GRAPHICAL MODELS AND IMAGE PROCESSING
Vol. 57, No. 2, March, pp. 175–182, 1995

NOTE

Efficient 2-D Convolution Algorithm with the Single-Data Multiple Kernel Approach

JIHONG KIM

Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195

AND

YONGMIN KIM

Department of Electrical Engineering, FT-10, University of Washington, Seattle, Washington 98195

Receiving January 24, 1994; revised December 22, 1994; accepted December 30, 1994

A fast implementation of convolution operation is described. The implementation takes advantage of various properties of commonly used kernels. First, the kernel elements are analyzed and rearranged, grouping the kernel elements of the same absolute value together and arranging trivial multiplication cases separately. Then the sequence of computing convolution sums is modified so that all the kernel elements could be multiplied by the same data element at each step. This new algorithm significantly reduces the number of multiplication operations for most widely used kernels. Our experiment shows that the performance improvement of two to seven over the direct implementation is achievable for most kernels. The required overhead (extra memory and preprocessing) is proportional to the kernel size and is negligible. © 1995 Academic Press, Inc.

1. INTRODUCTION

Convolution operations are widely used as an effective tool for detecting features (such as points, lines, or edges), smoothing noises, and sharpening image details. The common difficulty of implementing the convolution algorithm has been its large computational requirements. For example, in order to convolve a 512 × 512 image with a 5 × 5 kernel, over 13 million multiplications and additions are necessary. Since convolution operations are usually employed in the preprocessing or early processing stage of many image processing, image analysis, and computer vision tasks, the support for fast convolution operations is quite valuable in order for a whole task to be completed without excessive delay. When real-time responses are necessary, a hardware convolver with the necessary supporting chips could be used. Many dedicated convolver chips have been developed for this purpose. For example, the LSI Logic's L64240 8 × 8 multibit finite impulse response (FIR) filter chip can convolve images very fast [1]. Many image processors (IPs) and digital signal processors (DSPs) such as Texas Instruments' TMS320 series have been developed and used to speed up the convolution operations as well [2–4].

With the advances in device technology and computer architecture, convolution operations on the general-purpose workstations are getting more practical, though they could not be done in real time [5]. For the software-only implementation, there are several implementation techniques available to reduce the number of arithmetic operations. If a kernel is separable, 2-D convolution could be implemented by applying two consecutive 1-D kernels. The separable kernel approach could achieve a considerable speed-up over the direct implementation especially when the kernel size is large [6]. The number of separable kernels, however, is rather limited. If the implementation is specially optimized for a specific kernel, convolution computation could be made more efficient. For example, if all the kernel elements had a value of 1, the box-filter or moving average method would work [7]. However, these techniques are not generally applicable; thus they are not useful in implementing convolution operations when kernels are not known in advance. In this paper, we describe a fast generalizable convolution algorithm.

We consider $p \times p$ square kernels. It is convenient to assume that p is an odd number and to denote $q = (p - 1)/2$. Let A be an $n \times m$ matrix input image and K be an $p \times p$ matrix kernel, where $n, m > p$. Then for all i, j satisfying $q < i \leq n - q$ and $q < j \leq m - q$, let $A_{i,j}$ be the $p \times p$ square submatrix of A centered in $A[i, j]$. We say

1077-3169/95 $6.00

that an output $n \times m$ matrix B is a discrete convolution of A with the kernel K:

$$B[i, j] = \sum_{1 \le k,l \le p} A_{i,j}[k, l]K[p - k + 1, p - l + 1]. \quad (1)$$

The boundary elements can be treated as a special case or ignored. The direct implementation of convolution operations would require p^2 multiplications and p^2 additions for each convolved element.

Our algorithm takes advantage of various properties of commonly used kernels. For most of the kernels we tested, we were able to achieve a speed-up factor of 2 to 7 over the direct implementation. Section 2 reports the analysis results for the commonly used kernels while our algorithm is presented in Section 3. The implementation and performance measurements are described in Section 4.

2. ANALYSIS OF COMMONLY USED KERNELS

We examined 165 commonly used kernels in various image processing applications. The kernels were collected from different sources including several image processing textbooks [8–11] and image processing systems [12, 13]. They include most kernels used for edge enhancement (e.g., Sobel, Laplacian, and Prewitt) and for spatial domain filtering (e.g., high-pass filter, low-pass filter, and unsharp masking). Figure 1 shows several examples of the kernels selected. Several interesting properties of these kernels are summarized below.

Property 1. For most kernels, the number of distinct kernel elements is low.

Property 2. 0, 1, and −1 are used frequently.

