Designing Efficient Matrix Transposition on Various Interconnection Networks Using Tensor Product Formulation*

Chin-Yi Tsai, Min-Hsuan Fan, and Chua-Huang Huang

Department of Information Engineering and Computer Science
Feng Chia University
Taichung, Taiwan, R.O.C.

Abstract

Matrix transposition is a simple, but an important computational problem. It explores many key issues on data locality. In this paper, we will design matrix transposition algorithms on various interconnection networks, including omega, baseline and hypercube networks. Since different interconnection networks have their own architectural characteristics and properties, an algorithm needs to be tuned in order to be efficiently implemented on various networks. We use a tensor product based algebraic theory to design matrix transposition algorithms on various interconnection networks. This method can be employed to design other block recursive algorithms. A major goal of this study is to provide an effective way for designing VLSI circuits of DSP algorithms.

Keywords: tensor product, block recursive algorithm, matrix transposition, interconnection network, hypercube, omega network, baseline network, VLSI circuit design.

1 Introduction

Matrix transposition is a simple, but an important, computational problem. It explores many key issues on data locality. In general, a matrix is a two-dimensional data structure which is stored in a one-dimensional computer memory. No mater whether a matrix is stored in the row-major or column-major order, a simple double-loop transposition program will perform poorly in modern computer architecture with memory hierarchy, including cache memory, external memory, and distributed memory. Efficient matrix transposition algorithms for different memory hierarchy have been broadly studied. Eklundh showed an efficient algorithm which is used to transpose large matrices [4]. Chatterjee and Sen investigated different algorithms for cache efficient matrix transposition [1]. Kaushik *et al.* presented efficient transposition algorithms for large matrices on external memory [15]. Johnsson and Ho reported an transposition algorithm on an n-cube [14]. Choi, Dongarra, and Walk presented a transposition algorithm on distributed-memory multiprocessors [2].

In this paper, we develop matrix transposition algorithms on various interconnection networks, including omega [19], baseline [20] and hypercube networks [14]. Due to different interconnection networks have their own architectural characteristics, an algorithm needs to be specifically designed in order to be efficiently implemented on a given network [8, 11]. The algorithms obtained in this paper can be used to design distributed-memory algorithms and VLSI circuits. In VLSI circuit design, matrix transposition can be implemented using either an easy, but slow, address decoding scheme or a difficult, but fast, data flow scheme with interconnection networks [18]. We will use a tensor product based programming methodology to design matrix transposition algorithms. The methodology is also used to design block recursive algorithms, such as digital signal processing (DSP) algorithms. Hence, it provides an effective method for DSP VLSI chip design.

Tensor product, also known as Kronecker product [7], has been successfully used for designing block recursive algorithms such as fast Fourier transform [12, 13], Strassen's matrix multiplication [9, 10], and parallel prefix algorithm [5, 6]. Tensor product formulas can be directly map to computer programs for parallel, vector, and distributed-memory multiprocessors [3]. Additionally, tensor product formulas are also suitable for specifying various interconnection networks including direct networks and multistage interconnection networks [16, 17]. We will combine tensor product formulation of algorithm representation and interconnection network specification to design matrix transposition algorithms on various interconnection networks.

The organization of this paper is as the following. In Section 2, we give an overview of tensor product oper-

*This work was supported in part by National Science Council, Taiwan, R.O.C. under grant NSC 91-2213-E-035-015.

ations and stride permutation which are the major operations for specifying matrix transposition algorithms. In Section 3, the tensor product formulation to perform matrix transposition is given. Matrix transposition is expressed as a stride permutation and is rewritten to tensor products of other stride permutations. The methodology of designing matrix transposition algorithms on various networks is explained in Section 4. Conclusions and future work are given in Section 5.

2 Tensor Product Notation

We briefly give an overview of the tensor product theory and its relevant properties. Also, a special permutation, stride permutation, is explained. Intuitively, the tensor product operation is a matrix operation that constructs a "large" matrix from two "small" matrices.

Definition 2.1 (Tensor Product) *Let A and B be two matrices of size $m \times n$ and $p \times q$, respectively. The tensor product of A and B is the block matrix obtained by replacing each element $a_{i,j}$ by $a_{i,j}B$, i.e., $A \otimes B$ is an $mp \times nq$ matrix defined as*

$$A \otimes B = \begin{bmatrix} a_{0,0}B_{p\times q} & \cdots & a_{0,n-1}B_{p\times q} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B_{p\times q} & \cdots & a_{m-1,n-1}B_{p\times q} \end{bmatrix}.$$

If the operands are vector bases, say, $e_i^m \otimes e_j^n$, the tensor product is called a *tensor basis*. According to the definition of tensor product, tensor basis $e_i^m \otimes e_j^n$ is equal to e_{in+j}^{mn}.

