MULTI-OPERAND ASSOCIATIVE ARITHMETIC

Isaac Scherson and Smil Ruhman
The Weizmann Institute of Science
Department of Applied Mathematics
Rehovot, Israel

ABSTRACT

Multi-operand associative techniques attain their full power in algorithms where the data may be recast into disjoint data sets, all acted upon concurrently, each by a different operand common to the set. But the multi-operand approach can also serve to enhance arithmetic operations significantly. The speed-up of associative multiplication by handling a number of multiplier bits at a time is described and analyzed, including an effective algorithm for a limited sum of products. The most complex process treated is convolution, which serves to illustrate the enhancement of an extended sum of products. Any number of vectors stored in memory can be convolved simultaneously by a common filter vector. Execution time is 45 milliseconds for 1024 element data and filter vectors, 2048 element results, and 16-bit precision.

INTRODUCTION

Parallel processing with associative memory has been the subject of considerable interest and much investigation for over twenty years [1]. However, fully parallel associative memory never developed as a standard component of high density. A number of efforts to develop a compact associative cell were reported [2,3], some resulting in an experimental chip, but most lacked some essential associative function, and none developed commercially. The only chip to reach the market was a 4x4 array (INTEL 3104). On the other hand, a very clever idea was conceived for emulating associative operation with conventional semiconductor memory [4], and was implemented in STARAN, the only associative processor to appear commercially. But emulated memory can access only a bit-slice at a time, operate on it off-chip, then return the resulting bit-slice. This leads to a direct loss in speed, due both to off-chip processing and inability to perform multi-bit compare and write operations. Furthermore, emulation limits the scale of integration to n bits, where n is the required word-length. In 1980 the authors proposed a partitioned associative architecture [5] in which many operands act concurrently, each on its own disjoint data set. This multi-operand approach requires fully parallel associative memory to take advantage of the added concurrency which is virtually lost in emulated associative memory. Hence a renewed effort was made to realize fully parallel associative memory as a high density integrated component. The results, which are summarized below, indicate that a 16K chip with 50 nanoseconds cycle time can be achieved using current MOS technology. The multi-operand approach has been successfully applied to such problems as tomographic back-projection [5,8] which may be recast into disjoint data sets (pixels 'belonging' to a given ray), each acted upon by a different operand (ray attenuation) common to the set. Our main topic here is the application of multi-operand techniques to enhance arithmetic processes such as summed multiplication (extended sum of products). In many problems offering a limited number of disjoint data sets, arithmetic enhancement may be combined with concurrent processing of the data sets.