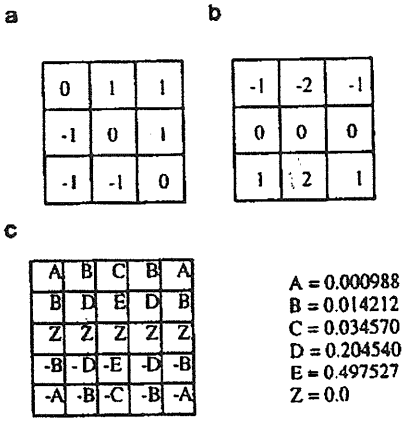

FIG. 1. Example kernels: (a) 3 × 3 gradient kernel (grad_135_degrees), (b) 3 × 3 Sobel operator (sobel_x), and (c) 5 × 5 Argyle's kernel (arg_0.75.h).

Property 3. Many kernel elements have the same absolute values.

Table 1 lists the analysis results of some of the kernels we examined. Property 1 can be verified by comparing the number of total kernel elements N_{total} column to the number of distinct kernel elements N_{distinct} column in Table 1. For example, a 3 × 3 gradient kernel (Fig. 1a) has only three distinct kernel elements. Property 2 is generally more evident in the integer kernels, though the floating-point (or floating) kernels use 0's fairly frequently as well. A 3 × 3 Sobel operator (Fig. 1b) has five distinct kernel elements, but after ignoring the trivial elements (0, 1, and −1), there are only two different kernel elements left. By comparing the N_{distinct} column and $N_{\text{nontrivial}}$ column (the number of distinct kernel elements excluding 0, 1, and −1) in Table 1, we can observe that a large number of kernels have at least one kernel element with the value of 0, 1, or −1. About 80% of kernels out of 165 examined use at least one of 0, 1, or −1. Property 3 is the direct result of the fact that many kernels are symmetric or antisymmetric. For example, a 5 × 5 Argyle's kernel (Fig. 1c) has 10 nontrivially distinct kernel elements, but considering the antisymmetry, there are only 5 kernel elements having absolutely distinct values. Figure 2 shows the distributions of the number of absolutely dis-

TABLE 1
Analysis of Commonly Used Kernels

Type	Name	N_{total}[a]	N_{distinct}[b]	$N_{\text{nontrivial}}$[c]	$N_{\text{abs-distinct}}$[d]	N_{zero}[e]
3 × 3	frei_chen_w9	9	1	0	0	0
	grad_135_degrees	9	3	0	0	3
	sobel_x	9	5	2	1	3
	unsharp3×3	9	2	1	1	0
	arg_0.5h	9	5	4	2	3
	hpf_3_int	9	3	3	2	4
	lpf_3	9	3	3	3	0
	pix_stack	9	9	8	8	0
5 × 5	boxcar_5_col	25	3	2	1	5
	trunc_pyr_5_row	25	5	4	2	5
	arg_0.75.h	25	11	10	5	5
	mar_0.6	25	6	5	5	0
	mac_0.6v	25	13	12	6	5
7 × 7	bxc_7h	49	3	2	1	7
	arg_1.0h	49	19	18	9	7
	mar_0.8	49	10	10	10	0
	dog_1.0v	49	25	24	12	7
	mac_0.8h	49	25	24	12	7

[a] N_{total} = the number of total kernel elements.
[b] N_{distinct} = the number of distinct kernel elements.
[c] $N_{\text{nontrivial}}$ = the number of distinct kernel elements excluding 0, 1, and −1.
[d] $N_{\text{abs-distinct}}$ = the number of kernel elements having distinct absolute values excluding 0, 1, and −1.
[e] N_{zero} = the number of zero elements.