A tensor product formula can be manipulated using algebraic theorems. Two special forms are often used to express tensor product formulas of computational algorithms. They are called parallel form and vector form. Parallel form and vector form are tensor products with the identity matrix as its first operand and second operand, respectively. Let I_n be the $n \times n$ identity matrix and A be a $p \times q$ matrix. The parallel form and vector form are given as:

$$I_n \otimes A = \begin{bmatrix} A & 0 & \cdots & 0 \\ 0 & A & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A \end{bmatrix},$$

$$A \otimes I_n = \begin{bmatrix} a_{0,0}I_n & a_{0,1}I_n & \cdots & a_{0,q-1}I_n \\ a_{1,0}I_n & a_{1,0}I_n & \cdots & a_{1,q-1}I_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{p-1,0}I_n & a_{p-1,1}I_n & \cdots & a_{p-1,q-1}I_n \end{bmatrix}.$$

Usually, a tensor product in parallel form is implemented as parallel operations of a mutli-processor computer and a tensor product in vector form is implemented as vector operations in a vector architecture. In a computational problem concerning locality issue, parallel form $I_n \otimes A$ can be implemented as the application of A to n segments of successive data elements; vector form $A \otimes I_n$ can be implemented as the application of A to n groups of data elements with fixed stride distance n.

In the tensor product theory, a permutation is often used to change data access pattern. This permutation is called *stride permutation* which rearranges data elements or allows access of data elements with a fixed stride distance. We define stride permutation as below:

Definition 2.2 (Stride Permutation)

$$L_n^{mn}(e_i^m \otimes e_j^n) = e_j^n \otimes e_j^m \ .$$

Actually, a stride permutation exchanges the vector operands of a tensor product. Let X be a vector of length m and Y be a vector of length n. We have $L_n^{mn}(X \otimes Y) = Y \otimes X$. For example,

$$L_2^6\left(\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \otimes \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}\right) = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \otimes \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}.$$

Stride permutation can be expressed as a permutation matrix. We illustrate L_2^6 as below:

$$L_2^6 \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{bmatrix}.$$

We list important properties of tensor product operations and stride permutations below without proof. Note that $\prod_{i=0}^{n-1} F_i$ is denoted as matrix product $F_{n-1}F_{n-2}\cdots F_1F_0$, because matrix product is not commutative.

1. $(\alpha A) \otimes B = A \otimes (\alpha B) = \alpha(A \otimes B)$
2. $1 \otimes A = A \otimes 1 = A$
3. $(A + B) \otimes C = (A \otimes C) + (B \otimes C)$
4. $A \otimes (B \otimes C) = (A \otimes B) \otimes C$
5. $(A_1 \otimes \cdots \otimes A_k)(B_1 \otimes \cdots \otimes B_k)$
$= A_1B_1 \otimes \cdots \otimes A_kB_k$
6. $(A_1 \otimes B_1)(A_2 \otimes B_2) \cdots (A_k \otimes B_k)$
$= (A_1A_2 \cdots A_k) \otimes (B_1B_2 \cdots B_k)$
7. $I_m \otimes I_n = I_{mn}$, $L^n_n = I_n$, $(L^{mn}_n)^{-1} = L^{mn}_m$
8. $\prod_{i=0}^{n-1}(I_n \otimes A_i) = I_n \otimes \prod_{i=0}^{n-1} A_i$
9. $\prod_{i=0}^{n-1}(A_i \otimes I_n) = \prod_{i=0}^{n-1} A_i \otimes I_n$
10. $A_{m\times m} \otimes B_{n\times n} = L^{mn}_m(B_{n\times n} \otimes A_{m\times m})L^{mn}_n$
11. $L^{rst}_{rs} = L^{rst}_r L^{rst}_s$
12. $L^{rst}_t = (L^{rt}_t \otimes I_s)(I_r \otimes L^{st}_t)$

