Fast Vectorization for Calculating a Moving Sum

Kuo-Liang Chung, *Member, IEEE*, and Wen-Ming Yan

***Abstract*—A simple vectorized method for calculating a moving sum is developed. Our proposed method is suitable for register-to-register vector computers and entails much less redundant floating-point operations than the vectorized algorithm of Mossberg [3]. We demonstrate the performance of our vectorized algorithm on the CRAY X-MP EA/116se supercomputer.**

***Index Terms*—Automatic gain control, CRAY X-MP, Fortran, moving sum, prefix sum, vectorization.**

I. Introduction

Among many methods used for smoothing and normalization of seismic traces, the automatic gain control (AGC) method is the best known. Commonly, more than 10-20% of the total time used for a seismic processing sequence is spent in an AGC subroutine. An essential part of the computations involved in the AGC method is the calculation of a *moving sum*. Given an input vector $A = (a_i)$ for $1 \le i \le n$ and a window of length w ($\ll n$), a moving sum calculation generates an output vector $B = (b_j)$ for $1 \le j \le n$, where

$$b_j = \sum_{t=1}^{w} a_{j-t+1} \ (a_i = 0 \text{ outside } 1 \le i \le n). \qquad (1)$$

By replacing the summation operator with an absolute–value summation operator, the moving sum of absolute values can be obtained directly. Although in [3], the window of odd length w used to compute the jth output value is centered around the jth element of the input vector, by adjusting the initial index, the computation of (1) can be obtained exactly the same computation as in [3].

On a vector machine, the output vector B as defined by (1) can be straightforwardly computed using the sequential method called the scalar algorithm (SA) [3]. The number of floating-point (FP) operations required in SA is about $2n$ but the SA is not a vectorization approach. Mossberg [3] first presented a vectorized moving sum algorithm for the CYBER 205 memory-to-memory supercomputer in which the floating-point functional units can communicate directly with main memory to receive and transfer data. Mossberg's vectorized algorithm can be accomplished by means of $s + t$ vector operations, and each needs operands of vector length n, where $2^s < w < 2^{s+1}$ and t is defined as the number of 1s in the binary representation of w. Totally, the number of FP operations required is $(s + t)n$ which ranges from $(\lceil \log w \rceil + 1)n$ to $(2\lceil \log w \rceil - 1)n$, where the logarithm is base two and $\lceil\,\rceil$ denotes the ceiling function. Since the concerning vector is of length n and with stride 1, Mossberg's algorithm is particular for the memory-to-memory supercomputer [2]. Nowadays, except for the CYBER 205 and ETA 10 all other vector computers are register-to-register machines such as the CRAY series, Fujitsu VP series, Hitachi S series, and NEC SX series [1].

The purpose of this paper is the design of a new vectorized moving sum algorithm for the register-to-register vector computers. The number of FP operations required in our proposed method is shown to be $(2 + (w - 4)/k)n$, which ranges from $2n$ to $2.5n$, where the value of k ($\ge 2w$) is determined according to the partition of the array. We show that the ratio of the number of FP operations required in our algorithm over Mossberg's algorithm ranges from $(\lceil \log w \rceil + 1)/2.5$ to $\lceil \log w \rceil - 1/2$. For the case $w > 4$, our method entails much less redundancy than the vectorized algorithm of Mossberg [3]. To alleviate the memory-bank conflicts, the value of stride in our program can be selected as an odd number since the number of memory banks is even in CRAY X-MP EA/116se. We demonstrate the performance of our vectorized algorithm on this supercomputer.

The rest of the paper is organized as follows. In Section II, we describe the proposed vectorized algorithm for computing a moving sum. It also provides the complexity analysis. In Section III, we present some experimental results that examine the performance of our method. Section IV concludes the paper.

II. A Vectorized Moving Sum Algorithm

Recall that the input vector is $A = (a_1, a_2, ..., a_n)$ and the output vector is $B = (b_1, b_2, ..., b_n)$. First we partition A into pq groups of k elements. $(a_1, a_2, ..., a_k)$ $(k \ge 2w)$ constitutes the first group, $(a_{k+1}, a_{k+2}, ..., a_{2k})$ constitutes the second group, and so on. If kpq ($= m$) $> n$, then a_i for $i > n$ is assigned to zero. We partition B in the same way. For convenience, we assume that $kpq = n$. In order to reveal the new structure of A and B, let us rearrange A and B into two $q \times k$ block matrices by

$$A = \begin{pmatrix} a_1 & a_2 & \cdots & a_k \\ a_{k+1} & a_{k+2} & \cdots & a_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ a_{n-k+1} & a_{n-k+2} & \cdots & a_n \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,k} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,k} \\ \cdots & \cdots & \cdots & \cdots \\ A_{q,1} & A_{q,2} & \cdots & A_{q,k} \end{pmatrix}$$

