ODD EVEN SHIFTS IN SIMD HYPERCUBES[1]

Sanjay Ranka[2] and Sartaj Sahni[3]

Abstract

We develop a linear time algorithm to perform all odd (even) length circular shifts of data in an SIMD hypercube. As an application, the algorithm is used to obtain an $O(M^2 + \log N)$ time and O(1) memory per processor algorithm to compute the two dimensional convolution of an $N \times N$ image and an $M \times M$ template on an N^2 processor SIMD hypercube. This improves the previous best complexity of $O(M^2 \log M + \log N)$.

Keywords and Phrases

SIMD hypercube, odd and even shifts, image template matching, one and two dimensional convolution

[1]This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374
[2]Professor Ranka's address is: School of CIS, 4-116 Center for Science & Technology, Syracuse University, Syracuse, NY 13244. [3]Professor Sahni's address is: Computer Science Department, 4-192 EE/CSci Bulding, University of Minnesota, Minneapolis, MN 55455.

1 INTRODUCTION

Data routing is a fundamental operation in parallel computers. Several researchers have developed algorithms for various data routing patterns in hypercubes. For example, the general one-to-one data routing problem may be solved by sorting on the destination tags ([NASS82a]). Efficient algorithms for random access reads and writes are developed in [NASS81] and an optimal algorithm to perform all data routes that fit into the class of bit-permute-complement permutations is developed in [NASS82b]. Broadcasting and personalized communication are considered in [JOHN87b]. Hypercube algorithms for data routing problems that arise in Gaussian elimination are developed in [SAAD89]. Efficient mappings for linear algebra problems are considered in [JOHN87a]. Dekel et al. [DEKE81] develop an optimal hypercube routing algorithm that begins with one data in each processor and circulates the collection of data through every processor such that each processor has only one data at any time. This is applied to the matrix multiplication problem in [DEKE81] and to the image template matching problem in [RANK88].

In this paper we consider the odd/even data shifting problem in SIMD hypercubes. Here we start with data in (say) the *A* register of each hypercube processor. This data is to be shifted clockwise circularly by all odd (even) amounts in the range [1 .. $P-1$] where P is the number of processors in the hypercube. There is no restriction placed on the order in which the odd (even) shifts are performed. So, in the case of odd shifts and $P = 16$ we need to accomplish shifts of 1, 3, 5, 7, 9, 11, 13, and 15. These could be accomplished in the order 11, 3, 7, 1, 5, 15, 13, 9. In Section 3, we develop an O(P) time algorithm for this.

We note that all odd and all even shifts are easily performed in O(P) time on an MIMD hypercube using the gray code mapping scheme ([JOHN87a]). The resulting routing algorithm requires that in one time unit different processors be able to route on different dimensions of the hypercube. Our hypercube model (Section 2.1) does not permit this. In any given time unit all processors that are routing data are required to do this along the same hypercube dimension.

In Section 4, we use our algorithm for even shifting to obtain an improved algorithm to compute two dimensional convolutions on SIMD hypercubes with O(1) memory per processor.

The inputs to this problem are an $N \times N$ image matrix $I[0..N-1, 0..N-1]$ and an $M \times M$ template $T[0..M-1, 0..M-1]$. The output is an $N \times N$ matrix C2D where

$$C2D[i, j] = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} I[(i+u) \bmod N, (j+v) \bmod N] * T[u,v]$$

$C2D$ is called the two dimensional convolution of I and T. Computing $C2D$, is a fundamental operation in computer vision and image processing. It is often used for edge and object detection; filtering; and image registration ([BALL85], [ROSE82]). PrassanaKumar and Krishnan [PRAS87] have developed an SIMD hypercube algorithm for this problem that has time complexity $O(M^2 \log M + \log N)$. This algorithm assumes an O(1) data broadcast capability between the control unit and the hypercube processors and requires N^2 processors and O(1) memory per processor. Using our data shifting algorithm we are able to compute $C2D$ in $O(M^2 + \log N)$ time. Computing C2D on SIMD hypercubes has also been considered in [FANG85] and [FANG86].

2 PRELIMINARIES

2.1 Hypercube Multicomputer

The important features of an SIMD hypercube and the programming notation we use are:

1. There are $P = 2^p$ processing elements connected together via a hypercube interconnection network (Figure 1). Each PE has a unique index in the range $[0, 2^p - 1]$. We shall use parentheses('( )') to index PEs. Thus A(i) refers to the A register of PE i. The local memory in each PE holds data only (i.e., no executable instructions). Hence PEs need to be able to perform only the basic arithmetic operations (i.e., no instruction fetch or decode is needed).