3 Matrix Transposition Algorithms

In this section, we will explain matrix transposition and its correspondence with tensor product operations and stride permutations. In mathematics, matrix transposition is to reshape a matrix as placing the rows of a matrix to the columns of another matrix. In computer science, matrices are stored in a one-dimensional memory. Matrix transposition can be viewed as changing the storage of matrix elements from the row-major order to the column-major order, or vice versa. Let matrix A be of size $m \times n$. If A is stored in a computer memory in the row-major order, the index scheme of element $A_{i,j}$ can be described as tensor basis $e^m_i \otimes e^n_j = e^{mn}_{in+j}$. Similarly, if A is stored in the column-major order, the index scheme of $A_{i,j}$ can be described as tensor basis $e^n_j \otimes e^m_i = e^{mn}_{jm+i}$. As a result, transposition of an $m \times n$ matrix from the row-major order to the column-major order is expressed as L^{mn}_n, for $L^{mn}_n(e^m_i \otimes e^n_j) = e^n_j \otimes e^m_i$.

Various matrix transposition algorithms can be designed by manipulating stride permutation L^{mn}_n. We summarize some algorithms given in [15]. If the the number of rows and the number of columns are composite numbers, say, $m = pq$ and $n = rs$. The block transposition algorithm can be expressed as:

$$\begin{aligned} L^{mn}_n &= L^{pqrs}_{rs} \\ &= (I_r \otimes L^{ps}_s \otimes I_q)(L^{pr}_r \otimes I_{qs}) \\ &\quad (I_{pr} \otimes L^{qs}_s)(I_p \otimes L^{qr}_r \otimes I_s). \end{aligned}$$

This algorithm is divided into four steps. Step 1, $I_p \otimes L^{qr}_r \otimes I_s$, rearranges matrix elements from the row-major order to a block structure order of $p \times r$ blocks with qs elements of each block. Step 2, $I_{pr} \otimes L^{qs}_s$, performs transposition of $q \times s$ matrix for pr blocks. Step 3, $L^{pr}_r \otimes I_{qs}$ transpose a $p \times r$ block matrix with each block of qs elements. Step 4, $I_r \otimes L^{ps}_s \otimes I_q$, converts a block structure order of $r \times p$ blocks with qs elements of each block to the row-major order of the transposed matrix. The result is exactly the column-major order allocation of the input matrix.

Consider a $2^n \times 2^n$ matrix. Eklundh's transposition algorithm [4] is formulated as:

$$L^{2^{2n}}_{2^n} = \prod_{k=0}^{n-1}(I_{2^{n-k-1}} \otimes (I_2 \otimes L^{2^n}_{2^{n-1}})L^{2^{n+1}}_2 \otimes I_{2^k}).$$

Eklundh's algorithm consists of n paths. In each path, there are 2^{n-1} steps and two rows of matrix elements in each step are processed. Appropriate elements of these two rows are exchanged and then the two rows are stored back in a different order.

Another matrix transposition algorithm for a $2^n \times 2^n$ matrix is to factor $L^{2^{2n}}_{2^n}$ into n times of $L^{2^{2n}}_2$:

$$L^{2^{2n}}_{2^n} = \prod_{k=0}^{n-1} L^{2^{2n}}_2.$$

This algorithm is exactly to treat tensor basis $e^{2^n}_i \otimes e^{2^n}_j$ as $e^2_{i_n} \otimes \cdots \otimes e^2_{i_2} \otimes e^2_{i_1} \otimes e^2_{j_n} \otimes \cdots \otimes e^2_{j_2} \otimes e^2_{j_1}$ and to perform right rotation of $e^2_{j_k}$ n times.

In the next section, we will present the design of matrix transposition on various interconnection networks, including omega networks, baseline networks, and hypercube networks.

4 Designing Matrix Transposition Algorithms on Various Networks

We begin with a brief review of representing interconnection networks in tensor product formulas. We consider two kinds of networks: multistage interconnection networks and direct interconnection networks.

The basic component of a multistage interconnection network is a switching element of two input lines and two output lines. A switching element has two states: through state and cross state as Figure 1. A switching element is expressed as a 2×2 matrix: I_2 for the through state and S_2 for the cross state, where

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad S_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

In the case when the state of a switching element is not specified, we use D_2 to denote the switching element. A multistage interconnection network of 2^n input/output lines is a network of n stages with 2^{n-1} switching elements in each stage. Two consecutive stages are connected by a permutation.