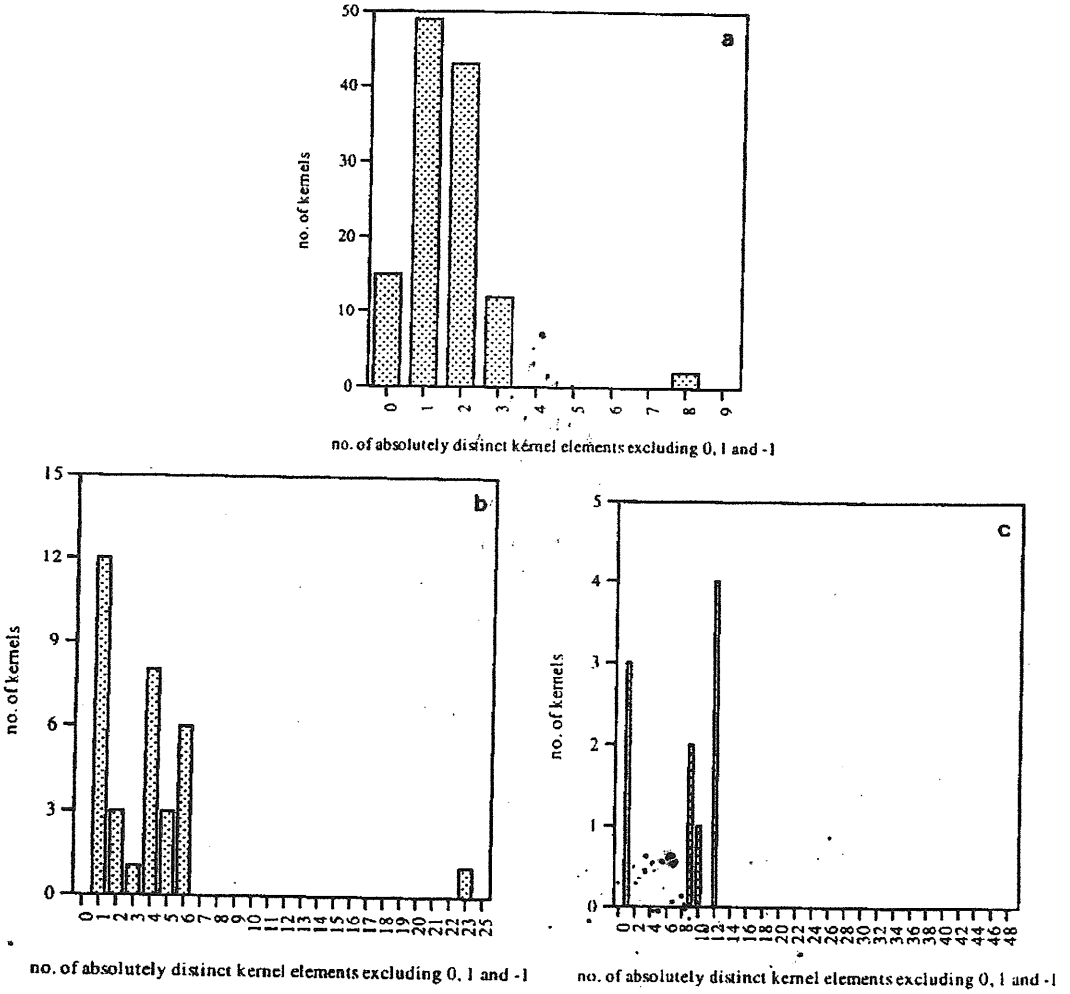

FIG. 2. Distribution of $N_{\text{abs-distinct}}$ among (a) 121 3 × 3 kernels, (b) 34 5 × 5 kernels and (c) 10 7 × 7 kernels.

tinct kernel elements ($N_{\text{abs-distinct}}$) among 3 × 3, 5 × 5, and 7 × 7 kernels examined. As shown in Fig. 2, the distributions are heavily skewed toward small values.

Property 2 has been used in reducing the number of multiplications in the direct implementation by skipping multiplications between input pixels and a kernel element which has a value of 0, 1, or −1. Properties 1 and 3 have the potential to reduce the number of multiplications further. But, in the direct implementation, it is not easy to utilize these properties efficiently, since pixel values being multiplied are generally different even though the kernel elements are of the same value. We have devised a new algorithm which takes full advantage of Properties 1–3. In our algorithm, the number of multiplications per convolved element is reduced to the value of $N_{\text{abs-distinct}}$ from the total number of kernel elements N_{total}, and the number of additions per convolved element is decreased by the number of zero elements N_{zero} with $O(p^2)$ extra memory.

3. SINGLE-DATA MULTIPLE KERNEL CONVOLUTION ALGORITHM

In this section, we present an efficient convolution algorithm which exploits the analysis results described in the previous section. For the purpose of illustration, we first consider the case of 1-D. Figure 3 shows the sequence of computations for the direct implementation of the 1 × 3 convolution operation. Each step produces a final convolved element for the current location. Since the portion of an input sequence which is multiplied by the kernel is changing with each step, we would not be able to utilize Properties 1 and 3 described above.

