SIMD Vectorization of Histogram Functions

Asadollah Shahbahrami[1,2], Ben Juurlink[1], and Stamatis Vassiliadis[1]

[1]Computer Engineering Laboratory
Delft University of Technology
2628 CD Delft, The Netherlands
shahbahrami,benj,stamatis@ce.et.tudelft.nl

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

Abstract

Existing SIMD extensions cannot efficiently vectorize the histogram function due to memory collisions. We propose two techniques to avoid this problem. In the first, a hierarchical structure of three levels is proposed. In order to provide n-way parallelism, auxiliary arrays that have n and $n/2$ subarrays are used in the first and second level, respectively. The last level has the primary histogram array. Indirect SIMD load and store instructions are designed in order to access different elements of different subarrays. The different subarrays in the lower levels are merged and finally at the end, the calculated results are stored in the primary histogram array. In the second method, parallel comparators are used in order to count the number of subwords within a media register that are the same. Thereafter, these numbers are added to the values of the histogram array simultaneously. Experimental results obtained by extending the SimpleScalar toolset show that proposed techniques improve the performance compared to the fastest scalar version by a factor of 7.37 and 5.52, respectively.

Keywords: Subword Parallelism, Multimedia Extensions, Histogram Calculation.

1 Introduction

Histogram features are used in some applications such as image and video retrieval, video sequence segmentation, and objects classification in image processing and pattern recognition [5]. Additionally, in [4] has been indicated that using the color histogram features is the most suitable compared to the texture and shape features for large databases such as Web. This is due to its simplicity, invariant to image rotation, and low storage requirements compared to the size of the image.

Given an image of size $N \times M$, a histogram is simply the count of how many pixels of the image map to each element

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

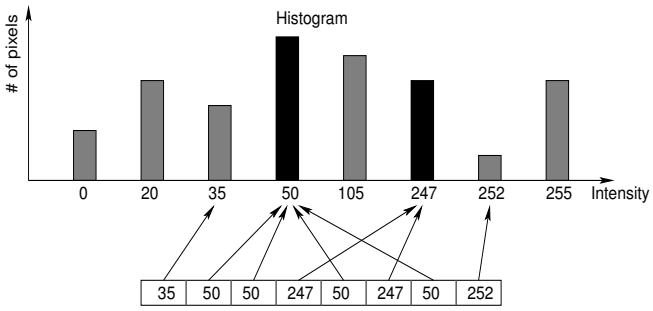

Figure 1. SIMD vectorization of histogram leads to memory collisions when multiple subwords contain the same value.

as shown in the code below:

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

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. We call this array the primary histogram array.

SIMD vectorization of the histogram computation, however, is a challenging problem. The most important reason for this is memory collisions [1] as illustrated in Figure 1. Memory collisions increase the number of memory accesses. In image and video processing collisions are common because there are many occurrences of the same pixel value in either an image or a frame.

Existing SIMD architectures such as MMX [9] and SSE [10] cannot efficiently vectorize this important function. The most important reason is that these SIMD extensions cannot support indexed (indirect) load or store instructions. Hence, we propose two techniques to vectorize the histogram calculation using subword processing. The first method is called *hierarchical structure*. This structure has three levels. In the first and second levels, auxiliary arrays with different sizes and data type are used. The primary histogram array is used in the last level. We use n pixel values as indirect pointers to these auxiliary arrays using SIMD

1-4244-1027-4/07/$25.00 ©2007 IEEE

indirect pointer. With this SIMD indirect pointer we can either load or store data from n locations at the same time using indexed load/store instructions. After the calculation of the histogram on this auxiliary arrays, the result will be merged at the end and stored in the primary array.

The second technique is based on *parallel comparators*. Parallel comparators are used to count the number of subwords that are the same. After counting the number of subwords that are the same, counted numbers are added with values of histogram array simultaneously. A special-purpose instruction has been designed for this technique. It is called *parallel count within a media register* (pcwar).

Our proposed techniques, SIMD instructions, and MMX instructions have been simulated by extending the SimpleScalar toolset [3]. The performance acquired by the proposed techniques has been compared to scalar implementation. The experimental results show that:

- The speedup of the hierarchical structure for 8-way parallel SIMD instructions ranges from 3.17 to 7.37.
- The speedup of the hierarchical structure for 4-way parallel ranges from 5.77 to 6.19.
- The 8-way parallelism is faster than 4-way parallelism for small bpps. For large bpps, on the other hand, the 4-way parallelism is faster than 8-way parallelism.
- Parallel comparators provide a speedup ranging from 5.14 to 5.52 using 4-way parallelism.