2. There is a separate program memory and control unit. The control unit performs instruction sequencing, fetching, and decoding. In addition, instructions and masks are broadcast by the control unit to the PEs for execution. An *instruction mask* is a boolean function used to select certain PEs to execute an instruction. For example, in the instruction

   $A(i) := A(i) + 1, \quad (i_0 = 1)$

   $(i_0 = 1)$ is a mask that selects only those PEs whose index has bit 0 equal to 1. I.e., odd

indexed PEs increment their A registers by 1. Sometimes, we shall omit the PE indexing of registers. So, the above statement is equivalent to the statement:

$A := A + 1, \quad (i_0 = 1)$

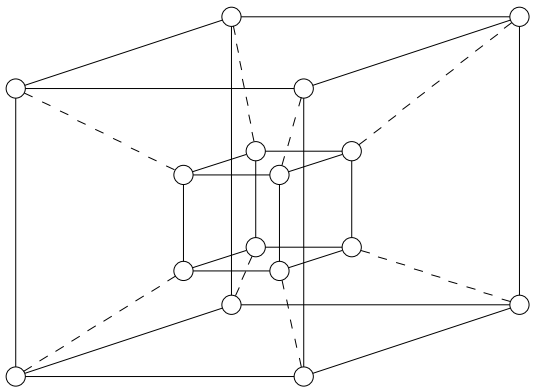

Figure 1: A 4 dimensional hypercube (16 PEs)

3. The topology of a 16 node hypercube interconnection network is shown in Figure 1. A p dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}....i_0$ be the binary representation of the PE index i. Let $\overline{i_k}$ be the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit. I.e., processor $i_{p-1}i_{p-2}...i_0$ is connected to processors $i_{p-1} \cdots \overline{i_k}....i_0$, $0 \le k \le p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

4. Interprocessor assignments are denoted using the symbol $\leftarrow$, while intraprocessor assignments are denoted using the symbol :=. Thus the assignment statement:

   $B(i^{(2)}) \leftarrow B(i), \quad (i_2 = 0)$

   is executed only by the processors with bit 2 equal to 0. These processors transmit their B register data to the corresponding processors with bit 2 equal to 1.

5. In a *unit route*, data may be transmitted from one processor to another if it is directly connected. If the interprocessor connections are unidirectional, then an exchange of data between two hypercube neighbors takes two unit routes. If the connections are bidirec-

tional, this exchange takes a single unit route. Our further discussion assumes unidirectional connections. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only these.

2.2 Arbitrary Circular Shifts

SHIFT(A,i,W) shifts the A register data circularly counter-clockwise by i in windows of size W (W is a power of 2). I.e, $A(qW+j)$ is replaced by $A(qW + (j-i) \bmod W)$, $0 \le q < (P/W)$. *SHIFT*(A, i, W) can be performed in $2\log W$ unit routes [PRAS87]. A minor modification of the algorithm given in [PRAS87] performs $i = 2^m$ shifts in $2\log(W/i)$ unit routes.

The strategy is to reduce a shift in a window of size W into two independent shifts in windows of size $W/2$. By using this repeatedly we end up with shifts in windows of size one. Shifts in windows of this size are equivalent to null shifts. Let the current window size be denoted by $2*M$ (W is the initial window size, initially $M = W/2$). For the reduction, we consider two cases: (a) $0 < i \le M$ and (b) $M < i < 2*M$.

(a) First consider an example. Suppose that $M = 4$ and $i = 3$. Then the initial configuration of line 1 of Figure 2 is to be transformed into that of line 2. The 8 processors may be partitioned into two windows of size 4 each. The left window consists of processors 0 through 3 and processors 4 through 7 make up the right window. Examining the initial and final configurations, we notice that b, c, and d are initially in the left window and they are in the right window after the shift. Also, f, g, and h are initially in the right window and are in the left window following the shift. If we exchange b, c, and d with f, g, and h, respectively, then we obtain the configuration of line 3. Now each of the two windows of size 4 has the data it needs for its final configuration. Furthermore, a shift of 3 in each window will result in the final configuration.

In general, we need to exchange the data in the processors *first* through *last* (*first* = $M - i$, *last* = $M - 1$) of the left window with that of the corresponding processors in the right window. Following this, a shift of i **mod** M is to be made in each size M window.