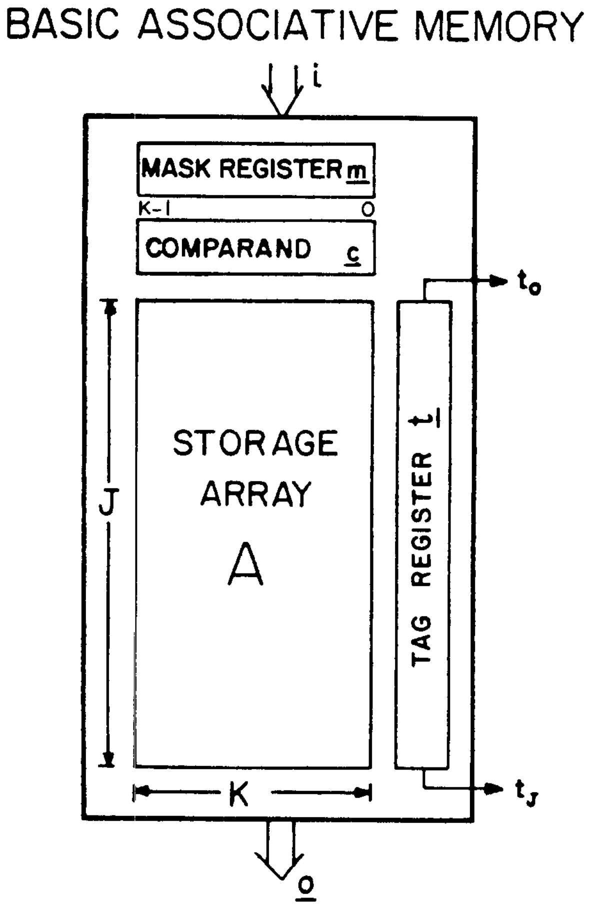


Figure 1

VLSI ASSOCIATIVE MEMORY

An associative memory is a device in which stored data words are identified according to their contents, hence it is sometimes called a content-addressable memory. Such a memory is to be distinguished from the more widely used coordinate addressable memories, such as Random Access Memory (RAM), in which stored data are accessed by their location or address. An associative memory accepts as inputs a comparand (or operand) word and a mask word, searches all stored data locations simultaneously for a match between the unmasked bits of the comparand and the corresponding bits in all the stored words, and identifies matching data words by setting a marker or tag in a TAG register. Our basic model of an associative memory is shown in Figure 1 and its primitive operations given below.

For all $j=0,1,\ldots,J-1$ and all $k=0,1,\ldots,K-1$

1. SETAG $t_j := 1$
2. SHIFTAG $t_j := t_{j-1}$
3. LOAD $\underline{c}$ $c_k := 0,\ c_k := 1,$ or $c_k := i_k$
4. LOAD $\underline{m}$ $m_k := 0,\ m_k := 1,$ or $m_k := i_k$
5. COMPARE $t_j := t_j\ \overline{\sum_k m_k (a_{jk} \bullet c_k)}$
6. WRITE $a_{jk} := \bar{t}_j\, a_{jk} + t_j\, (m_k c_k + \bar{m}_k a_{jk})$
7. READ $o_k := \sum_j t_j\, a_{jk}$

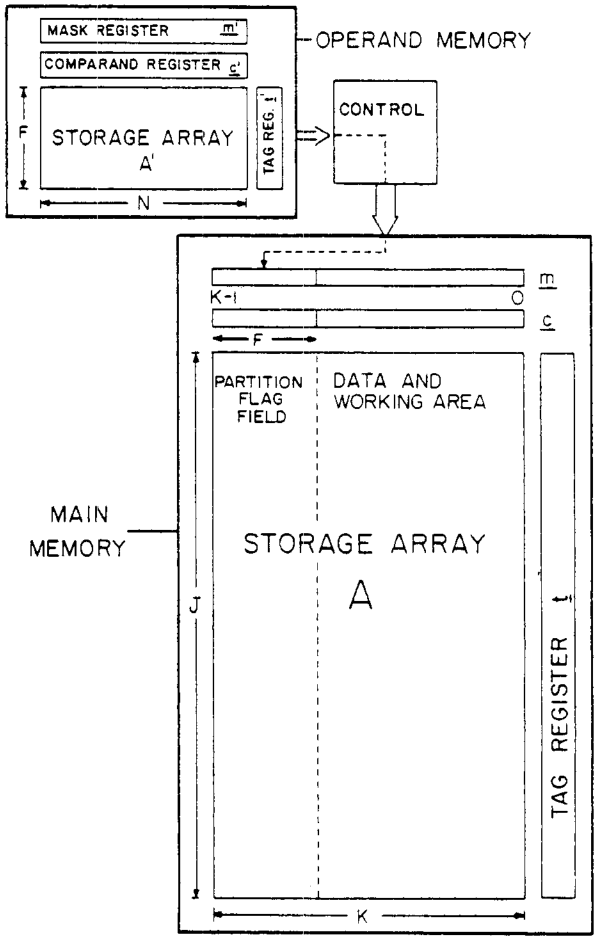
The symbols $+$, $\bullet$ and $\sum$ stand for OR, Exclusive-OR, and OR expansion respectively. $\bar{m}$ denotes the complement of m. Both WRITE and COMPARE operate just on masked ON bits ($m_k=1$) of words (rows) that are tagged ($t_j=1$). Hence

the SETAG operation is required initially to make all words eligible for processing. Up to four operations may be done concurrently during a given memory cycle: SETAG or SHIFTAG, LOAD $\underline{c}$, Load $\underline{m}$, and COMPARE, WRITE or READ. Clearly, any subset of the above may also be concurrent, and when registers $\underline{c}$ and $\underline{m}$ are loaded from the input bus simultaneously, they must receive the same data.

