Proc. 11th Int. Symp. on High-Performance Computer Architecture*, pages 132–142, February 2005.
3. B. R. Gaeke, P. Husbands, X. S. Li, L. Oliker, K. A. Yelick, and R. Biswas, "Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines," In *Proc. Int. Symp. on Parallel and Distributed Processing*, April 2002.
4. A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "SIMD Vectorization of Histogram Functions," In *Proc. 18th IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP07)*, pages 174–179, July 2007.
5. S. Muller, "A New Programmable VLSI Architecture for Histogram and Statistics Computation in Different Windows," In *Proc. Int. Conf. on Image Processing*, pages 73–76, 1995.
6. E. Jamro, M. Wielgosz, and K. Wiatr, "FPGA Implementation of the Dynamic Huffman Encoder," In *Proc. Workshop of Programmable Devices and Embedded Systems*, pages 60–65, February 2006.
7. E. Jamro, M. Wielgosz, and K. Wiatr, "FPGA Implementation of the Strongly Parallel Histogram Equalization," In *Proc. 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, April 2007.
8. E. Garcia, "Implementing a Histogram for Image Processing Applications," *Xilinx Xcell Magazine*, (38):40–46, 2000.