The two multistage interconnection networks on which we are going to design matrix transposition algorithms are omega networks and baseline networks. They are expressed as the following tensor product formulas [16]:

$$\begin{array}{l}\text{omega network :}\\ \Omega^N = \prod_{i=0}^{n-1}\left((I_{2^{n-1}} \otimes D_2)L^{2^n}_{2^{n-1}}\right),\\ \text{baseline network :}\\ B^N = \prod_{i=0}^{n-1}\left(I_{2^i} \otimes L_2^{2^{n-i}}\right)(I_{2^{n-1}} \otimes D_2), \text{ and}\end{array}$$

where $N = 2^n$.

A direct interconnection network is a set of processors connected by a set of links. We use the input vector $[0, 1, \cdots, n-1]^T$ to denote the indices of n processors. The links are expressed as permutations of processor indices. Hence, we use a set of tensor product of permutation operations to describe direct interconnection networks. Note that the cross state operation S_2 is equivalent to the complement operation of a binary bit. The hypercube network of N ($N = 2^n$) processors is given as below [17]:

$$\begin{array}{l}\text{hypercube network :}\\ H^N = \left\{I_{2^{n-i-1}} \otimes S_2 \otimes I_{2^i} | 0 \le i < n\right\},\end{array}$$

where $I_{2^{n-i-1}} \otimes S_2 \otimes I_{2^i}$ is the links along dimension i of the hypercube network.

4.1 Matrix Transposition on Omega Networks

We consider transposition of a $2^n \times 2^n$ matrix. Its transposition operation is the stride permutation $L^{2^{2n}}_{2^n}$. To design a matrix transposition on an omega network is equivalent

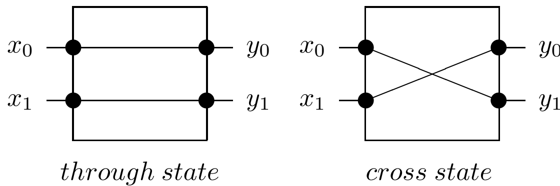

Figure 1: Through state and cross state of a switching element

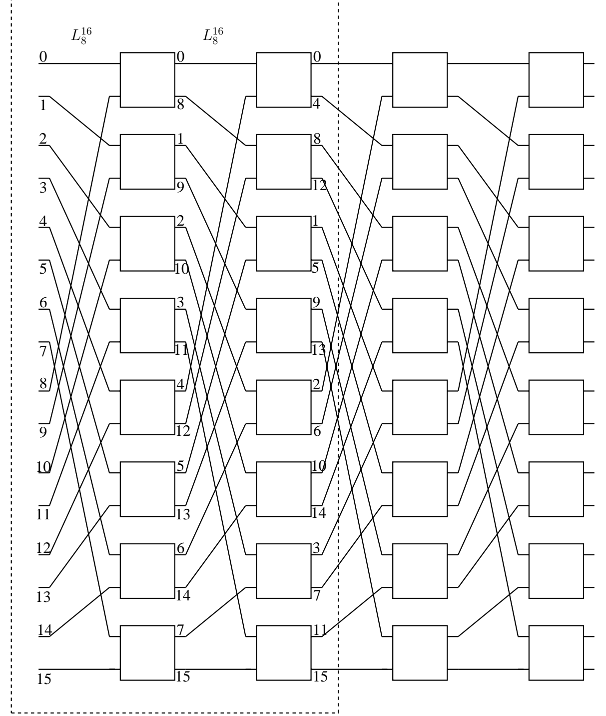

Figure 2: 4×4 matrix transposition on half omega network

to manipulate stride permutation $L^{2^{2n}}_{2^n}$ to fit into the tensor product formula of an omega network. Since there are 2^{2n} matrix elements, we will use an omega network of 2^{2n} input/output lines to implement the transposition algorithm. First, we note that $L^{2^{2n}}_{2^n} = \prod_{i=0}^{n-1} L_2^{2^{2n}}$ and $L^{2^{2n}}_{2^n} = (L^{2^{2n}}_{2^n})^{-1}$. We derive the matrix transposition algorithm as below:

$$\begin{aligned} L^{2^{2n}}_{2^n} &= (L^{2^{2n}}_{2^n})^{-1} \\ &= (\textstyle\prod_{i=0}^{n-1} L_2^{2^{2n}})^{-1} \\ &= \textstyle\prod_{i=n-1}^{0} (L_2^{2^{2n}})^{-1} \\ &= \textstyle\prod_{i=0}^{n-1} (L_2^{2^{2n}})^{-1} \\ &= \textstyle\prod_{i=0}^{n-1} L^{2^{2n}}_{2^{2n-1}} \\ &= \textstyle\prod_{i=0}^{n-1} (I_{2^{2n-1}} \otimes I_2)(L^{2^{2n}}_{2^{2n-1}}). \end{aligned}$$

The above result is exactly the tensor product formula of the omega network with half number of stages and all switching elements being set to the through state. That is, transposition of a $2^n \times 2^n$ matrix can be implemented on a half omega network. If the half omega network is implemented as a VLSI circuit, the switching elements can be simply implemented as delay registers and the shuffle operations are implemented as wire connections. Figure 2 illustrates the half omega network of transposing a 4×4 matrix.