This paper is organized as follows. Related work is discussed in Section 2. Different algorithms to calculate the histogram function and the proposed techniques are discussed in Section 3 and Section 4, respectively.. Experimental results are presented in Section 5, and conclusions are drawn in Section 6.

2 Related Work

Existing techniques to calculate the histogram function can be divided into two groups based on how they deal with memory collisions. The first group contains techniques that accept the conflicts. For example, such a method was proposed by Suehiro et al. [12]. It is called the retry method. This algorithm detects store conflicts and stores them in a retry queue, which stores conflicting keys for later retry. This technique is a vectorization algorithm that has been developed for vector computers based on gather and scatter operations. This algorithm has some limitations such as it requires additional memory for the retry queue and extra computation for detecting store conflicts.

The second group consists of techniques that try to avoid store conflicts. For instance, Ahn et al. [1] have discussed such techniques. They have explained three software methods : sorting, privatization, and coloring. In the sorting method, to avoid memory collisions, the data is first sorted so that identical values are stored consecutively. Thereafter, the sum of data for each memory address is calculated before writing it to memory. In the privatization technique there are 2^{bpp} iterations and each iteration computes the sum for a particular gray level. In the coloring scheme each color contains non-colliding elements. Then each iteration updates the sums in memory for a particular color.

Ahn et al. [1] have also proposed a hardware scatter-add operation for vector architectures. 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 the memory collisions by comparing the memory addresses with each other. The performance of the hardware scatter-add depends on the distribution of data. When the range of indices is small, the performance of the scatter-add is reduced due to the hot bank effect. This effect causes some of the scatter-add units to be idle. Furthermore, in [1] the performance of the scatter-add unit was compared to the performance of the sort and privatization algorithms. Based on our results these are not the fastest algorithms.

SIMD instructions have been used to implement many media algorithms such as the (I)DCT [11, 8]. To the best of our knowledge, however, they have not been used to implement the histogram function. Compared to other works, we make the following contributions. First, we focus on the use of SIMD instructions to calculate the histogram function. Second, in order to use SIMD instructions and to provide n-way parallelism, we propose two techniques, hierarchical structure and parallel comparators. Finally, we have designed and implemented indirect SIMD load and store instructions as well as a special-purpose instruction.

3 Background

In order to calculate a histogram for an image size $N \times M$, five algorithms have been implemented in C. These methods are : fully scalar, mixed scalar/SIMD, privatization, modified privatization, and using merge sort to calculate the histogram. In the fully scalar algorithm, a single pixel value is processed in each iteration. In the mixed scalar/SIMD algorithm eight pixel values are loaded using the SIMD load instruction. Thereafter, the pixel values are processed as in the scalar algorithm. Privatization algorithm scans the entire image 2^{bpp} times. In each iteration it computes the total number of pixels of a particular gray level. In the modified privatization algorithm, we put all of the same values close to each other during the calculation of the histogram. This means that after the calculation of the

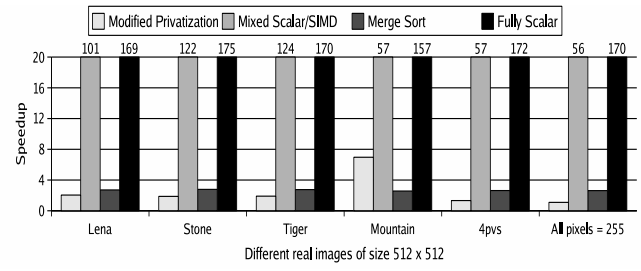

Figure 2. Speedup of the different algorithms over the privatization algorithm for different real images with 8 bpps on the Pentium 4.

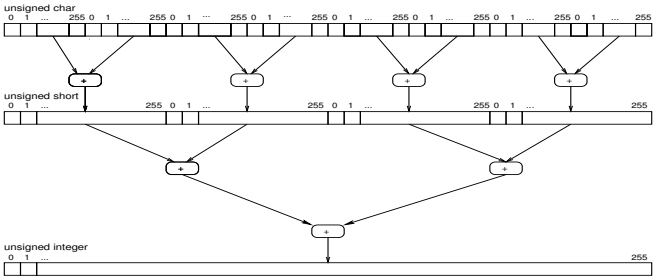

Figure 3. Providing 8-way parallelism using hierarchical structure.

histogram, the image pixels are sorted in ascending order. The merge sort algorithm has also been used to compute the histogram. First, the image pixels are sorted. Then the sum of data for each memory address is calculated before writing it to memory.