To implement these functions, a 12-transistor, 5-bus, static NMOS associative memory cell was devised, and was laid out in a 6 micron silicon-gate technology, yielding a cell area of 78x132 microns. Allowing for peripheral registers and logic, a 32x32 array will fit on a 4x6 millimeter chip. To gauge the performance of such a chip, a 2x2 cell array with its bus drivers were simulated at circuit level using SPICE2 [6]. The buses were represented by lumped capacitances corresponding to a 32x32 array. The results obtained were a maximum cycle time of 100 nanoseconds and a maximum dissipation of 250 milliwatts when operating continously at the full rate. Scaling down the cell to a line width of 1.5 microns indicates a capacity of 16K bits per chip and a cycle time of 50 nanoseconds at the same power dissipation. It should be noted that only a major operation, COMPARE, WRITE, or READ, requires the full cycle time. A half-cycle will suffice for a minor operation, or any legal combination thereof.

PARTITIONED ASSOCIATIVE ARCHITECTURE

The distinctive associative COMPARE operation is limited to a single comparand. Digby [7] proposed simultaneous bit-by-bit comparison against a number of comparands, F, by providing F flags per word and denoting agreement with any comparand by setting a ONE in the corresponding flag. When examining a single bit-slice of the comparands, they all fall into two groups, those having a ONE in that bit, and those having a ZERO, hence it suffices to deal with each group appropriately, regardless of the number of comparands involved. Referring to Figure 2, main memory A contains the data to be processed and an F-bit flag field. Although the comparands could be stored in a set of serial registers, they are shown residing in an auxiliary associative memory, A'. The flag field is initially set to ONE throughout A, and the data is processed bit serially.
For any $k \in \{0,1,\ldots,N-1\}$ we select the data cells in A having a ONE in bit position k, and write ZERO in all flags corresponding to mismatching comparands - those having a ZERO in this bit position. We do the same for data cells with a ZERO, and comparands having a ONE in bit position k. After repeating this process for each of the bits to be compared, the operation is complete,



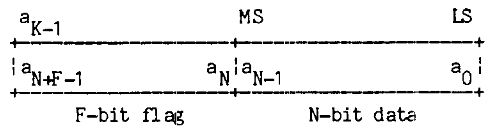
PARTITIONED ASSOCIATIVE PROCESSOR

Figure 2

TABLE 1

Many-to-many Comparison

STEP	MEMORY A	MEMORY A'	CONTROL
0	$\underline{c},\underline{m}:=\underline{d}(N,N+1,..,N+F-1)$; SETAG;WRITE	$\underline{c}':=0;\underline{m}':=\underline{d}(0)$; SETAG;COMPARE	CNT:=0
1	$\underline{c},\underline{m}:=\underline{d}(CNT)$; SETAG;COMPARE		
2	$\underline{m}:=\underline{s}(\underline{t}',K-F,0)$; WRITE	$\underline{c}':=\underline{d}(CNT)$; SETAG;COMPARE	
3	$\underline{c}:=0;\underline{m}:=\underline{d}(CNT)$; SETAG;COMPARE		CNT:= CNT+1
4	$\underline{m}:=\underline{s}(\underline{t}',K-F,0)$; WRITE	$\underline{c}':=0;\underline{m}':=\underline{d}(CNT)$; SETAG;COMPARE	If CNT<N go to 1



Main memory format

and every cell is labeled with the comparand it matches. Cells not matching any comparand will have an all-ZERO flag-field. Clearly, if the comparands are distinct, no cell can be labeled in more than one flag position.