Our algorithm is different from the direct implementation in two ways. First, each step computes partial sums for the multiple locations. Once the first convolved element is computed, each subsequent step produces a new convolved element. Second, in each step, all the kernel elements are multiplied by the same input data. We call

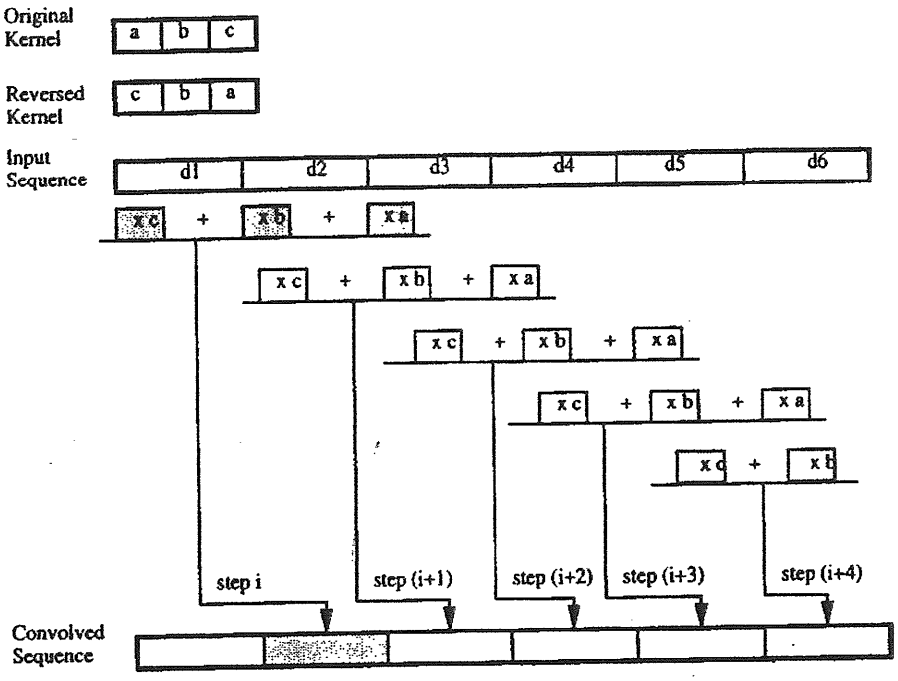

FIG. 3. Direct implementation of convolution operation.

our approach *a single-data multiple kernel* (SDMK) *method* in order to emphasize that the algorithm works on a single pixel with the multiple kernel elements. Figure 4 illustrates the key idea of the SDMK approach. Each step works with a single pixel (e.g., d3 for the step $(i + 2)$), and the computed partial sums are distributed to the appropriate locations. The SDMK approach could be easily modified to take full advantage of Properties 1–3, since all of the kernel elements are multiplied by the same data element.

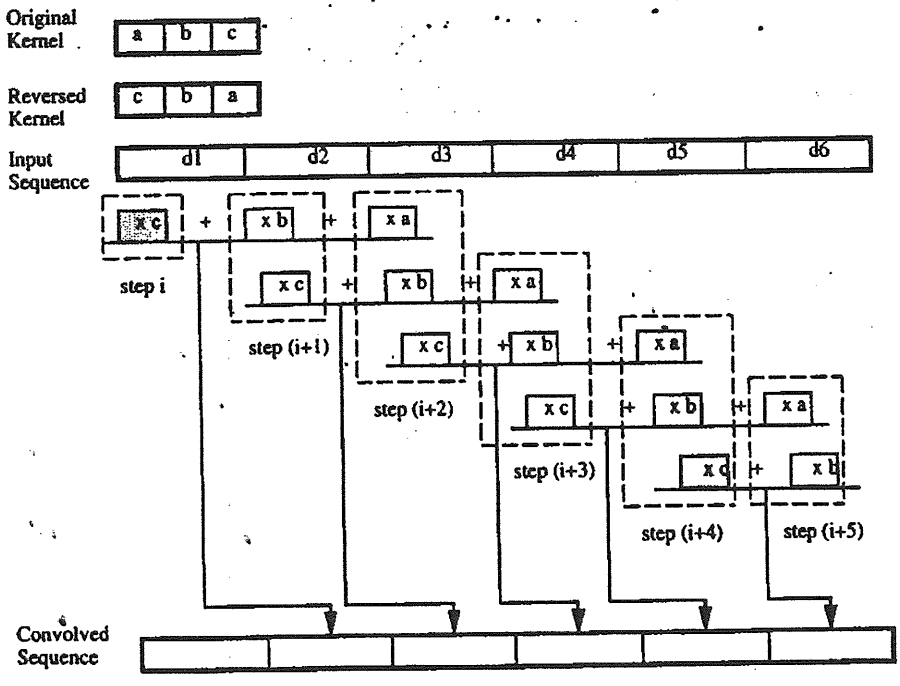

FIG. 4. Single-data multiple kernel (SDMK) implementation of convolution operation.