In order to determine which algorithm is fastest, we have executed these algorithms on some real images. All programs were compiled using gcc with optimization level *-O2* except the SIMD part of the mixed scalar/SIMD algorithm. This algorithm was implemented using MMX [9]. As experimental platform we have employed a 3.0GHz Pentium 4 processor. Performance was measured using the IA-32 cycle counter [6].

Figure 2 depicts the speedup of the different algorithms over the privatization algorithm for images of size 512×512 with 8-bpp. As this figure depicts fully scalar is the fastest algorithm. In the mixed scalar/SIMD algorithm, many instructions to transfer data from media registers to scalar registers are needed. Furthermore, many shift instructions are needed to separate different subwords. The performance of the privatization and modified privatization algorithms depend on the distribution of the data. For example, for "Mountain" which is an almost black and white image, modified privatization is about 6.97 times faster than privatization. The "4pvs" is a generated image where each square of size 256×256 is assigned the same pixel value. In this case modified privatization is only 1.33 times faster than privatization. The performance of merge sort does not depend on the distribution of data and for all cases its speedup is about 2.65. In this paper, we use the fully scalar algorithm as the reference implementation.

4 Vectorization of the Histogram Function

In this section we propose two techniques to vectorize the histogram function, hierarchical structure and parallel comparators.

4.1 Hierarchical Structure Algorithm

To provide n-way parallelism, a hierarchical structure of three levels is proposed. In the first and second level, auxiliary arrays consisting of n and $n/2$ subarrays are used, respectively. The primary histogram array is used for the last level. Each subarray has 2^{bpp} elements, the same as the primary histogram array. In addition, the elements of the auxiliary array in the first level are of type byte (unsigned char), the elements of the second level auxiliary array of type short, and the elements of the third level primary array of type int. The elements of each two subarrays are added together and the results are stored in a subarray of the next level. At the end, the calculated histograms are stored in the primary histogram array. Figure 3 depicts an example of hierarchical structure to provide 8-way parallelism with $bpp = 8$.

We load n pixel values using an SIMD load instruction. For example, 8 pixel values are read using the `movq` instruction of the MMX ISA. These pixel values are used as a set of pointers for an indexed load from the auxiliary array. This is called an SIMD indirect pointer, allowing to retrieve data from multiple locations at the same time. For instance, the indexed load takes an index vector and fetches four elements into an SIMD register. The first element is fetched from the first subarray, the second one from the second subarray, and so on. In this way, different memory addresses are accessed even if some or all of the indexes in the index vector are the same. Figure 4 illustrates the indexed load operation with $bpp = 8$ bits. It can be seen that the addresses of the elements are obtained by adding the base address of the auxiliary array to the offsets given in the index vector and constant displacements.

As this figure shows the result of the indexed load is adjacent elements in a vector register. After these elements are computed using SIMD instructions, they can be stored in their memory locations by an indexed store, using the same index vector.

4.2 Parallel Comparators

The second technique proposed to avoid memory collisions is to count the different subwords in a media register using parallel comparators. After counting the number of subwords that are the same, the sums are added to the values in the histogram array simultaneously. If there are n sub-

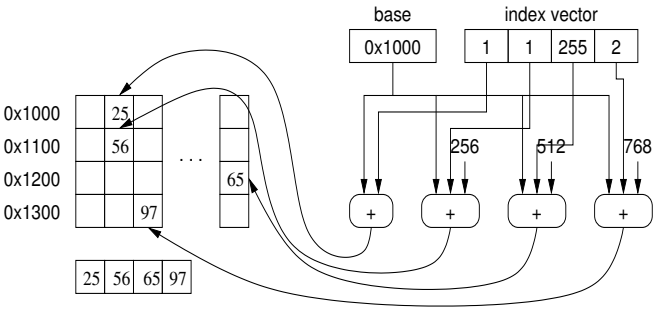

Figure 4. Explanation of the indexed vector load instruction.