This 'many-to-many' algorithm, in which 'many' comparands are concurrently matched against 'many' data words, is detailed in Table 1. In describing associative algorithms it is convenient to define two binary vector operators which can only appear in assignment statements and assume the length of the vector being assigned. The d operator is akin to Kronecker's delta,

$$\underline{d}(i_1,i_2,..,i_n) \Longrightarrow d_j=1\ \forall j=i_1,..,i_n$$
$$d_j=0\ \forall j\neq i_1,..,i_n$$

where the i's are integral constants, expressions, or symbols. The other is a shift operator,

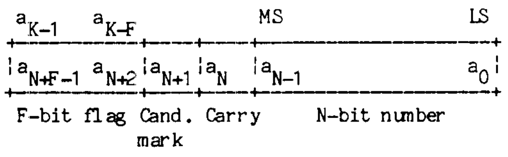
$\underline{s}(\underline{t}',N,B) \Longrightarrow$ vector $\underline{t}'$, initially assumed right justified, is shifted left N places padding with ONEs or ZEROs as indicated by Boolean B, and is truncated or extended by padding on the left to the required length.

The algorithm executes in 4 cycles per bit and offers little advantage in emulated associative memory where writing is bit-serial. For our parallel memory, and a data field-length of N, an appreciable advantage may be gained when F>2N. Unfortunately, efforts to extend the many-to-many idea to other useful algorithms were not successful.

TABLE 2

Multi-Operand Addition

STEP	MEMORY A	MEMORY A'	CONTROL
0	$\underline{c}:=0;\underline{m}:=\underline{d}(N)$; SETAG;WRITE	$\underline{c}',\underline{m}':=\underline{d}(0)$; SETAG;COMPARE	CNT := 0
1	$\underline{c}:=\underline{d}(N)$;SETAG		
2	$\underline{m}:=\underline{d}(CNT,N,N+1)+$ $\underline{s}(\underline{t}',K-F,0)$; COMPARE		
3	$\underline{c}:=\underline{d}(CNT)$;WRITE		
4	$\underline{c}:=\underline{d}(CNT,N)$; SETAG;COMPARE		
5	$\underline{c}:=\underline{d}(N)$; WRITE	$\underline{c}':=0$; SETAG;COMPARE	
6	$\underline{c}:=\underline{d}(CNT)$;SETAG		
7	$\underline{m}:=\underline{d}(CNT,N,N+1)+$ $\underline{s}(\underline{t}',K-F,0)$; COMPARE		
8	$\underline{c}:=\underline{d}(N)$;WRITE		
9	$\underline{c}:=0$; SETAG;COMPARE		
10	$\underline{c}:=\underline{d}(CNT)$;WRITE	$\underline{c}',\underline{m}':=\underline{d}(CNT+1)$; SETAG;COMPARE	CNT:=CNT+1 If CNT<N go to 1



Main memory format

The authors introduced 'multi-operand' algorithms [5] operating in the 'partitioned' architecture shown in Figure 2. Main memory A is dynamically partitioned into F disjoint subsets by an F-bit flag field, thus enabling F operands, stored in memory A', to act concurrently, each on its own distinct subset of data in A. To illustrate this idea, consider multi-operand addition.
Memory A' holds the F addends, and its tag output is routed to the flag-field of the mask register in memory A, permitting the flag-field to be masked with a bit-slice of the addends (or their complement). Each one of the F data sets in memory A is distinguished by setting the flag-bit corresponding to its addend. Elements in memory A not belonging to any one of the F data sets, and not to be involved in the operation, are distinguished by a ONE in the candidate mark. The algorithm is

TABLE 3

Addition Truth Table

IN			OUT		Order of
Addend	Carry	a_{CNT}	Carry	a_{CNT}	Execution
0	0	0	0	0	-
0	0	1	0	1	-
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	4
1	0	1	1	0	3
1	1	0	1	0	-
1	1	1	1	1	-

carried out sequentially by bit and is detailed in Table 2. The truth table for serial addition into an accumulator, shown in Table 3, lists four cases requiring action, and gives a valid order for their execution. Word-format in main memory assigns bit N to store the sequential carry, which is initially set to ZERO (step 0). Steps 1 to 5 operate on data sets belonging to operands whose current bit is ZERO. The current bit-slice in operand memory A' is obtained by executing a COMPARE to ONE (step 0 or step 10), and is applied as a mask to the flag field of main memory A (step 2). A COMPARE against ZERO in A (step 2 and step 4) will then select only cells whose flags correspond to operands with a ZERO in the current bit position. Similarly, steps 6 to 10 operate on data sets belonging to operands whose current bit is ONE. After repeating steps 1 to 10 for each operand bit, the process is complete and each operand has been added to its corresponding data set. It should be noted that this algorithm requires a multi-bit compare to achieve any speed-up, but also takes advantage of a multi-bit write. With the fully parallel associative memory assumed here, execution time is 9 cycles per bit.