3.1. *Algorithm Description*

Figure 5 shows the steps involved in the 2-D SDMK convolution algorithm. The complete description of the convolution algorithm is given in Fig. 6 while Fig. 7 describes two auxiliary routines used in the implementation of the SDMK algorithm. The algorithm consists of two parts. The first part analyzes the kernel and rearranges it for the SDMK processing. The analysis partitions the kernel elements into groups based on their absolute values, and within each group, they are classified into two lists: a positive list and a negative list. If the value of a kernel element is positive, it is added to the positive list, and if negative, it is added to the negative list. The lists are maintained using standard linked list data structures. Each group is represented by a LinkedListHeader structure, which has three fields: a value v and two pointers, PosPtr and NegPtr. The PosPtr maintains the kernel elements of value v, and the NegPtr points the list of the kernel elements of value $-v$. For each kernel element, we allocate a LinkedListElement structure with three fields: xoffset, yoffset, and next. The xoffset and yoffset fields contain the positional information of a kernel element within the input kernel K. The reference point for the xoffset and yoffset fields is the lower right corner of the kernel, i.e., the kernel element in the lower right corner of the kernel has 0 for both the xoffset and yoffset fields. The kernel element in the upper-left corner has $-(p-1)$ for both the xoffset and yoffset fields. These fields are used to locate the appropriate location when the partial sum is accumulated. The next field points to the next kernel element having the same kernel value. Two special pointers (OneList and MinusOneList) to the linked lists are maintained as well. OneList points to the kernel elements having a value of 1 and MinusOneList points to the kernel elements having a value of -1.

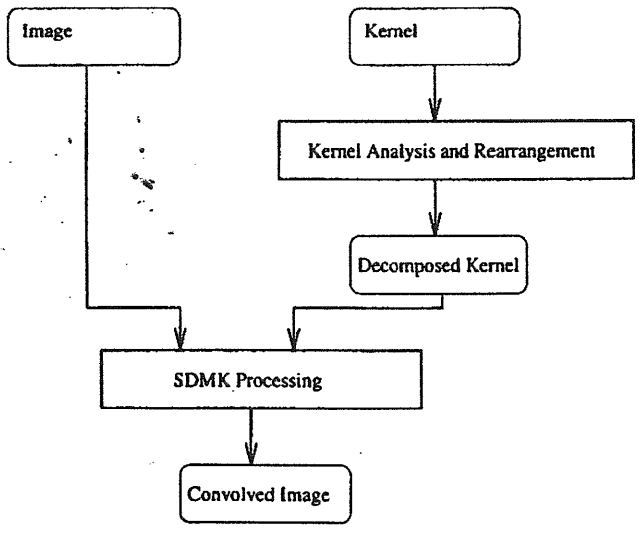

FIG. 5. Processing steps involved in the 2-D SDMK convolution algorithm.

After the kernel is analyzed and rearranged, the actual convolution computation is performed next. For each pixel $A[i, j]$, the kernel value field *Sorted Kernel*$[i].v$ of the LinkedListHeader array is multiplied by $A[i, j]$. The multiplication result is added to the appropriate locations of the convolution output image *ConvolvedMatrix* depending on the xoffset and yoffset values of the kernel elements in both positive list *SortedKernel*$[i]$.*PosPtr* and negative list *SortedKernel*$[i]$. *NegPtr*. The convolution output image *ConvolvedMatrix* is initialized to zero. For the kernel elements in the linked lists pointed by OneList and MinusOneList pointers, the multiplication step is skipped. The zero kernel elements are automatically excluded from the computation, since they are not included in any linked list in the first stage. Figure 8 shows the result of the preprocessing steps with the 5×5 Argyle kernel described in Fig. 1c.

3.2. *Algorithm Analysis*

The SDMK implementation requires $N_{\text{abs-distinct}}$ multiplications and $p^2 - N_{\text{zero}}$ additions per each convolved element. The number of multiplications is not dependent on N_{total} because the multiplication result is reused for all the kernel elements in each linked list. Since we are excluding zero elements during the analysis stage, the number of additions is reduced to $p^2 - N_{\text{zero}}$ as well.