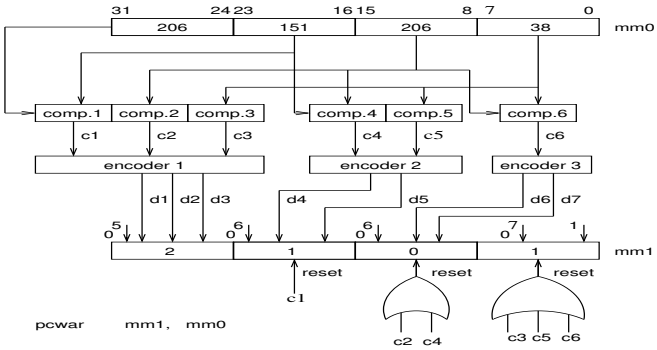

Figure 5. Using parallel comparators to count the similar values within a media register.

words within a media register then $(n-1)+(n-2)+...+1$ comparators are needed. In addition, this hardware unit needs $n-1$ encoders and a few OR gates. Figure 5 depicts an example, where there are 4 subwords in each media register and each subword is 8-bit. For simplicity, write and clock signals and also the functionality of the encoders have been omitted.

A special-purpose instruction for this hardware unit has been implemented. It is referred to as *parallel count within a register (pcwar)*. This instruction has two operands. The source operand consists of an SIMD vector of input values and the destination operand contains of an SIMD vector register of the counted number of the similar subwords. Let the source operand be denoted by s with subwords $s_0, s_1, \ldots, s_{n-1}$ (from left to right). Similarly, the destination SIMD register is denoted by d with subwords $d_0, d_1, \ldots, d_{n-1}$. Then d_i will be set to the number of subwords s_j that are identical to s_i. Furthermore, if $s_i = s_j$ with $i < j$, then d_j will be set to zero.

5 Experimental Evaluation

In this section, first our methodology is explained and then we compare the performance obtained by the proposed techniques to the performance of the fully scalar version.

5.1 Simulation Environment

In order to evaluate the proposed techniques, we have extended the `sim-outorder` simulator of the SimpleScalar toolset [2] using the SSIT and SSAT tools [7].

The main parameters of the modeled processors are depicted in Table 1. We modeled processors with issue widths varying from 1 to 4 instructions per cycle. So, when four SIMD instructions are issued simultaneously, up to 32 data operations are executed in parallel. When the issue width is doubled, the number of functional units is scaled accordingly except for the integer and SIMD multipliers, of which there are at most 2. The latency and throughput of SIMD instructions is considered to be equal to the latency and throughput of the corresponding scalar instructions. This is a very reasonable assumption given that the SIMD instructions perform the same operation but on narrower data types. In addition, we set the latency of indirect load and store accesses larger than normal load and store. Although indirect accesses are less efficient than unit-stride accesses, in our case the sparseness of the indices is not very much. For example, the distance between two index patterns is at most 1022 bytes.

Two versions of each algorithm have been implemented. One program is completely written in C. To calculate the histogram function in this program we have used the fully scalar algorithm as was discussed in Section 3. This program was compiled using gcc (version 3.3.2) with optimization level *-O2*. The other program was implemented using MMX and our new SIMD instructions. As input we use randomly generated images with 4 and 8 bpps. The correctness of the MMX codes has been validated by comparing their output to the output of the C program.

5.2 Experimental Results

Figure 6 depicts the speedup and the ratio of committed instructions of the 8-way hierarchical structure over the fully scalar implementation for different image sizes and 4 and 8 bpps. The speedup ranges from 3.17 to 7.37.

Parameter	Value
Issue width	1/ 2/ 4
Integer ALU, SIMD ALU	1/ 2/ 4
Integer MULT, SIMD MULT	1/ 2/ 2
L1 Instruction cache	512-set, direct-mapped 64-byte line LRU, 1-cycle hit, total of 32 KB
L1 Data cache	128-set, 4-way, 64-byte line, 1-cycle hit, total of 32 KB
L2 unified cache	1024-set, 4-way, 64-byte line, 6-cycle hit, total of 256 KB
Main memory latency	18 cycles for first chunk, 2 thereafter
Memory bus width	16 bytes
RUU (register update unit) entries	64
Load-store queue size	8
Execution	out-of-order

Table 1. Processor configuration.

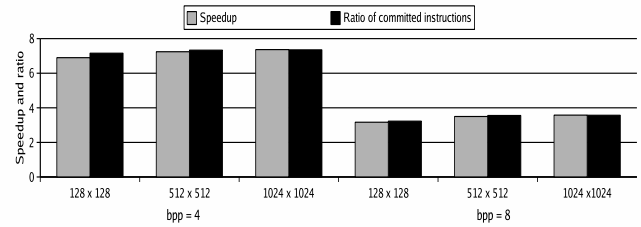

Figure 6. Speedup and ratio of committed instructions of the 8-way hierarchical structure over the fully scalar implementation for different image sizes and bpps on the single issue processor.