4.2 Matrix Transposition on Baseline Networks

We also consider transposition of a $2^n \times 2^n$ matrix, $L^{2^{2n}}_{2^n}$. To design a matrix transposition on a baseline network is equivalent to manipulate stride permutation $L^{2^{2n}}_{2^n}$ to fit into the tensor product formula of a baseline network. Since there are 2^{2n} matrix elements, we will use a baseline network of 2^{2n} input/output lines to implement the transposition algorithm. Note that the inter-stage permutation of the baseline network of size 2^{2n} is $I_{2^i} \otimes L^{2^{2n-i}}_2$ which is the right rotation of basis e^2 for the $2n-i$ terms on the right-hand-side of the tensor basis $e^2_{i_{n-1}} \otimes \cdots \otimes e^2_{i_1} \otimes e^2_{i_0} \otimes e^2_{j_{n-1}} \otimes \cdots \otimes e^2_{j_1} \otimes e^2_{j_0}$. Let us introduce a bit-reversal operation R^n which reverses the order of the tensor basis $e^2_{i_{n-1}} \otimes \cdots \otimes e^2_{i_1} \otimes e^2_{i_0}$. That is,

$$R^n(e^2_{i_{n-1}} \otimes \cdots \otimes e^2_{i_1} \otimes e^2_{i_0}) = e^2_{i_0} \otimes e^2_{i_1} \otimes \cdots \otimes e^2_{i_{n-1}}.$$

The bit-reversal operation R^n is defined as:

$$R^n = \prod_{i=0}^{n-1} (I_{2^i} \otimes L^{2^{n-i}}_2).$$

To achieve matrix transposition of a $2^n \times 2^n$ matrix, we first reverse n terms on the right-hand-side of the tensor basis, followed by a *partial* bit-reversal of the entire $2n$ terms by performing the first n steps only. We use notation $R^{n,k}$ to denote the first k steps of the n-term bit-reversal operation. The transposition algorithm of a $2^n \times 2^n$ matrix on the baseline network is derived as below:

$$\begin{aligned}
L^{2^{2n}}_{2^n} &= R^{2n,n}(I_{2^n} \otimes R^n) \\
&= [\textstyle\prod_{i=0}^{n-1}(I_{2^i} \otimes L^{2^{2n-i}}_2)] \\
&\quad [I_{2^n} \otimes \textstyle\prod_{i=0}^{n-1}(I_{2^i} \otimes L^{2^{n-i}}_2)] \\
&= [\textstyle\prod_{i=0}^{n-1}(I_{2^i} \otimes L^{2^{2n-i}}_2)] \\
&\quad [\textstyle\prod_{i=0}^{n-1}(I_{2^{n+i}} \otimes L^{2^{n-i}}_2)] \\
&= [\textstyle\prod_{i=0}^{n-1}(I_{2^i} \otimes L^{2^{2n-i}}_2)] \\
&\quad [\textstyle\prod_{i=n}^{2n-1}(I_{2^i} \otimes L^{2^{2n-i}}_2)] \\
&= [\textstyle\prod_{i=0}^{n-1}(I_{2^i} \otimes L^{2^{2n-i}}_2)(I_{2^{2n-1}} \otimes I_2)] \\
&\quad [\textstyle\prod_{i=n}^{2n-1}(I_{2^i} \otimes L^{2^{2n-i}}_2)(I_{2^{2n-1}} \otimes I_2)].
\end{aligned}$$

The resulting formula is a modified baseline network of 2^{2n} input/output lines with all through states. The modification is to perform the last n stages first and then followed by the first n stages. We illustrate the above algorithm for 4×4 matrix transposition in Figure 3.