and

$$B = \begin{pmatrix} b_1 & b_2 & \cdots & b_k \\ b_{k+1} & b_{k+2} & \cdots & b_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ b_{n-k+1} & b_{n-k+2} & \cdots & b_n \end{pmatrix} = \begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,k} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,k} \\ \cdots & \cdots & \cdots & \cdots \\ B_{q,1} & B_{q,2} & \cdots & B_{q,k} \end{pmatrix},$$

where $A_{i,j}$ and $B_{i,j}$, $1 \le i \le q$ and $1 \le j \le k$, are $p \times 1$ matrices, i.e., $A_{i,j} = (a_{(i-1)pk+j}, a_{(i-1)pk+j+k}, \ldots, a_{(i-1)pk+j+(p-1)k})^t$ and $B_{i,j} = (b_{(i-1)pk+j}, b_{(i-1)pk+j+k}, \ldots, b_{(i-1)pk+j+(p-1)k})^t$.

For $1 \le i \le q$, by (1), it follows that

$$B_{i,j} = \sum_{t=1}^{w} A_{i,j-t+1} \text{ for } w \le j \le k$$

and

$$B_{i,j} = \sum_{t=1}^{j} A_{i,t} + \sum_{t=k-(w-j)+1}^{k} \overline{A}_{i,t} \text{ for } 1 \le j < w,$$

where $\overline{A}_{i,j} = (a_{(i-1)pk+j-k}, a_{(i-1)pk+j}, \ldots, a_{(i-1)pk+j+(p-2)k})^t$. Notice that $a_i = 0$ outside $1 \le i \le n$ (see (1)).

Given a set of inputs $x_1, x_2, ..., x_n$ and a summation operator +, find all partial sums

$$x_1,\ x_1 + x_2,\ \ldots,\ \sum_{i=1}^{n} x_i .$$

This problem is known as the prefix sum problem. Similarly, find all partial sums:

Manuscript received Sept. 23, 1993; revised June 1, 1994.

K.-L. Chung is with the Department of Information Management, National Taiwan Institute of Technology, Taipei, Taiwan 10672, Republic of China; e-mail: klchung@cs.ntit.edu.tw.

W.-M. Yan is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 10764, Republic of China.

To order reprints of this article, e-mail: transactions@computer.org, and reference IEEECS Log Number C95110.

$$\sum_{i=1}^{n} x_i, \sum_{i=2}^{n} x_i, \ldots, x_{n-1} + x_n, x_n.$$

This problem is known as the suffix sum problem.

Based on some prefix-sum and suffix-sum operations in vectorized ways, our *vectorized moving sum* algorithm works as follows. Consider the first band of A

$$A_{1,1}, A_{1,2}, \ldots, A_{1,k}.$$

First, we compute the following suffix–sum computations by means of $(w-2)$ vector summations:

$$C_1 \leftarrow \sum_{i=k-w+2}^{k} \overline{A}_{1,i}, C_2 \leftarrow \sum_{i=k-w+3}^{k} \overline{A}_{1,i}, \ldots, C_{w-2} \leftarrow \overline{A}_{1,k-1} + \overline{A}_{1,k}, C_{w-1} \leftarrow \overline{A}_{1,k},$$

where the symbol '$\leftarrow$' denotes an assignment operator. Equivalently, the above suffix-computations can be performed by the following vectorized process, where each vector summation takes operands of vector length p.

$$C_{w-1} \leftarrow \overline{A}_{1,k}$$

For $i = w-2$ **downto 1 do**

$$C_i \leftarrow C_{i+1} + \overline{A}_{1,k-w+i+1}$$

For the vectors $A_{1,1}, A_{1,2}, \ldots, A_{1,k}$, we compute the prefix sum by means of $(k-1)$ vector summations, and each needs operands of vector length p. We then obtain all partial–sum vectors

$$D_1 \leftarrow A_{1,1}, D_2 \leftarrow A_{1,1} + A_{1,2}, \ldots, D_k \leftarrow \sum_{i=1}^{k} A_{1,i}.$$

Thereafter, for i from k down to $(w+1)$, we do $D_i \leftarrow D_i - D_{i-w}$. It takes $(k-w)$ vector subtractions, and each also needs operands of vector length p. For i from 1 to $(w-1)$, we do $D_{w-i} \leftarrow D_{w-i} + C_{w-i}$. Then, the moving sum of the first band of A is flowed from D_i for $1 \le i \le k$. That is, the values of b_j of (1) for $1 \le j \le kp$ have been determined. If $q = 1$ then we stop the algorithm; otherwise the following do-loop is performed.