	PE							
line	0	1	2	3	4	5	6	7
1	a	b	c	d	e	f	g	h
2	f	g	h	a	b	c	d	e
3	a	f	g	h	e	b	c	d
4	c	d	e	f	g	h	a	b
5	e	f	c	d	a	b	g	h

Figure 2 SIMD shift example

(b) Consider the initial configuration of line 1 of Figure 2. This time assume that $i = 6$. The final configuration is shown in line 4. This time a, b, e, and f change windows between the initial and final configurations. When $i > M$, *first* = 0 and *last* = $2M - i - 1$. The data in processors *first* through *last* of the left and right windows need to be exchanged. Following this, the final configuration can be obtained by performing a shift of i **mod** M in each window of size M.

The preceding discussion results in the procedure of Figure 3. Here *RightmostOne* (i) is a function that returns the least significant bit of i that is one. So, *RightmostOne* (12) = 2 and *RightmostOne* (9) = 0. The complexity of *SIMDShift* is O(log$W$). In case the shift amount i is a power of 2, the complexity becomes O(log(M/i)).

3 ODD/EVEN SHIFTS

We explicitly consider only the case of even shifts. There are exactly $(P/2) - 1$ such shifts (recall that $P = 2^p$ is a power of 2). A *shift distance sequence*, E_k, is a sequence $d_1 d_2 \cdots d_{2^{k-1}-1}$ of positive integers such that a clockwise shift of d_1, followed by one of d_2, followed by one of d_3, etc. covers all even length shifts.

```
procedure SIMDShift (A, i, W);
{Counterclockwise shift of A by i in windows of size W.}
{SIMD version.}
   i := i mod W;
   b := RightmostOne (i);
   M := W; k := log2M;
   for j := k–1 downto b do
   begin
      M := M div 2;
      if i <= M then begin
                        first := M – i;
                        last := M – 1;
                     end
                else begin
                        first := 0;
                        last := 2*M – i – 1;
                     end;
      a := p mod M; {p is this processor's index, a is position in size M window}
      A(p^(j)) ← A(p), (first <= a <= last);
      i := i mod M; {remaining shift}
   end; {of for loop}
end; {of SIMDShift}
```

Figure 3 SIMD hypercube shift

Note that $E_0 = E_1 = null$ as there are no even length shifts in the range $[1, 2^m–1]$ when $m = 0$ and 1. $E_2 = 2$. This transforms the length $P = 2^2$ sequence *abcd* into the sequence *cdab*. In general, the choice $E_k = 2, 2, 2, \cdots$ will serve to obtain all even length shifts. From the complexity standpoint this choice is poor as each shift requires $2\log(P/2)$ unit routes. Better performance is obtained by defining

$$E_0 = E_1 = null,\ E_2 = 2$$

$$E_k = InterLeave\ (E_{k-1},\ 2^{k-1}),\ k>2$$

where InterLeave is an operation that inserts a 2^{k-1} in front of E_{k-1}, at the end of E_{k-1}, and between every pair of adjacent distances in E_{k-1}. Thus,

$$E_3 = Interleave\ (E_2,\ 4)$$

$$= 4\ 2\ 4$$

$$E_4 = Interleave\ (E_3,\ 8)$$

$$= 8\ 4\ 8\ 2\ 8\ 4\ 8$$

When a shift sequence E_k is used, the effective shift following d_i is $(\sum_{j=1}^{i} d_j)\ mod\ 2^k$. Thus

when E_3 is used on the sequence *abcdefgh*, we get

d	sequence	effective shift
4	*efghabcd*	4
2	*ghabcdef*	6
4	*cdefghab*	2 = 10 mod 8

Intutively, E_k can be expected to have a low cost as more than half the shifts are of size 2^{k-1}. Each of these has a cost of 2. Further, more than half of the remaining shifts are of size 2^{k-2} and each of these has a cost of 4; etc. So the shift sequence is biased towards the inexpensive shifts.

Theorem 1: Let $E[k, i]$ be d_i in the sequence E_k, $k \geq 2$. Let $ESUM[k, i] = (\sum_{j=1}^{i} E[k, j]) \bmod 2^k$. Then $\{ESUM[k, i] \mid 1 \leq i \leq 2^{k-1}-1\} = \{2, 4, 6, 8, ..., 2^k - 2\}$.

Proof: The proof is by induction on k. The theorem is clearly true for $k = 2$. Let it be true for $k = l \geq 2$. We prove that it will be true for $k = l + 1$. Hence by induction it will be true for all values of k, $k \geq 2$.

We prove the following statements which prove the required result.

(1) $ESUM[l + 1, i] < 2^{l+1}$.

(2) $ESUM[l + 1, i]$ is even.