Multi-operand subtraction in two's complement notation, with the "difference" replacing the "minuend", can be implemented in a similar manner and executes in the same time. Hence multi-operand multiplication and division follow directly by successive application of addition and subtraction. Consider multi-operand multiply, with main memory partitioned into F disjoint sets of multipliers, and F multiplicands stored in operand memory. The word format in main memory is as shown below,

F	1	1	N	M+N
Part.flag	Cand. mark	Carry	Multiplier	Product

where each N-bit multiplier operates on the M-bit multiplicand belonging to its set, and the full (M+N) bit product is formed in a field adjacent to the multiplier. The algorithm consists of N successive multi-operand additions, one for each bit of the multiplier, starting with the least significant; addition of the appropriate multiplicand is conditional upon the candidate mark and current multiplier bit. The multiplier bit and the starting bit position for addition are incremented at each iteration. Execution time in memory cycles will be $N(9M+2.5)\approx 9MN$, where the added 2.5N cycles are due to one place carry propagation after each addition. At a modest cost in program complexity, word-length in main memory can be reduced by N bits if the multiplier may be overwritten as the product is formed. Multi-operand multiplication can be viewed as a vector-scalar product for a number of vectors and scalars simultaneously. It should be noted that the vectors may be of different length, and their elements may be arranged in any order and intermingled in any fashion.

In computations requiring a single vector-scalar multiplication, this simple successive addition method will run about twice as fast as the multi-vector case, because the additions are now conventional rather than multi-operand. Execution time is now $N(9M+5)/2\approx 9MN/2$. Nevertheless, it is possible to speed up single vector-scalar multiplication significantly through the multi-operand approach by using it to handle several bits of the multiplier at each iteration. Let b denote the number of bits handled at a time, then auxiliary memory is loaded with the binary codes 0 to 2^b-1, alongside the corresponding multiples of the common multiplicand. The number of iterations performed is now N/b, and each iteration starts with a many-to-many comparison to partition main memory into 2^b disjoint sets according to the b-bit multiplier code being handled. This is followed by a multi-operand addition in which each multiple from auxiliary memory is added into the product field of the corresponding set of multipliers. The starting bit position for addition is incremented by b at each iteration. Execution time now becomes,

$$(N/b+9N/2)+9N(M+b)/b = N(9M+1)/b+27N/2\approx 9MN/b+27N/2$$

where the first expression in parenthesis accounts for many-to-many comparison. Multiplication time is plotted in Figure 3 for M=N=60 and b ranging from 1 to 6. The value plotted for b=1 is that for conventional associative multiplication by a constant, $N(9M+5)/2$. The word-length was chosen to be the least common multiple of b=2,3,4,5,6. As was to be expected, multi-operand enhancement of multiplication starts paying off at b=3. The speed-up factor here approaches $b/2=(\log_2 F)/2$, whereas an algorithm directly fitted to the multi-operand model would yield a factor of F/2.