The SDMK implementation requires extra $O(p^2)$ time and space in order to preprocess the kernel elements and maintain the sorted lists of kernel elements, but this overhead is negligible compared with the convolution operation itself, since $p^2 \ll nm$. If memory is not a critical resource or $N_{\text{abs-distinct}}$ of a kernel is very small, we could further improve the SDMK implementation by using a table lookup technique which is widely used in the implementation of multiplier-free signal processors [14]. We could precompute the multiplications for all the possible combinations of pixel and kernel values and store the results in memory as a lookup table. If there are $N_{\text{abs-distinct}}$ absolutely distinct kernel elements and $2^{d_{\text{image}}}$ possible pixel values, we would need $O(2^{d_{\text{image}}} N_{\text{abs-distinct}})$ extra memory, where d_{image} is the depth of an image in the number of bits. For example, if there are two absolutely distinct kernel elements ($N_{\text{abs-distinct}} = 2$), the input image has 8 bits per pixel ($d_{\text{image}} = 8$), and each multiplication result requires 4 bytes (a single-precision floating-point number) for storage, 2 Kbytes of lookup table memory would be necessary. Once the lookup table is formed after $2^{d_{\text{image}}} N_{\text{abs-distinct}}$ multiplications, the convolution operation could be performed with memory accesses and additions only.

Single Data Multiple Kernel Convolution Algorithm

Given: an nxm matrix A and a pxp kernel K

Find: an nxm convolved matrix B of a matrix A with a kernel K

```
begin

  LinkedListHeader SortedKernel[MAXNOKERNELELMTS];
      Each element SortedKernel[i] of SortedKernel array maintains three fields: value v of a kernel element,
      pointer PosPtr to the list of positive kernel elements (value) and pointer NegPtr to the list of negative
      kernel elements (-value).
  LinkedListPtr OneList, MinusOneList;
      OneList points to the list of kernel elements equal to 1 and MinusOneList points to the list of kernel elements
      equal to -1.
  Integer NoElmtsInSortedKernel;
      the number of kernel elements having the distinct absolute values excluding 0, 1 and -1
  Matrix ConvolvedMatrix;
      ConvolvedMatrix holds the convolution results.

  comment : Rearrange p^2 kernel elements such that all the kernel elements having the same
            absolute values are grouped together.

  for i := 1 to p step 1 do
      for j := 1 to p step 1 do
          UpdateSortedListOfKernelElements(K[i,j], i, j);
      od
  od


  comment : For each element A[i,j] of a matrix A, multiply absolutely distinct kernel elements and
            accumulate the partial sum into the appropriate elements of ConvolvedMatrix array.

  for i := 1 to n step 1 do
      for j := 1 to m step 1 do
          if NotEmpty(OneList) then AccumulatePartialSum(A[i,j], OneList, i, j); fi;
          if NotEmpty(MinusOneList) then AccumulatePartialSum(-A[i,j], OneList, i, j); fi;
          for k := 1 to NoElemtsInSortedKernel step 1 do
              if NotEmpty(SortedKernel[k].PosPtr) then
                AccumulatePartialSum(A[i,j] * SortedKernel[k].v, SortedKernel[k].PosPtr, i, j); fi;
              if NotEmpty(SortedKernel[k].NegPtr) then
                AccumulatePartialSum(-A[i,j] * SortedKernel[k].v, SortedKernel[k].NegPtr, i, j); fi;
          od
      od
  od
end
```

FIG. 6. SDMK convolution algorithm.

```
proc UpdateSortedListOfKernelElements(elmt, i, j) ≡
    do if elmt = 0 then return fi;
       for k := 1 to NoElmtsInSortedKernel step 1 do
           if elmt = 1 then UpdateOneList(); return fi
           if elmt = -1 then UpdateMinusOneList(); return fi
           if elmt = SortedKernel[k].v then UpdatePosPtr(SortedKernel[k]); return
           elsif elmt = -SortedKernel[k].v then UpdateNegPtr(SortedKernel[k]); return fi
       od
       AllocateNewLinkedListHeaderAndElement(elmt, i, j);
       NoElmtsInSortedKernel := NoElmtsInSortedKernel + 1;
       return
    od.


proc AccumulatePartialSum(psum, ptrtolinkedlist, x, y) ≡
    ptr := ptrtolinkedlist;
    while ptr ≠ NULL do
          Xoffset := *ptr.xoffset;
          Yoffset := *ptr.yoffset;
          ConvolvedMatrix[x + Xoffset, y + Yoffset] := ConvolvedMatrix[x + Xoffset, y + Yoffset] + psum;
          ptr := *ptr.next;
    od.
```

FIG. 7. Auxiliary routines for the SDMK convolution algorithm.