(3) $ESUM[l + 1, i] \neq 0$

(4) $ESUM[l + 1, i] \neq ESUM[l + 1, k]$ if $i \neq k$

(1) and (2) follow directly from the definitions of $ESUM[l + 1, i]$ and E_{l+1}. We prove (3) and (4) by contradiction. Suppose $ESUM[l + 1, i] = 0$ for some value of i, say $a > 1$ (The case of $a = 1$ is obvious as $E[l + 1, 1] \neq 0$). Then,

$$(\sum_{j=1}^{a} E[l + 1, j]) \bmod 2^{l+1} = 0$$

$$(\equiv)(\sum_{j=1}^{\lfloor a/2 \rfloor} E[l, j] + \lceil a/2 \rceil 2^l) \bmod 2^{l+1} = 0$$

$$(=>)(\sum_{j=1}^{\lfloor a/2 \rfloor} E[l, j]) \bmod 2^l = 0$$

$(\equiv) ESUM[l,\ \lfloor a/2 \rfloor\ ] = 0$

Contradiction (as $\lfloor a/2 \rfloor > 0$ and $ESUM[l,\ i\ ] > 0$ for $i > 0$).

If $k = i \pm 1$ then (4) is true by the definition of *ESUM*. Suppose $ESUM[l + 1,\ i\ ] = ESUM[l + 1,\ k\ ]$, $i \neq k,\ k \neq i \pm 1$. So,

$$(\sum_{j=1}^{i} E[l + 1,\ j\ ])\ mod\ 2^{l+1} = (\sum_{j=1}^{k} E[l + 1,\ j\ ])\ mod\ 2^{l+1}$$

$$(\equiv)(\sum_{j=1}^{\lfloor i/2 \rfloor} E[l,\ j\ ] + \lceil i/2) \rceil\ 2^{l})\ mod\ 2^{l+1} = (\sum_{j=1}^{\lfloor k/2 \rfloor} E[l,\ j\ ] + \lceil k/2 \rceil\ 2^{l})\ mod\ 2^{l+1}$$

$$(=>)(\sum_{j=1}^{\lfloor i/2 \rfloor} E[l,\ j\ ])\ mod\ 2^{l} = (\sum_{j=1}^{\lfloor k/2 \rfloor} E[l,\ j\ ])\ mod\ 2^{l}$$

$$(\equiv) ESUM[l,\ \lfloor i/2 \rfloor\ ] = ESUM[l,\ \lfloor k/2 \rfloor\ ]$$

Contradiction (as $\lfloor i/2 \rfloor \neq \lfloor k/2 \rfloor$). □

Theorem 2: The shift sequence E_k can be done in $2(2^k - k - 1)$ unit routes, $k \geq 2$.

Proof: In Section 2, we presented a shift algorithm that performs a power of 2 shift 2^i in a window of size 2^k using only $2\log(2^k/2^i) = 2(k - i)$ unit routes. Let cost(E_k) be the number of unit routes required by the sequence E_k. The 2^{k-2}, 2^{k-1}'s in E_k take 2 routes each. The cost of the remaining shifts in E_k is cost(E_{k-1}) + $2(2^{k-2}-1)$. The additive term $2(2^{k-2}-1)$ accounts for the fact that each of the remaining $2^{k-2}-1$ routes is done in a window of size 2^k rather than 2^{k-1} (as assumed for E_{k-1}). Hence,

$$cost\,(E_k) = \begin{cases} cost\,(E_{k-1}) + 2(2^{k-2} - 1) + 2*2^{k-2}, & k > 2 \\ 2, & k = 2 \end{cases}$$

So, $cost\,(E_k) = 2(2^k - k - 1)$, $k \geq 2$. □.

The result of the preceding theorem is important as it says that the average cost of rotation in E_k is $\frac{2(2^k - k - 1)}{2^{k-1}-1} < 4$. So, we can perform even length rotations with $O(1)$ average cost.

Let F_k be the sequence obtained by dividing each distance in E_k by 2. So, $F_0 = F_1 = null$, $F_2 = 1$, $F_3 = 2, 1, 2$, etc.

Theorem 3: Let $FSUM[k,\ i\ ] = (\sum_{j=1}^{i} F[k,\ j\ ])\ mod\ 2^{k-1}$ where $F[k,\ j\ ]$ is the j'th distance in F_k

(a) $\{FSUM[k, i] \mid 1 \le i \le 2^{k-1}-1\} = \{1, 2, 3, ..., 2^{k-1}-1\}$

(b) All the shifts in F_k can be done in a window of size 2^{k-1} in $2(2^k - k - 1)$ unit routes.