For $j = 2$ to q **do**

Step_1. Compute the suffix sum for the vectors

$$\overline{A}_{j,k-w+2}, \overline{A}_{j,k-w+3}, \ldots, \overline{A}_{j,k}.$$

We then obtain all partial-sum vectors

$$C_1 \leftarrow \sum_{i=k-w+2}^{k} \overline{A}_{j,i}, C_2 \leftarrow \sum_{i=k-w+3}^{k} \overline{A}_{j,i}, \ldots, C_{w-2} \leftarrow \overline{A}_{j,k-1} + \overline{A}_{j,k}, C_{w-1} \leftarrow \overline{A}_{j,k}.$$

Step_2. Compute the prefix sum for the vectors $A_{j,1}, A_{j,2}, \ldots, A_{j,k}$. Thus, we obtain all partial-sum vectors

$$D_1 \leftarrow A_{j,1}, D_2 \leftarrow A_{j,1} + A_{j,2}, \ldots, D_k \leftarrow \sum_{i=1}^{k} A_{j,i}.$$

Step_3. For i from k down to $w+1$, we perform $D_i \leftarrow D_i - D_{i-w}$.

Step_4. Perform the additions: $D_{w-i} \leftarrow D_{w-i} + C_{w-i}$ for $1 \le i \le w-1$. The moving sum is flowed from D_i, $1 \le i \le k$.

Enddo

After completing the above vectorized algorithm for calculating a moving sum of A, the total number of FP operations needed in the suffix-sum computations of Step_1 is $pq(w-2)$; the number of FP operations needed in the prefix-sum computations of Step_2 is $pq(k-1)$; the number of FP operations needed in Step_3 is $pq(k-w)$; the number of FP operations needed in Step_4 is $pq(w-1)$. So it takes $(2 + (w-4)/k)n$ $(= pqw - 4pq + 2pqk)$ FP operations to finish computing a moving sum of A. The number of FP operations used in our algorithm ranges from $2n$ to $2.5n$, while the number of FP operations used in Mossberg's method ranges from $(\lceil \log w \rceil + 1)n$ to $(2\lceil \log w \rceil - 1)n$. For the case $w > 4$, our method entails much less redundant FP operations than the vectorized algorithm of Mossberg [3]. For example, if $w = 65$ and $k = 10w$, our method needs about $2.1n$ FP operations, but Mossberg's method needs FP operations ranging from $8n$ to $15n$. The approach of partitioning into blocks makes our vectorized method suitable for either the vector computer with multiple vector elements or for on-line reading of the input vector.

III. Implementations on Cray X-MP EA/116se

In this section, we implement our vectorized moving sum algorithm on the Cray X-MP EA/116se supercomputer. Before illustrating the corresponding experimental results, let us introduce some features of this machine. This machine has a register–to–register architecture without cache memory and has one vector processor which contains eight 64-bit vector registers of length 64. Memory is divided into 16 banks and each bank contains 1M 64-bit words.

The input vector A is generated by a random number generator, a function call **ranf**(). The length of the vector A is specified to be 10,000, 20,000, 30,000, ..., and 90,000, respectively. The Cray Fortran 77 source code of our vectorized algorithm called **movesum1** is listed in the Appendix. Table I shows the performance of running our vectorized algorithm. The operating system used here is UNICOS 6.1.6 and the compiler is called CF77.

TABLE I
Cray X–MP EA/116se Execution Times for Our Algorithm

n	w	time	m	k	q
10,000	11	0.483*mi*	10,176	53	3
20,000	11	0.960*mi*	20,352	53	6
30,000	21	1.451*mi*	30,400	95	5
40,000	21	1.941*mi*	40,768	91	7
50,000	31	2.482*mi*	50,304	131	6
60,000	31	2.849*mi*	60,480	135	7
70,000	41	3.365*mi*	70,272	183	6
80,000	41	3.852*mi*	80,192	179	7
90,000	51	4.223*mi*	90,240	235	6

In Table I, the symbols n, w, q, and mi denote the size of A, the size of the window, the number of the bands in A, and millisecond, respectively. m and k have been defined in Section II. To alleviate the memory-bank conflicts, the value of k is selected as an odd number and is greater than $4w$.

IV. Conclusions

We have presented the design of a new vectorized moving sum algorithm for register-to-register vector computers. Our algorithm is not only more efficient than Mossberg's, but also based on a simpler idea which could be applied to pattern matching problems. Some experimental results for our method have been obtained on the CRAY X-MP EA/116se supercomputer. In addition, due to the approach of partitioning into blocks, our vectorized algorithm is suitable for either the vector computer with multiple vector elements or for on–line reading of the input vector.