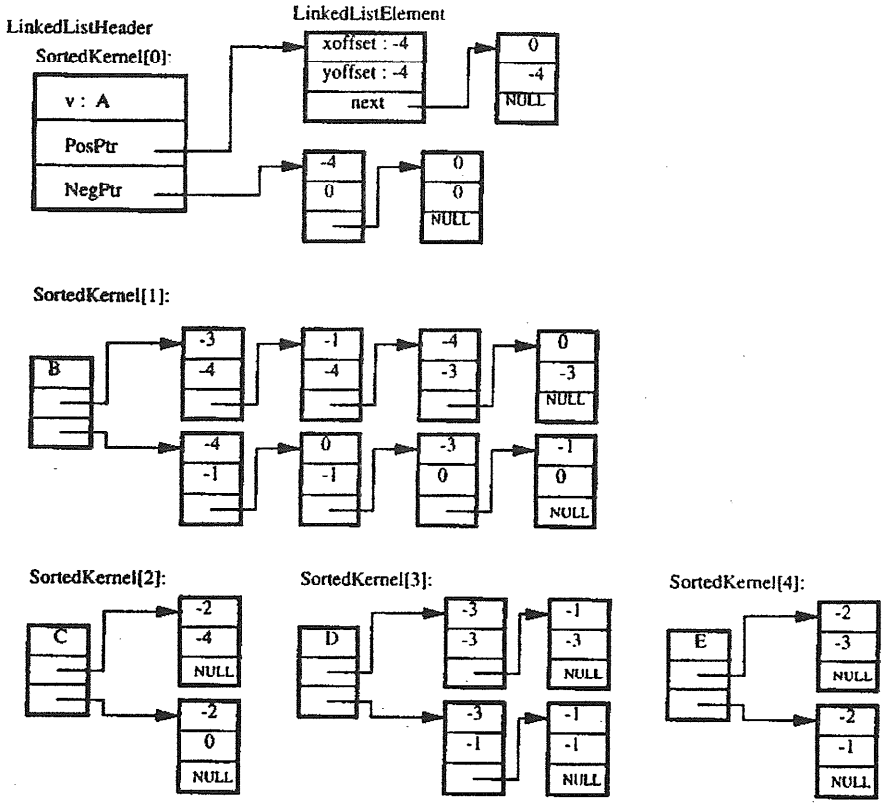

FIG. 8. Result of the preprocessing steps with the 5 × 5 Argyle kernel shown in Fig. 1c.

4. EXPERIMENT

We have implemented the SDMK convolution algorithm on SUN SPARCstation 10 using C programming language. In order to compare the performance over the direct implementation, we used the convolution routine from the Khoros image processing system [12] which is widely available and used.

We measured the execution time for both the integer and floating-point kernels under the carefully controlled measurement environment, and programs were compiled using the GNU C compiler (gcc) with the optimization option turned on. For each kernel, we made 1500 measurements and adopted the average of 1500 values as the execution time for the kernel. Since the performance of the SDMK and direct convolution algorithms does not depend on pixel values, a single 512 × 480 image was used as an input image for all the convolution routines.

Tables 2 and 3 summarize the measurement results with selected integer and floating-point kernels. Even considering the several detailed implementation differences between the SDMK and direct implementations (e.g., how to compute the indexes for accessing both kernels and image data), Tables 2 and 3 clearly show the substantial performance increase with the SDMK implementation over the direct implementation. For most kernels, the speed-up factor of between 2 to 7 is achieved. With the integer kernels, the performance increase is influenced by the number of absolutely distinct kernel elements $N_{\text{abs-distinct}}$, while the effect of the number of zero elements N_{zero} is somewhat less. For example, in Table 2, cubic has four more zero elements and two more absolutely distinct elements than nev_bau_60, and the execution time of nev_bau_60 is shorter than that of cubic.

TABLE 2
Performance Results with Integer Kernels

Type	Name	$N_{\text{abs-distinct}}$	N_{zero}	$t_{\text{direct}}{}^a$ (s)	$t_{\text{sdmk}}{}^b$ (s)	Speed-up[c]
3 × 3	frei_chen_w5_int	0	5	2.23	0.31	7.15
	grad_135_degrees	0	3	2.23	0.42	5.30
	sobel_x	1	3	2.23	0.56	4.00
	hpf_3_int	2	0	2.23	0.79	2.82
5 × 5	nev_bau_0	1	5	5.35	1.12	4.77
	nev_bau_60	4	1	5.35	1.71	3.12
	cubic	6	5	5.35	1.81	2.95

[a] t_{direct} = an execution time measured by the direct implementation.
[b] t_{sdmk} = an execution time measured by the SDMK implementation.
[c] Speed-up = $t_{\text{direct}}/t_{\text{sdmk}}$.