Proof: Similar to the proof of Theorems 1 and 2.

4 CONVOLUTIONS

4.1 One Dimensional Convolution

The inputs to the one dimensional convolution problem are vectors $I[0..N-1]$ and $T[0..M-1]$. The output is the vector C1D where:

$$C1D[i] = \sum_{v=0}^{M-1} I[(i + v) \bmod N] * T[v] \quad , 0 \le v < N$$

Our algorithm assumes that there are $P = N$ processors and that the vector I is mapped onto the hypercube using the identity mapping (i.e., $I[i]$ on PE i). The N processors may be divided into N/M blocks with each block containing M processors. Further, we assume that there are (N/M) copies of T in the hypercube with one copy in each block of M processors. Within a block, the mapping of T is the same as that of I (i.e., PE i contains $T[i \bmod M]$).

Each PE will compute two quantities A and B. For any PE, A is the sum of all the C1D terms that are in the M block containing the PE. B is the sum of all C1D terms that are needed by the corresponding PE in the previous M block. The terms contributing to A and B are shown in Figure 4. The AB values are computed in two stages. In the first, we compute the contribution to A and B by all I terms I_j for j even. In the next stage, we do this for the case j odd.

Consider the case M =8. If we begin by computing the terms on the major diagonal of Figure 4 , then PEs (0, 1, 2, ,,, 7) compute (I_0T_0, I_2T_1, I_4T_2, I_6T_3, I_0T_4, I_2T_5,I_4T_6, I_6T_7). The I and T values required by each of the 8 PEs are shown in the first two rows of Figure 5. Notice that if we rotate the I values in windows of size 4 by some amount j, then the T values need to be rotated by $2j$ so that each PE has a pair (I, T) whose product is needed in the computating of its A or B value. For this rotation we use the sequences F_3 and E_3. Rotating I by $F[3, 0]$ in size 4 windows and T by $E[3,0]$ in a size 8 window gives the next two rows of Figure 5. The result of per-

P_0	I_0T_0	+	I_1T_1	+	I_2T_2	+	I_3T_3	+	I_4T_4	+	I_5T_5	+	I_6T_6	+ I_7T_7
P_1	I_1T_0	+	I_2T_1	+	I_3T_2	+	I_4T_3	+	I_5T_4	+	I_6T_5	+	I_7T_6	. I_0T_7
P_2	I_2T_0	+	I_3T_1	+	I_4T_2	+	I_5T_3	+	I_6T_4	+	I_7T_5	.	I_0T_6	+ I_1T_7
P_3	I_3T_0	+	I_4T_1	+	I_5T_2	+	I_6T_3	+	I_7T_4	.	I_0T_5	+	I_1T_6	+ I_2T_7
P_4	I_4T_0	+	I_5T_1	+	I_6T_2	+	I_7T_3	.	I_0T_4	+	I_1T_5	+	I_2T_6	+ I_3T_7
P_5	I_5T_0	+	I_6T_1	+	I_7T_2	.	I_0T_3	+	I_1T_4	+	I_2T_5	+	I_3T_6	+ I_4T_7
P_6	I_6T_0	+	I_7T_1	.	I_0T_2	+	I_1T_3	+	I_2T_4	+	I_3T_5	+	I_4T_6	+ I_5T_7
P_7	I_7T_0	.	I_0T_1	+	I_1T_2	+	I_2T_3	+	I_3T_4	+	I_4T_5	+	I_5T_6	+ I_6T_7

Sums above and including the off diagonal are A
Sums below the off diagonal are B

Figure 4: A and B values to be completed by each PE

PE	0	1	2	3	4	5	6	7
I	I_0	I_2	I_4	I_6	I_0	I_2	I_4	I_6
T	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7
I	I_4	I_6	I_0	I_2	I_4	I_6	I_0	I_2
T	T_4	T_5	T_6	T_7	T_0	T_1	T_2	T_3
I	I_6	I_0	I_2	I_4	I_6	I_0	I_2	I_4
T	T_6	T_7	T_0	T_1	T_2	T_3	T_4	T_5
I	I_2	I_4	I_6	I_0	I_2	I_4	I_6	I_0
T	T_2	T_3	T_4	T_5	T_6	T_7	T_0	T_1

Figure 5: Computing the even terms

forming the remaining rotations is also given in Figure 5. Figure 6 gives the computation of the odd terms.