An interesting variation of multi-operand multiplication enhancement arises in extending it to a limited sum of products. Such an expression occurs frequently in coordinate transformation of an image or body representation. A simple example is two-dimensional coordinate rotation, where the new abscissa is given by,

$$x'=x\cos\theta + y\sin\theta$$

The angle of rotation is common to all elements in the image, and so are the trigonometric multipliers applied. Assume main memory contains a description of the image and its word format is as follows:

2^{Tb}	N	N	1	M+N+1
Part. flag	x	y	Carry	x'

As before, we treat x and y as multipliers, cosθ and sinθ as multiplicands. A straightforward approach to enhancement might be to multiply cosθ by field x, then sinθ by field y, in both cases summing into field x' and handling b bits of the multiplier at a time. A better idea, however, is to handle one or more corresponding bits of x and y simultaneously. Assuming two bits of each at a time, the 16 precomputed multiples stored in auxiliary memory A' are:
0,sinθ,2sinθ,3sinθ,cosθ,sinθ+cosθ,...,3sinθ +3cosθ.
The resulting algorithm is simpler and appreciably faster, its execution time being given by,

$$4.5TN+N(9M+9\lceil \log_2 T(2^b-1)\rceil+3.5)/b$$

where T is the number of products summed, and b is now the number of bits per multiplier handled at a time. Appropriate cases are plotted in Figure 3 for comparison against multi-operand enhancement of a single product. Enhancement of an extended sum of products will be treated under convolution below. It should be noted that all our expressions for multiply execution time have assumed the product field to be initially cleared. As a consequence, some improvement in speed may be achieved by executing the first iteration as a copy (rather than add) operation. The speed advantage grows with decreasing multiplier precision N and increasing number of multiplier bits handled at a time (b or Tb).

CONVOLUTION

Multi-operand enhancement of compound associative arithmetic operations will be discussed with reference to convolution, a function much used in signal and image processing. Let $\underline{p}=[p_i]$ be the data vector and $\underline{h}=[h_i]$ the filter vector, both of length P. The discrete convolution of $\underline{p}$ by $\underline{h}$ is defined by:

$$(\underline{h}*\underline{p})_k = \sum_j h_j p_{k-j}$$

where k=0,1,...,2(P-1). Or in matrix form,

$$\begin{bmatrix} h_0 & 0 & \cdots\cdots & 0 \\ h_1 & h_0 & \cdots\cdots & 0 \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ h_{P-1} & \cdots\cdots\cdots & & h_0 \\ 0 & h_{P-1} & \cdots\cdots & h_1 \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ 0 & 0 & \cdots\cdots\cdots & h_{P-1} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \cdot \\ \cdot \\ p_{P-1} \end{bmatrix}$$

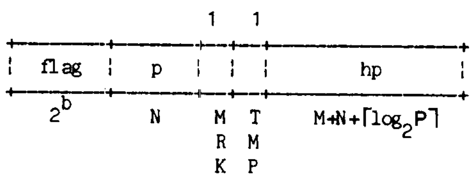
Note that every element of $\underline{p}$ is multiplied by every element of $\underline{h}$. The resulting products are added up to form the convolution vector whose k-th element is the sum of all $h_i p_j$ such that i+j=k. Let us examine the h-matrix:

its non-zero elements occupy the central parallelogram consisting of full diagonals, each diagonal filled with a common h, starting with h_0 in the first diagonal and continuing top to bottom in increasing order of h subscript to the last diagonal which is h_{P-1}. This fact