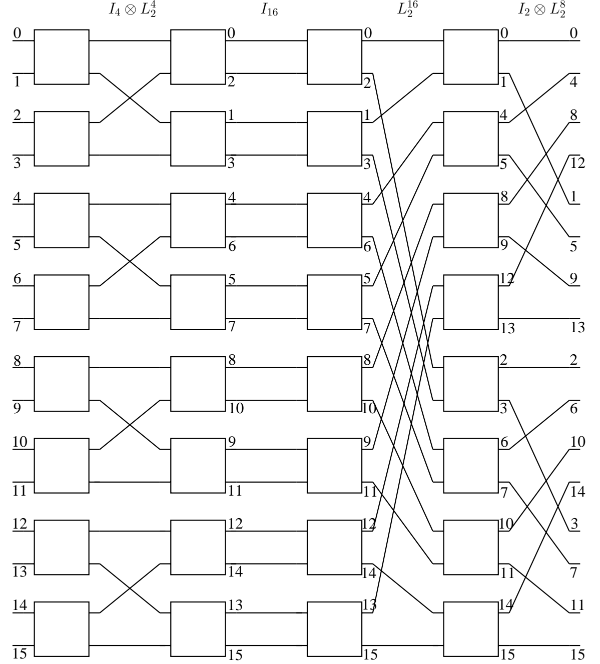

Figure 3: 4×4 matrix transposition on modified baseline network

4.3 Matrix Transposition on Hypercube Networks

In the design of matrix transposition algorithm on hypercube networks, we must implement stride permutations using permutation operations corresponding to the links of hypercube networks. However, since the links of a hypercube is based on bit complement operation S_2, a stride permutation cannot be directly implemented as a product of bit complement operations. That is, a stride permutation cannot be "directly" implemented on hypercube networks.

To solve the problem, we will first implement the basic stride permutation L^4_2 on a two-dimensional hypercube network using an alternative method. The alternative method extends the input data and requires temporary data to achieve stride permutation L^4_2. Let us consider three matrix operations A, B, and C as below:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Let operations P and Q be defined as:

$$P = \begin{bmatrix} A & 0 \\ B & 0 \end{bmatrix}, \quad Q = \begin{bmatrix} A & C \\ 0 & 0 \end{bmatrix}.$$

We rewrite L_2^4 to the formula below:

$$\begin{aligned}
\begin{bmatrix} L_2^4 & 0 \\ 0 & 0 \end{bmatrix} &= QP \\
&= \begin{bmatrix} A & C \\ 0 & 0 \end{bmatrix}\begin{bmatrix} A & 0 \\ B & 0 \end{bmatrix} \\
&= \begin{bmatrix} AA+CB & 0 \\ 0 & 0 \end{bmatrix}.
\end{aligned}$$

Suppose X is a column vector of the four input data elements $[x_0, x_1, x_2, x_3]^T$. The input elements are distributed on four processors indexed from 0 to 3. The augmented operation of L_2^4 requires eight data elements. They include the four input elements in addition to an auxiliary vector T representing four temporary data elements $[t_0, t_1, t_2, t_3]^T$ distributed on four processors. That is, each processor must allocate one input data element and one temporary data element. Hence, the input vector to the augmented operation is $[x_0, x_1, x_2, x_3, t_0, t_1, t_2, t_3]^T$. Let us apply the augmented operation of L_2^4 to the augmented data vector:

$$\begin{bmatrix} L_2^4 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ t_0 \\ t_1 \\ t_2 \\ t_3 \end{bmatrix} = QP\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ t_0 \\ t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

$$= \begin{bmatrix} A & C \\ 0 & 0 \end{bmatrix}\begin{bmatrix} A & 0 \\ B & 0 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ t_0 \\ t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

$$= \begin{bmatrix} A & C \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_0 \\ 0 \\ 0 \\ x_3 \\ x_2 \\ 0 \\ 0 \\ x_1 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

For operation P, since the second column contains all zero matrices, the initial values of auxiliary vector T do not affect the computational result. Thus, the values of $[t_0, t_1, t_2, t_3]^T$ need not be specified. The application of P to the augmented vector preserves input data elements x_0 and x_3, and moves input data elements x_2 and x_1 on processors 2 and 1 to temporary storage t_0 and t_3 on processor 0 and 3, respectively. This step of data passing can be carried out by using links $S_2 \otimes I_2$, i.e., the links along dimension 1 of the 2-D hypercube network. The application of Q to the intermediate result still preserves input data elements x_0 and x_3, and moves x_2 and x_1 in the temporary storage of processors 0 and 3 to the original location of x_1 and x_2 in processors 1 and 2, respectively. This step of data passing can be carried out by using links $I_2 \otimes S_2$, i.e., the links along dimension 0 of the 2-D hypercube network. In fact, implementation of L_2^4 on a 2-D hypercube requires only two steps of data communication and two temporary registers as illustrated in Figure 4.