TABLE 3
Performance Results with Floating-Point Kernels

Type	Name	$N_{abs\text{-}distinct}$	N_{zero}	t_{direct}[a] (s)	t_{sdmk}[b] (s)	Speed-up[c]
3 × 3	arg_0.5h	2	3	1.98	0.66	3.00
	unsharp3×3	1	0	1.98	0.72	2.75
	kirsch_2	3	1	1.98	0.84	2.36
	lpf_3	3	0	1.98	0.89	2.21
5 × 5	box_car_5_col	1	5	4.63	1.33	3.47
	arg_0.75.h	5	5	4.63	1.76	2.62
	mac_0.6v	6	5	4.63	1.89	2.45
	mar_0.6	5	0	4.63	2.00	2.31
7 × 7	bxc_7h	1	7	8.56	2.69	3.18
	arg_1.0h	9	7	8.56	3.64	2.35
	dog_1.0v	12	7	8.56	3.85	2.22
	mar_0.8	10	0	8.56	4.00	2.14

[a] t_{direct} = an execution time measured by direct implementation.
[b] t_{sdmk} = an execution time measured by SDMK implementation.
[c] Speed-up = t_{direct}/t_{sdmk}.

With floating-point kernels, the performance is less dependent on $N_{abs\text{-}distinct}$ and the significance of N_{zero} increases. This can be observed in mac_0.6v and mar_0.6 in Table 3. While mac_0.6v has one more absolutely distinct element than mar_0.6, the execution time of mac_0.6v is shorter than that of mar_0.6 because of the large number of zero elements.

5. CONCLUSION

A new algorithm in efficiently implementing 2-D convolution operation is presented. It is based on the common properties of the widely used kernels in many image processing applications. Unlike the specialized optimization techniques such as separable kernels and moving average method, our algorithm is applicable to a wide range of kernels. The efficiency of the algorithm comes from the intelligent analysis and management of the kernel elements and SDMK approach in computing the convolution results. The SDMK convolution algorithm was implemented on the SUN workstations and its performance was compared with that from direct implementation. From the test runs, we have achieved a speed-up factor of two to seven over the direct implementation. The extra memory and preprocessing requirements are proportional to the kernel size which is usually negligible compared with the input image size.

REFERENCES

1. LSI Logic Corporation, L64240 Multi-bit filter, in *Digital Signal Processing (DSP) Databook*, pp. 99–123, LSI Logic Corporation, 1991.
2. K. Guttag, R. J. Gove, and J. R. Van Aken, A single-chip multiprocessor for multimedia: The MVP, *IEEE Comput. Graphics Appl.* **12**(6), 1992, 53–64.
3. K. S. Mills, G. K. Wong, and Y. Kim, A high performance floating-point image computing workstation for medical applications, in *Proceedings, SPIE Medical Imaging IV: Image Capture and Display, Newport Beach, 1990*, Vol. 1232, pp. 246–256.
4. K. Konstantinides and V. Bhaskaran, Monolithic architectures for image processing and compression, *IEEE Comput. Graphics Appl.* **12**(6), 1992, 75–86.
5. D. K. Yee, D. R. Haynor, H. S. Choi, S. W. Milton, and Y. Kim, Development of a prototypical PACS workstation based on the IBM RS6000 and the X Window system, in *Proceedings, SPIE Medical Imaging VI: Image Capture, Formatting, and Display, Newport Beach, 1992*, Vol. 1653, pp. 337–348.
6. H. C. Andrews and J. Kane, Kronecker products, computer implementation and generalized spectra, *J. Assoc. Comput. Mach.* **17**(2), 1970, 260–268.
7. M. J. McDonnell, Box-filtering techniques, *Comput. Graphics Image Process.* **17**, 1981, 65–70.
8. B. Jähne, *Digital Image Processing: Concepts, Algorithms, and Scientific Applications*, Springer-Verlag, Berlin/New York, 1991.
9. R. Gonzalez and R. E. Woods, *Digital Image Processing*, Addison–Wesley, Reading, MA, 1992.
10. R. Haralick and L. Shapiro, *Computer and Robot Vision*, Vol. 1, Addison–Wesley, Reading, MA, 1992.
11. W. Pratt, *Digital Image Processing*, 2nd ed., Wiley, New York, 1991.
12. J. Rasure, D. Argiro, T. Sauer and C. Williams, Visual language and software development environment for image processing, *Int. J. Imaging Systems Technol.* **2**, 1990, 183–199.
13. SunVision 1.1, Sun Microsystems Inc., 1990.
14. A. Peled and B. Liu, A new hardware realization of digital filters, *IEEE Trans. Acoust. Speech, Signal Process.* **ASSP-22**(6), 1974, 456–462.