TABLE 4

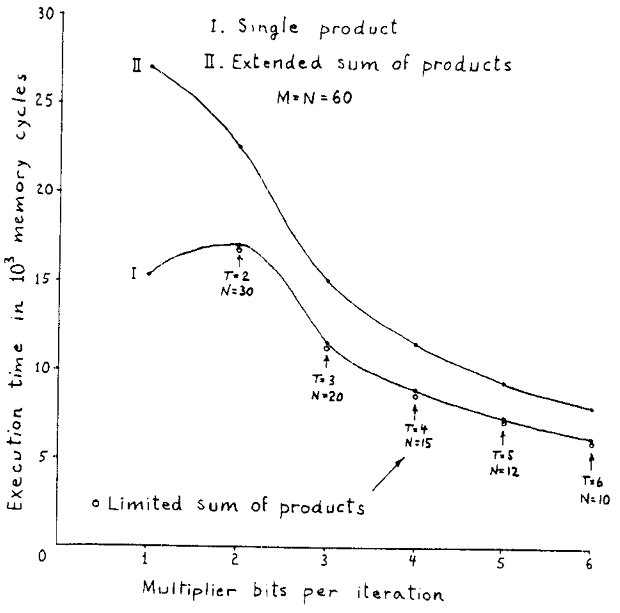
Convolution Algorithm

PHASE	OPERATION
1	Initialize convolution:
	Read in p's and set their markers;
	zero hp field and carry (TMP) column.
	Load b-bit code sequence in memory A'.
	Zero vector element count, EC := 0.
2	Initialize major loop:
	Load multiples of h_{EC} into memory A'.
	Zero bit-group count, BG := 0.
*3	Partition memory A according to current BG of p-field using many-to-many compare.
*4	Perform multi-add from memory A' into hp-field of memory A starting at bit-position b(BG).
*5	Propagate resulting carry through bit-position $M+N+\lceil \log_2 P\rceil$.
6	Increment BG; if b(BG)<N, go to 3.
7	Increment EC; if EC$\geq$P, exit.
8	Shift p-field and marker down. Go to 2.



Main memory format

* Phase operates on 'marked' elements only.



Multi-operand enhanced associative multiplication
Figure 3

TABLE 5

Carry Propagation

STEP	OPERATION
0	BCT := 0
1	$\underline{m}$:= $\underline{d}$(b(BG+1)+N+BCT,TMP);SETAG
2	$\underline{c}$:= $\underline{d}$(TMP);COMPARE
3	$\underline{c}$:= $\underline{d}$(b(BG+1)+N+BCT);WRITE
4	$\underline{c}$:= $\underline{d}$(b(BG+1)+N+BCT,TMP);SETAG;COMPARE
5	$\underline{c}$:= $\underline{d}$(TMP);WRITE
6	BCT := BCT+1; if b(BG+1)+N+BCT<2N+$\lceil \log_2 P \rceil$ go to 1, else exit.

suggests a simple shifting algorithm summarized in Table 4, which includes the word format. The p's are entered in order and their marker bits set; a gap of P-1 words (or greater) is provided below the p vector to develop the last terms of the hp vector. We start by multiplying the elements of $\underline{p}$ by h_0, sum the product into the hp field and shift the data vector down together with the marker. This cycle is repeated for each successive element of h. At the end of this process, the resulting vector will be sitting in proper order in the hp field. Note that the marker was used to limit arithmetic to the elements of $\underline{p}$, regardless of their location in a given cycle. Returning to Table 4, the extended sum of products, hereafter referred to as 'summed multiplication', is described in phases 2 through 6 and is speeded up by handling b multiplier bits at a time. The sub-algorithms of phases 5 and 8 are detailed in Tables 5 and 6 respectively. We now write expressions for the execution time of significant phases of the algorithm, given in memory cyles per vector element.

Phase	
3 :	$N/b + 9N/2$
4 :	$9N(M+b)/b$
5 :	$9N(N + 2\lceil \log_2 P \rceil - b)/4b$
8 :	$5(N+1)$

We pause here to focus on summed multiplication and compare its performance with that of the multiplication algorithms discussed earlier. The execution time of summed multiplication is given by that of phases 3, 4, and 5; adding and simplifying we obtain:

$$N(9M+1)/b+9N(N+2\lceil \log_2 P \rceil+5b)/4b$$

where b>1, N is the precision of p_i, and M the precision of h_i. For direct comparison, summed multiplication time is again plotted in Figure 3 for a 60-bit wordlength and b ranging from 1 to 6. As before, the value plotted for b=1 is that for associative summed multiplication without enhancement:

$$9N(2M+N+2\lceil \log_2 P \rceil+1)/4$$

It is interesting to note that multi-operand enhancement of summed multiplication pays off even at b=2. We deliberately chose one of the simplest compound arithmetic operations to illustrate this point.