We further design the transposition algorithm of $2^n \times 2^n$ matrix on hypercube network based on L_2^4. Let the elements of $2^n \times 2^n$ matrix be stored in the row-major order and be distributed to 2^{2n} processors. The tensor basis is given as $e^2_{i_{n-1}} \otimes \cdots \otimes e^2_{i_1} \otimes e^2_{i_0} \otimes e^2_{j_{n-1}} \otimes \cdots \otimes e^2_{j_1} \otimes e^2_{j_0}$. Since L_2^4 exchanges two neighboring terms of the tensor basis, we design the matrix transposition algorithm in the following two steps:

1. Factorize $L_{2^n}^{2^{2n}}$ into tensor products of $L_{2^n}^{2^{n+1}}$ and the identity matrix. This step can be done by applying Property 12 in Section 2 recursively. Note that $L_{2^n}^{2^{n+1}}$ is left-rotation of a tensor basis consisting of $n+1$ terms of e^2.
2. Achieve left-rotation operation $L_{2^n}^{2^{n+1}}$ by exchanging pairs of two neighboring terms from the most left-hand-side toward the right.

Formally, we derive the matrix transposition algorithm as below:

$$\begin{aligned}
&L_{2^n}^{2^{2n}} \\
&= \prod_{i=0}^{n-1} (I_{2^{n-i-1}} \otimes L_{2^n}^{2^{n+1}} \otimes I_{2^i}) \\
&= \prod_{i=0}^{n-1} \left(I_{2^{n-i-1}} \otimes \prod_{j=0}^{n-1} \left(I_{2^j} \otimes L_2^4 \otimes I_{2^{n-j-1}} \right) \otimes I_{2^i} \right).
\end{aligned}$$

The two iterative matrix products of the above transposition algorithm correspond two levels to subcube structure in the $2n$ dimensional hypercube network. For each iteration of the outer iterative matrix product, 2^{n-1} subcubes of 2^{n+1} dimensions are utilized. For each iteration of the inner iterative matrix product, 2^{n-1} 2-D subcubes are utilized to perform L_2^4 permutation. Finally, we show 4×4 matrix transposition on the four dimensional hypercube network. The algorithm is described as:

$$L_4^{16} = (I_2 \otimes L_2^4 \otimes I_2)(L^4 \otimes I_4)(I_4 \otimes L_2^4)(I_2 \otimes L_2^4 \otimes I_2).$$

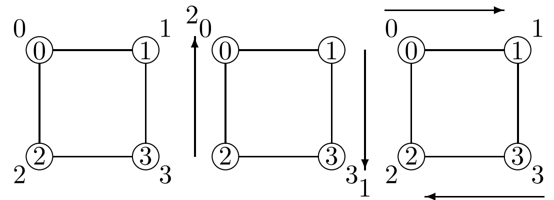

Figure 4: 2×2 matrix transposition on two dimensional hypercube network

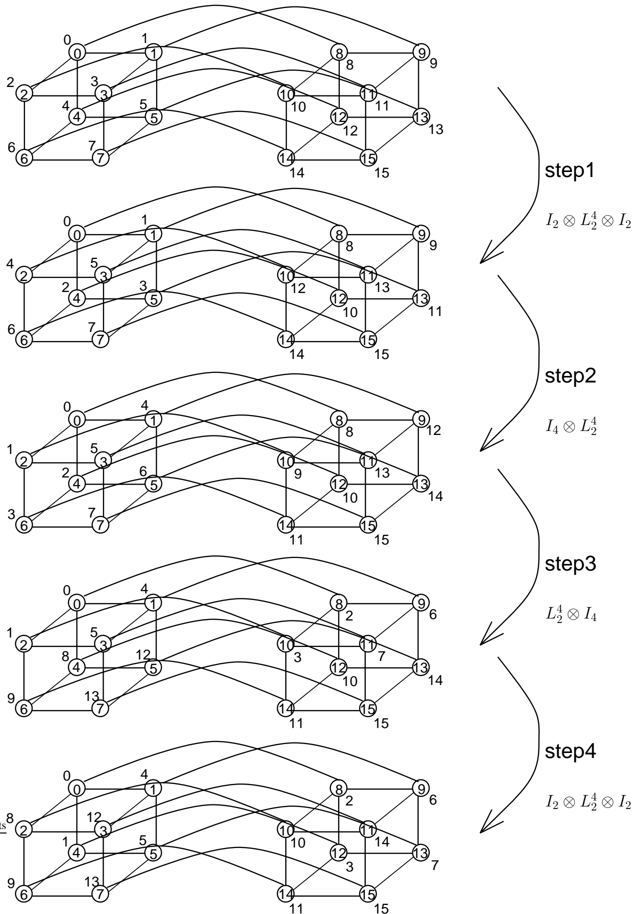

Figure 5: 4×4 matrix transposition on hypercube network

We illustrate the steps of data movement in Figure 5.