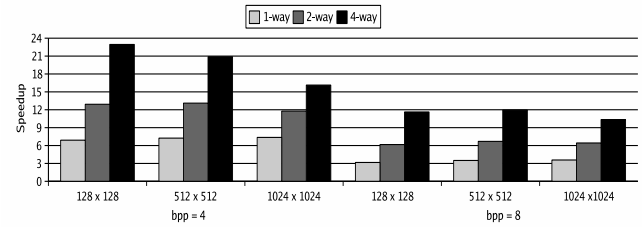

Figure 7. Speedup of the 8-way hierarchical structure over the fully scalar implementation on a single issue processor for different issue widths.

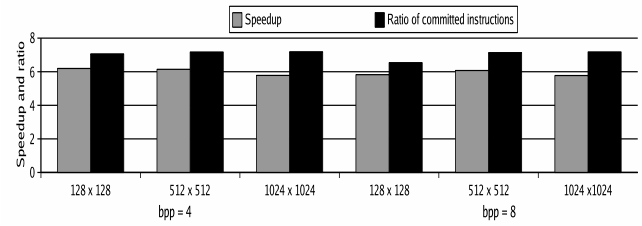

Figure 8. Speedup and ratio of committed instructions of the 4-way hierarchical algorithm over the fully scalar implementation for different image sizes on the single issue processor.

When increasing the image size from 128×128 to 1024×1024 with the same bpp, the speedup of our approach slightly increases. It can also be observed that the speedup for 4 bpps is higher than for 8 bpps. The most important reasons for this are the following. First, for small bpps the auxiliary arrays are small. Hence, the number of iterations to merge the subarrays for 4 bpps is 16 times less than the number of iterations for 8 bpps. For example, for 4 bpps, the auxiliary arrays in the first and second level of the hierarchical structure have 8×16 and 4×16 elements, respectively. For 8 bpps, on the other hand, those arrays have 8×256 and 4×256 elements, respectively. Second, to merge two subarrays, many overhead instructions are needed for unpacking the subwords. For instance, to merge the subarrays from the first level to the second level for 4 and 8 bpps, 32 and 512 unpack instructions are needed, respectively. As a result, using the 8-way parallelism is more efficient for small bpp than for large bpp.

Figure 7 depicts the speedup of the 8-way parallel implementation that uses the hierarchical structure over the fully scalar algorithm on out-of-order processors with different issue widths. Obviously, when increasing the issue width from 1- to 4-way the speedup increases. However, increasing the issue width yields diminishing returns. For example, for the image size of 128×128 and for 4 bpps, the speedups are 6.90, 12.92, and 22.95 for issue width of 1, 2, and 4, respectively. For the image size of 1024×1024, the speedups are 7.37, 11.78, and 16.14, respectively. The main reasons for this are following. First, in the main loop body of the program, there are four SIMD instructions, three of which are load and store instructions. With increasing issue width and image sizes the memory stall time will increase. In other words, the memory stall time for a small issue width is less than for a large issue width. Second, ILP exploitation with increasing issue width is limited, due to data dependencies between instructions.

In the 4-way parallel implementation the auxiliary array consists of 4 subarrays. As in the 8-way parallel implementation, we load 8 pixel values using an SIMD load instruction. These 8 values are divided into two groups, low and high values. These four low and four high values are used as indices to the auxiliary array for reading eight different elements. This means that we use two times as many indirect SIMD load and store instructions in each loop iteration as the 8-way parallel implementation. We therefore expect that the ratio of committed instructions is larger than the speedup because of memory stalls. Figure 8 depicts the speedup and the ratio of committed instructions of the 4-way hierarchical structure over the fully scalar implementation for different image sizes and 4 and 8 bpps.

As this figure shows, the ratio of committed instructions ranges from 7.06 to 7.19, while the speedup is 5.77 and 6.19. In contrast to the 8-way hierarchical implementation, there is not much difference between the speedups for different image sizes and bpps. This is because of the following reasons. First, the code of the main loop body is the same for all image sizes. Second, although the auxiliary array for 4 bpps are smaller than for 8 bpps, the merge operation of the subarrays from the second level to the third level is performed once for both. Comparing Figure 6 to Figure 8 shows that the 8-way hierarchical implementation is faster than the 4-way implementation for 4 bpps, while for 8 bpps it is much slower.