Returning to convolution, from the phase execution times given above we obtain the following expression for the total convolution time in memory cycles:

$$P[N(9M+1)/b+9N(N+2\lceil \log_2 P \rceil+5b)/4b+5(N+1)]$$

TABLE 6

Shift Field

FB=first bit of field

STEP	OPERATION
0	BCT := 0
1	$\underline{c}$:= 0,$\underline{m}$:= $\underline{d}$(TMP);SETAG;WRITE
2	$\underline{c}$,$\underline{m}$:= $\underline{d}$(FB+BCT);COMPARE
3	$\underline{c}$,$\underline{m}$:= $\underline{d}$(FB+BCT,TMP);SHIFTAG;WRITE
4	$\underline{c}$:= 0,$\underline{m}$:= $\underline{d}$(TMP);SETAG;COMPARE
5	$\underline{m}$:= $\underline{d}$(FB+BCT);WRITE
6	BCT := BCT+1; if BCT<N go to 1, else exit

This may be approximated by,

$$9NP(N+2\lceil \log_2 P \rceil+5b)/4b + NP(9M+5b)/b$$

Examination of the main algorithm reveals the memory utilization factor during processing to be 50 per cent (neglecting shift). Hence there was no point in looking for a possible partitioning into disjoint data sets that might be processed concurrently, and multi-operand enhancement was applied to the arithmetic instead. If vector and filter have 1024 elements given to 16-bit precision, and the multiplier bits are handled 4 at a time, then convolution will execute in 60 milliseconds. Both the wordlength and execution time can be reduced considerably if the hp field is shortened to 28 bits and computation truncated accordingly, still producing the full attainable precision. In that case, convolution requires 8 chips of main memory per data vector to simultaneously convolve any number of data vectors by a common filter in under 45 milliseconds. For comparison we list some current commercial machines [9,10,11], giving their approximate convolution time for two 1024 element vectors yielding a 2048 element result.

COMPANY	MODEL	CONVOLUTION Time - sec.
Digital Equipment	VAX-11/780	1.5
Floating Point Sys.	AP-120B	0.4
Cray Research	CRAY-1	0.1

The VAX is representative of general purpose sequential machines. The Floating Point AP-120B is an array processor especially designed for problems like convolution. Architecturally, it employs a pipeline to achieve parallelism in vector processing. Even the CRAY-1, a super computer with a pipeline architecture, takes more than double the time to execute convolution. Moreover, an associative convolver can filter many data vectors in parallel at a cost of 8 memory chips per additional 1024 element vector. Thus, for a main memory capacity of 64Kx64 bits (256 chips) the processing rate will be one vector every 1.4 milliseconds.

REFERENCES

1. C. C. Foster , "Content Addressable Parallel Processors" , Van Nostrand Reinhold Co. 1976.
2. J. T. Koo, "Integrated Circuit CAM", IEEE J. of Solid State Circuits, SC-5, 1970, pp. 208-215.
3. J. L. Mundy, J. F. Burgess, R. E. Joynson, "Low Cost Associative Memory", IEEE Journal of Solid-State Circuits, Vol. SC-7, No. 6, 1972, pp. 364-369.
4. K. E. Batcher, "The Multidimensional Access Memory in STARAN", IEEE Transactions on Computers, C-26, 1977, pp. 174-177.
5. I. Scherson, S. Ruhman, "Many-to-many Arithmetic in Associative Memory and its Application to Tomographic Back-projection", Report No. ISR1, Weizmann Institute of Science, March 1980. Also in Proc. 1980 IEEE Int. Conf. on Circuits and Computers, pp. 1492-1495.
6. L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits", Memorandum No. ERL-M520, University of California, Berkeley, 1975.
7. D. W. Digby , "A Search Memory for Many-to-Many Comparisons", IEEE Trans. on Computers, August 1973, pp. 768-772
8. S. Ruhman, I. Scherson, "Associative Processor for Tomographic Image Reconstruction", Proc. 1982 IEEE Comp. Soc. Int. Conf. on Medical Computer Sc./Computational Medicine, pp. 353-358.
9. "VAX Technical Summary", Digital Equipment Corporation, 1982.
10. A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family", IEEE Computer, Vol. 14, No. 9, September 1981, pp. 18-27.
11. R. M. Russel, "The CRAY-1 Computer System", Comm. ACM, Vol. 21, No. 1, January 1978, pp. 63-72.