5 Conclusions and Future Work

In this paper, we use tensor product as the framework to design matrix transposition algorithms on different interconnection networks. These networks include omega network, baseline network, and hypercube network. In our previous research, the interconnection networks can be represented as matrices. The matrix transposition computation problem is actually a data re-allocation in memory and can be represented as a stride permutation matrix applied on a vector of all data in problem matrix. The key issue is to factorize the stride permutation matrix to a sequence of matrices such that each term can fit into the specified network matrix.

The main concern of designing an algorithm to fit into a specified network is for the ASIC design. The wiring of a ASIC is similar to the links of interconnection network. If we design an algorithm to fit into a specified network, we may also design an similar circuit layout for the computational algorithm. The result of this research is that we may solve a complex computational problem in a simple interconnection network or a fewer wires IC. In the future research, we will try to solve more DSP and image processing problems on different interconnection networks.

References

[1] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 195–205, 2000.

[2] J. Choi, J. J. Dongarra, and D. W. Walker. Parallel matrix transpose algorithms on distributed memory concurrent computers. *Parallel Computing*, 21(9):1387–1405, 1995.

[3] D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan, and R. W. Johnson. EXTENT: A portable programming environment for designing and implementing high-performance block-recursive algorithms. In *Proceedings of Supercomputing '94*, pages 49–58, 1994.

[4] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21:801–803, 1972.

[5] M.-H. Fan, C.-H. Huang, and Y.-C. Chung. A programming methodology for designing parallel prefix algorithms on cube networks. In *Proceedings of ICPP Workshop on Compiler/Runtime Techniques for Parallel Computing*, pages 607–614, 2002.

[6] M.-H. Fan, C.-H. Huang, Y.-C. Chung, J.-S. Liu, and J.-Z. Lee. A programming methodology for designing parallel prefix algorithms. In *Proceedings of the 2001 International Conference on Parallel Processing*, pages 463–470, 2001.

[7] A. Graham. *Kronecker Products and Matrix Calculus: With Applications*. Ellis Horwood Limited, 1981.

[8] J. Hennessy and D. Patterson. *Computer Architecture*. Mogan Kaufmann, 3rd. edition, 2002.

[9] C.-H. Huang, J. R. Johnson, and R. W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm. *Appl. Math Letters*, 3(3):104–108, 1990.

[10] C.-H. Huang, J. R. Johnson, and R. W. Johnson. Generating parallel programs from tensor product

formulas: A case study of Strassen's matrix multiplication algorithm. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume III, Algorithms and Applications, pages III:104–108, 1992.

[11] K. Hwang and Z. Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998.

[12] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying and implementing Fourier transform algorithms on various architectures. *Circuits Systems Signal Process*, 9(4):450–500, 1990.

[13] R. W. Johnson, C.-H. Huang, and J. R. Johnson. Multilinear algebra and parallel programming. *The Journal of Supercomputing*, 5(2–3):189–217, 1991.

[14] S. L. Johnsson and C.-T. Ho. Algorithms for matrix transposition on boolean n-cube configured ensemble architecture. *SIAM Journal on Matrix Analysis and Applications*, 9:419–454, 1988.

[15] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 656–665, 1993.

[16] S. D. Kaushik, S. Sharma, and C.-H. Huang. An algebraic theory for modeling multistage interconnection networks. *Journal of Information Science and Engineering*, 9(1):1–26, 1993.

[17] S. D. Kaushik, S. Sharma, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. An algebraic theory for modeling direct interconnection networks. *Journal of Information Science and Engineering*, 12(1):25–49, 1996.

[18] K. K. Parhi. *VLSI Digital Signal Processing Systems*. John Wiley & Sons, Inc., 1999.

[19] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20:153–161, 1971.

[20] C.-L. Wu and T. Feng. On a class of multistage interconnection networks. *IEEE Transactions on Computers*, 29(8):801–810, 1980.