The initial configuration for the I's can be obtained by concentrating the even I's using the strategy described in Figure 7 for the case of M = 16. This requires $\log M$ unit routes. Let CONCENTRATE(I, M) be the algorithm that does this. The algorithm for one dimensional convolution now takes the form given in Figure 8. Note that the E's and F's are known only to the control unit. These may be computed, on the fly, in linear time using a stack of height $m = \log M$. The memory required in each hypercube PE is only $O(1)$. Lines 5 through 15 handle the even terms. Lines 20-27 of the odd terms computation are identical to lines 8-15 of the even terms computation. Notice that $(CShift + 2p)\ mod\ M$ gives the index of the I value currently in $C(p)$. So,

PE	0	1	2	3	4	5	6	7
I	I_1	I_3	I_5	I_7	I_1	I_3	I_5	I_7
T	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_0
I	I_5	I_7	I_1	I_3	I_5	I_7	I_1	I_3
T	T_5	T_6	T_7	T_0	T_1	T_2	T_3	T_4
I	I_7	I_1	I_3	I_5	I_7	I_1	I_3	I_5
T	T_7	T_0	T_1	T_2	T_3	T_4	T_5	T_6
I	I_3	I_5	I_7	I_1	I_3	I_5	I_7	I_1
T	T_3	T_4	T_5	T_6	T_7	T_0	T_1	T_2

Figure 6: Computing the odd terms

Route																
0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
1	I_0	-	-	I_2	I_4	-	-	I_6	I_8	-	-	I_{10}	I_{12}	-	-	I_{14}
2	I_0	I_2	-	-	-	-	I_4	I_6	I_8	I_{10}	-	-	-	-	I_{12}	I_{14}
3	I_0	I_2	I_4	I_6	-	-	-	-	-	-	-	-	I_8	I_{10}	I_{12}	I_{14}
4	I_0	I_2	I_4	I_6	I_8	I_{10}	I_{12}	I_{14}	I_0	I_2	I_4	I_6	I_8	I_{10}	I_{12}	I_{14}

Figure 7: Initial configuration for even terms

if this index is less than *p* the term CD corresponds to the previous block. Otherwise the term CD is for this PE. The fact that each PE always has a C and a D whose product contributes to either A or B follows from the observations that this is so initially and on each iteration, D rotates twice as much as C. The total number of unit routes is $8M + O(\log N) + O(\log M)$.

In the next section we shall make M invocations of one dimensional convolution with the same I values. We describe how these can be done in $12M^2 + O(\log N) + O(M \log M)$ unit routes. First perform a SHIFT(I, -M, P), followed by lines 7, 18, and 19 on the old and new I values and store these results for the later invocations. Now by defining C = $\{I_{old,}\ I_{new}\}$ and modifying steps 11 and 23 so that they calculate terms for this block, we can show that M invocations can be completed in $12M^2 + O(\log N) + O(M \log M)$ unit routes.

```
line   procedure C1D (M)
1      {C1D algorithm}
2      begin
3            A := 0; B := 0; m = log M;
4            {even terms}
5            C := I; D:= T;
6            Cshift := 0;
7            CONCENTRATE (C, M);
8            for j := 1 to M/2 do
9            begin
10           A := A + C * D; ((CShift + 2p) mod M ≥ p)
11           B := B + C * D; ((CShift + 2p) mod M < p)
12           SHIFT(C, F[m, j - 1], M/2);
13           CShift := (CShift + F[m, j - 1]) mod (M/2);
14           SHIFT(D, E[m, j - 1], M);
15           end
16           {odd terms}
17           C := I; D := T;
18           SHIFT(C, -1, M); CShift := 1; SHIFT(D, -1, M);
19           CONCENTRATE(C, M);
20           for j := 1 to M/2 do
21           begin
22           A := A + C * D; ((CShift + 2p) mod M ≥ p)
23           B := B + C * D; ((CShift + 2p) mod M < p)
24           SHIFT(C, F[m, j - 1], M/2);
25           CShift := (CShift + F[m, j - 1]) mod (M/2);
26           SHIFT(D, E[m, j - 1], M);
27           end
28           SHIFT(B, -M, P);
29           C1D := A + B;
30     end; {of C1D}
```

Figure 8: Computing a one dimensional convolution

4.2 Two Dimensional Convolution

Assume that $P = N^2$ PEs are available. These may be viewed as an N × N array (Figure 9). We assume that $I(i, j)$ is initially in the I register of PE(i, j). Further since N and M are assumed to be powers of 2, the N × N array may further be viewed as composed of (N^2/M^2) arrays of size

$M \times M$ (Figure 10). We assume that T is initially in the top left such array.