Appendix

```
C--Our vectorized algorithm for moving sum---
   Program movesum1
C--b:initially save seismic vector; eventually
   save the moving-sum vector-
C--bb: save one copy of seismic vector used for
   the brute force method -
```

```
C--bbb: save the moving-sum vector of bb by
  brute force method -
C--c: temporary array for boundary processing—
  real b(100000), bb(100000), bbb(100000),
    c(100000), sum
  real starttime,totaltime
C--w: window size; n: length of seismic vector--
  integer i,j,k,n,m,p,ww,w
  integer q,nt,kk,l
C--array wt is used for boundary processing be-
  tween two consecutive bands-
  real wt(1000)
  write(*,*) 'INPUT N:'
  read(*,*) n
  write(*,*) 'INPUT w:'
  read(*,*) w
c--the vector length is 64--
  p=64
C--generate random seismic vector—
  do 5 i=1,n
    b(i)=range*ranf()
    bb(i)=b(i)
5 continue
C--start timing—
  starttime=SECOND()
  k=4*w+1
  kk=k*p
C--q=ceiling function of (n/kk)--
  q=(n+kk-1)/kk
  if (q.gt.1) then
    q=q-1
    k=(n+64*q-1)/(64*q)
    k=2*(k/2)+1
    kk=k*p
  endif
  m=kk*q
  ww=((w+1)/2)*2-1
  do 8 i=n+1,m
    b(i)=0.0
8 continue
  wt(1:w-1)=0.0
  nt=0
  do 200 l=1,q
C--calculate array c—
  do 10 i=1,p
    c((i-1)*ww+w-1)=b(i*k+nt)
10  continue
  do 20 j=w-2,1,-1
cdir $ ivdep
  do 30 i=1,p
    c((i-1)*ww+j)=c((i-1)*ww+j+1)+b(i*k+j+nt-
    w+1)
30  continue
20  continue
C--prefix sum for one band—
  do 40 j=2,k
cdir $ ivdep
  do 50 i=1,p
    b((i-1)*k+j+nt)=b((i-1)*k+j+nt)+b((i-1)*k+j-
    1+nt)
50  continue
40  continue
C--calculate partial moving sums—
  do 60 j=k,w+1,-1
cdir $ ivdep
    do 70 i=1,p
      b((i-1)*k+nt+j)=b((i-1)*k+nt+j)-b((i-
      1)*k+nt+j-w)
70  continue
60  continue
C--boundary processing for (p-1) pair of con-
   secutive rows—
   do 71 j=1,w-1
     b(j+nt)=b(j+nt)+wt(j)
71  continue
   do 80 j=1,w-1
cdir $ ivdep
    do 90 i=2,p
      b((i-1)*k+nt+j)=b((i-1)*k+nt+j)+c((i-
      2)*ww+j)
90  continue
80  continue
  do 72 j=1,w-1
    wt(j)=c((p-1)*ww+j)
72  continue
  nt=nt+kk
200  continue
C--end of timing—
  totaltime=SECOND()-starttime
  print *, 'OUTPUT FOR MOVSUM1.F'
  print *,'n=,'n,' m=,'m
  print *,'CPU TIME FOR OURS=,'totaltime
  print *,'w=,'w,' k=,'k,' q=,'q
C--calculate moving sums by brute force method—
C--which takes (w-1)n FP operations and is used
  to verify the result-
  bbb(1)=bb(1)
  do 100 i=2,w
    bbb(i)=bbb(i-1)+bb(i)
100  continue
  do 110 i=w+1,m
    sum=0.0
    do 120 j=i-w+1,i
      sum=sum+bb(j)
120 continue
    bbb(i)=sum
110 continue
C--calculate the difference between our method
  and the brute force method-
C--by sup-norm measurement--
  sum=0.0
  do 130 i=1,m
    if (sum.lt.abs(bbb(i)-b(i))) then
      sum=abs(bbb(i)-b(i))
    endif
130 continue
  write(*,*) 'The difference is ,' sum
  end
```

ACKNOWLEDGMENTS

The authors appreciate the anonymous referees and Dr. Kornerup for their constructive comments that helped to improve the paper.

This research was supported in part by the National Science Council of the Republic of China under Grants NSC83-0408-E011-008 and NSC84-0408-E011-011.

REFERENCES

[1] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, Chapter 4: "Pipeline computers and vectorization methods." New York: McGraw-Hill, 1984.

[2] J.M. Levesque and J.W. Williamson, *A Guidebook to Fortran on Supercomputers*, Section 2.2.1: "Memory-to-memory vector processors," pp. 28-35. New York: Academic Press, 1989.

[3] B. Mossberg, "Vectorization of the calculation of a moving sum," *IEEE Trans. Computers*, vol. 36, pp. 362-365, Mar. 1987.

[4] "Supercomputer programming (I): Advanced Fortran: Architecture, vectorization, and parallel computing," Working manual for CRAY X-MP EA/116se, 1991.