Figure 9 depicts the speedup and ratio of committed instructions of the implementation based on parallel compara-

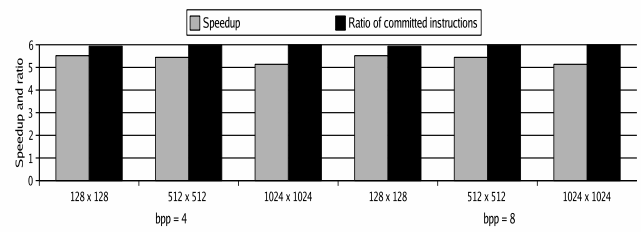

Figure 9. Speedup and ratio of committed instructions of the implementation based on parallel comparators over the fully scalar implementation for different image sizes and bpp on the single issue processor.

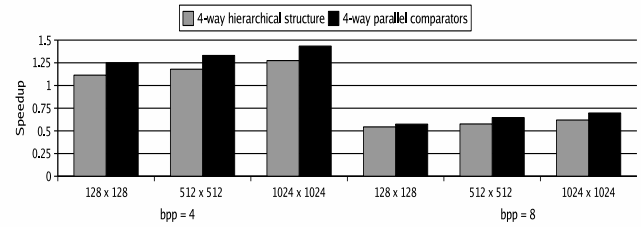

Figure 10. Speedup of the 8-way hierarchical implementation over the 4-way hierarchical algorithm and the implementation based on parallel comparators for different image sizes and bpps on the single issue processor.

tors over the fully scalar algorithm for different image sizes and bpps. With increasing image size the speedup slightly decreases from 5.52 to 5.14. On the other hand, the ratio of committed instructions slightly increases from 5.94 to 5.99. We have used 4-way parallel SIMD instructions. This implies that the elements of the histogram array are of type unsigned short. In each iteration, 8 pixel values are loaded using an SIMD load instruction as in previous algorithms. The special-purpose instruction `pcwar` is used two times for low and high values in each iteration.

Figure 10 depicts the speedup of the 8-way hierarchical implementation over the 4-way hierarchical algorithm and over the implementation based on parallel comparators. For 4 bpps, the 8-way hierarchical implementation is faster than both other algorithms. On the other hand, for 8 bpps both the 4-way hierarchical implementation and the implementation that uses parallel comparators are faster than the 8-way hierarchical implementation. Additionally, the 4-way hierarchical implementation is faster than the implementation based on parallel comparators. This can also be seen in Figure 8 and Figure 9. The main reason for this is that the number of committed instructions of the parallel comparators technique is larger than the number of instructions of the 4-way hierarchical implementation.

6 Conclusions

In this paper, two techniques to avoid memory collisions in the histogram functions have been proposed. In the first technique different subarrays are used in the first and second level of the hierarchical structure in order to provide subword parallelism. Indirect SIMD load and store instructions have been designed to access the different elements of the subarrays simultaneously. The different subarrays in the lower levels are merged and finally at the end, the calculated results are stored in the primary histogram array in the last level. In the second method, we have used a special hardware unit of parallel comparators to count identical values of different subwords in a media register. The sums are added to the values of the histogram array in parallel. Experimental results obtained by extending the SimpleScalar toolset have shown that the proposed techniques improve the performance compared to the fastest scalar version by a factor of 7.37 and 5.52, respectively.

References

[1] 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.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[3] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. www.simplescalar.com.

[4] S. Deb. *Multimedia System and Content-Based Image Retrieval*. Idea Group Publishing, 2004.

[5] S. Deb. *Video Data Management and Information Retrieval*. IRM Press, 2005.

[6] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual Volume 3 System Programming Guide*, 2004. Order Number: 253668.

[7] B. Juurlink, D. Borodin, R. J. Meeuws, G. T. Aalbers, and H. Leisink. The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT). Available via `http://ce.et.tudelft.nl/~shahbahrami/`

[8] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[9] A. Peleg, S. Wiljie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):25–38, 1997.

[10] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium 3 Processor. *IEEE Micro*, 20(4):47–57, July-August 2000.

[11] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. Matrix Register File and Extended Subwords: Two Techniques for Embedded Media Processors. In *Proc. 2nd ACM Int. Conf. on Computing Frontiers*, pages 171–180, May 2005.

[12] K. Suehiro, H. Murai, and Y. Seo. Integer Sorting on Shared Memory Vector Parallel Computers. In *Proc. of ICS*, 1998.