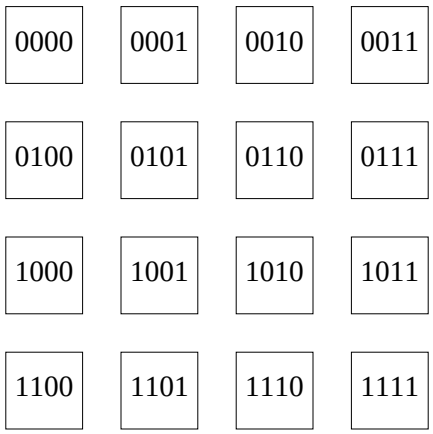

Figure 9: A 16 PE hypercube viewed as a 4×4 array

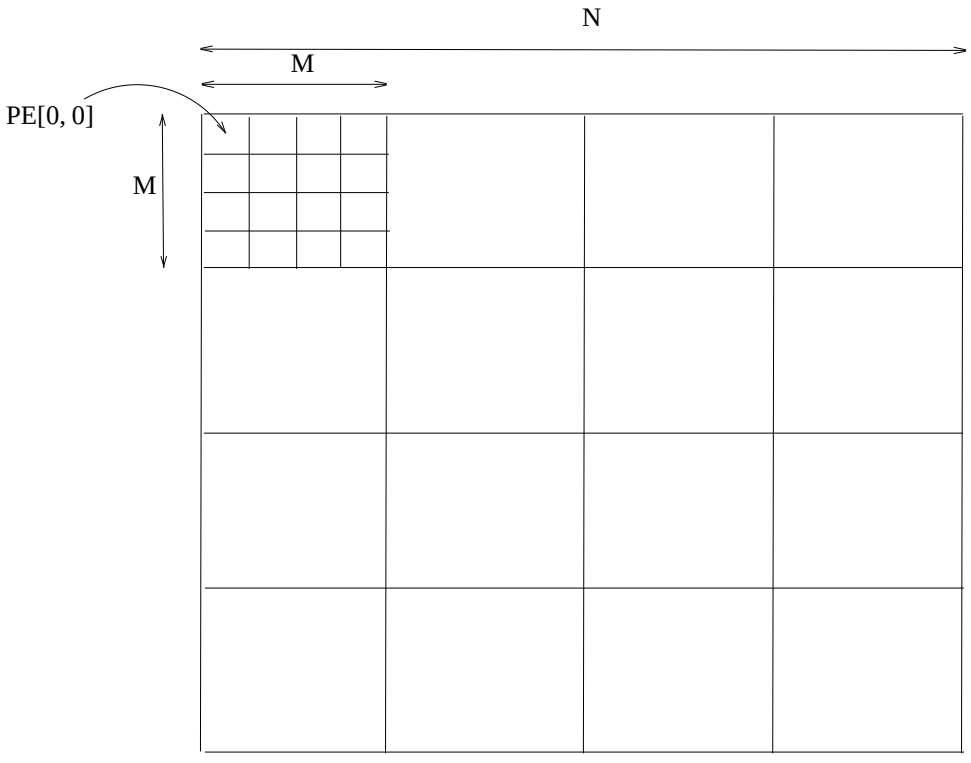

Figure 10: An $N \times N$ array viewed as N^2/M^2 $M \times M$ arrays

The strategy is similar to that used in computing C1D. We may rewrite the definition of

C2D as

$$C2D[i, j] = \sum_{r=0}^{M-1} CXD[i, r, j]$$

where

$$CXD[i, r, j] = \sum_{a=0}^{M-1} I[(i + r) \bmod N, (j + a) \bmod N] * T[r, a]$$

Since each CXD is a one dimensional convolution, it can be computed using algorithm C1D. PE(i, j) computes $E = \sum_{r=0}^{M-i \bmod M-1} CXD[i, r\ j]$ and $F = \sum_{r\ =\ M-i \bmod M}^{M-1} CXD[(i-M) \bmod N, r, j]$. Thus each PE computes a value for itself (i.e., E) and a value for the corresponding PE in the adjacent upper M $\times$ M block (i.e., F). The F values are then shifted M units along the columns and added to the E values to get the C2D values. A high level description of the algorithm is provided in Figure 11. In iteration k of *Step* 3, the PEs in column j of an M $\times$ M PE block compute the CXD terms needed for the E and F of PE$(\lfloor i/M \rfloor M + k, j)$. Then in *Step* 4, these terms are added together to get the E and F for this PE. Figure 12 gives the complexity analysis. The total number of unit routes taken is $12M^2 + O(M \log M) + O(\log N)$. A slightly more efficient algorithm results if we interpret C2D as:

$$C2D[i, j] = \sum_{r=0}^{M-1} X[i, j, r] * Y[r]$$

where $X[i, j, r]$ is the $1 \times$ M vector $I[(i+r) \bmod N, j\ ..\ (j+M-1) \bmod N]$ and $Y[r]$ is the $1 \times$ M vector $T[r, 0\ ..\ M-1]$. Thus C2D is viewed as the one dimensional convolution of X and Y where X and Y are vectors. We can extend algorithm C1D to obtain an algorithm that requires $12M^2 + O(M) + O(\log N)$ unit routes and computes this one dimensional convolution. This algorithm is quite a bit more complex than Figure 11 and is omitted.

5 CONCLUSION

In this paper, we have developed an efficient algorithm to accomplish all odd length as well as all even length circular shifts in an SIMD hypercube. We have applied this algorithm to the image template matching problem and obtained an improved algorithm for the case when the number of processors is N^2 and each processor has O(1) memory.

procedure C2D(N, M)

Step1: Broadcast T to all M × M blocks in the N × N PE array

Step2: Repeat Steps 3 and 4 for k := 0 to M-1

Step3: Compute $CXD[(\left\lfloor \frac{i}{M} \right\rfloor M + k) \bmod N,\ i \bmod M - k,\ j]$ if $i \bmod M \geq k$ using C1D(M) and put the result in A, otherwise A = 0;
Compute $CXD[(\left\lfloor \frac{i}{M} \right\rfloor M + k - M) \bmod N,\ i \bmod M - k + M,\ j]$ if $i \bmod M < k$ using C1D(M) and put the result in B, otherwise B = 0;

Step4: Use the data sum operation, described in Section 2, to sum the results for the adjacent upper block and itself, by summing up B's and A's in $PE(\left\lfloor \frac{i}{M} \right\rfloor M + k,\ j)$ in F and E respectively. Shift the T values along columns by 1, using the algorithm of Section 2.

Step5: Shift(F, -M, N) along columns . E := E + F .

Figure 11: High level description of two dimensional convolution

Step1:	$\log(N^2/M^2)$
Step2:	M(Step3 + Step4) + $O(\log N)$
Step3:	$12M + O(\log M)$
Step4:	$3\log M$
Step5:	$2\log(N/M)$

Figure 12: Unit routes for each step of Figure 11

6 REFERENCES

[BALL85] D. H. Ballard and C. M. Brown, "*Computer Vision*", **1985**, Prentice Hall, New Jersey.

[DEKE81] E. Dekel, D. Nassimi and S. Sahni, " Parallel matrix and graph algorithms", *SIAM Journal on computing*, **1981**, pp. 657-675.

[FANG85] Z. Fang, X. Li and L. M. Ni, "Parallel Algorithms for Image Template Matching on Hypercube SIMD Computers", *IEEE CAPAMI workshop*, **1985**, pp 33-40.

[FANG86] Z. Fang and L. M. Ni, "Parallel Algorithms for 2-D convolution", *International Conference on Parallel Processing*, **1986**, pp 262-269.

[JOHN87a] S. L. Johnsson, "Communication efficient basic linear algebra computations on

hypercube architectures", *Journal of Parallel and Distributed Computing*, 4, 1987, 133-172.

[JOHN87b] S. L. Johnsson and C. Ho, "Optimum broadcasting and personalized communication in hypercubes", Yale University Technical Report, DCS-TR-610, 1987.

[NASS81] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers", *IEEE Transactions on Computers*, C-30, No 2, Feb 1981, pp 101-107.

[NASS82a] D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network", *JACM*, vol 29, no 3, pp 642-667, 1982.

[NASS82b] D. Nassimi and S. Sahni, "Optimal BPC permutations on a cube connected computer", *IEEE Transactions on Computers*, C-31, no 4, pp 338-341, 1982.

[PRAS87] V. K. Prasanna Kumar and V. Krishnan, "Efficient Image Template Matching on SIMD Hypercube Machines", *International Conference on Parallel Processing*, **1987**, pp 765-771.

[RANK88] S. Ranka and S. Sahni, "Image Template Matching on SIMD hypercube multicomputers", *Proceedings 1988 International Conference on Parallel Processing*, Vol III, Algorithms and Applications, pp 84-91, Penn State University Press.

[ROSE82] A. Rosenfeld and A. C. Kak, "*Digital Picture Processing*", Academic Press, **1982**

[SAAD89] Y. Saad and M. Schultz, "Data communication in hypercubes", *JPDC*, 6, 115-135, 1989.