Matrix Factorizations for Parallel Integer Transformation

Yiyuan She, Pengwei Hao, Member, IEEE, and Yakup Paker, Member, IEEE

Abstract—Integer mapping is critical for lossless source coding and has been used for multicomponent image compression in the new international image compression standard JPEG 2000. In this paper, starting from block factorizations for any nonsingular transform matrix, we introduce two types of parallel elementary reversible matrix (PERM) factorizations which are helpful for the parallelization of perfectly reversible integer transforms. With improved degree of parallelism and parallel performance, the cost of multiplications and additions can be, respectively, reduced to $O(\log N)$ and $O(\log^2 N)$ for an N by N transform matrix. These make PERM factorizations an effective means of developing parallel integer transforms for large matrices. We also present a scheme to block the matrix and allocate the load of processors for efficient transformation.

Index Terms—Integer-to-integer transforms, lossless compression, matrix factorization, parallel algorithms, parallel architectures.

I. INTRODUCTION

UE to the limitation of computational precision and storage capacity, it is preferable for transforms used in data compression to be integer reversible. Integer transform (or integer mapping) is a type of transform that maps integers to integers and realizes perfect reconstruction. This area has been explored for some time. The early work has concentrated on some simple integer-reversible transforms, such as S transform [1], TS transform [2], and S+P transform [3]. This has suggested a promising future for reversible integer mapping in image compression, region-of-interest coding, and unified lossy/lossless compression systems. However, not until the lifting scheme [4] was proposed for constructing second-generation wavelets did people try to break away from various specific transforms and specific rounding methods to build generic integer wavelet transforms [6] based on the simplified ladder structure [5]. From then on, research in this area has grown fast and the technique is widely adopted in a variety of applications.

Y. She was with the Center for Information Science, Peking University, Beijing 100871, China. He is now with the Department of Statistics, Stanford University, Stanford, CA 94305 USA (e-mail: yyshe@stanford.edu).

P. Hao is with the Department of Computer Science, Queen Mary, University of London, London E1 4NS, U.K., and with the Center for Information Science, Peking University, Beijing 100871, China (e-mail: phao@dcs.qmul.ac.uk; phao@cis.pku.edu.cn).

Y. Paker is with the Department of Computer Science, Queen Mary, University of London, London E1 4NS, U.K. (e-mail: paker@dcs.qmul.ac.uk).

Digital Object Identifier 10.1109/TSP.2006.881227

For a finite dimensional signal, the transform matrix can be simplified from a polyphase matrix consisting of Laurent polynomials [7] to a constant matrix of finite dimension. By matrix factorization, Hao and Shi [8] first considered the reversible integer transform implementations for such invertible linear transforms in a finite dimensional space, and later obtained an optimal factorization with minimum number of matrices [9]. This technique has been included in the new international image compression standard, JPEG 2000 [10].

Plonka [15] recently proposed a simple method of integer transform with small errors by scaling the matrix by a global constant, but the dynamic range may have been expanded and the choice of this constant strongly relies on the specific rounding operator-it may not even exist for some rounding operators makes this method in our algorithms [8]-[12]. Moreover, this method needs postprocessing to recover the true transformed values, which are hardly integers. Accordingly, even after this "integer transform" is done, no simple software/hardware only dealing with integers is adequate in practice, which implies that the coding advantage and the speedup of true integer transform may get lost, in contrast to the general procedure [1]–[6] we focus on. (Of course, this procedure may not be a problem if the relative transformed quantities matter only.) For the problem we have solved and the methods we propose, there is no need for preprocessing or postprocessing, the dynamic range of the transform is compact, the rounding arithmetic can be flexibly chosen, and the transform is truly reversible for integers.

However, the computational efficiency of the inverse integer transform based on the matrix factorization technique in [8]–[10] remains a problem, especially for large matrices, due to the recursiveness of the reconstruction. To overcome this drawback, in this paper we introduce two new matrix factorizations, called parallel elementary reversible matrix (PERM) factorizations, which are more suitable for improving computational efficiency and for parallel algorithm design. In fact, even for sequential computation these two factorizations offer advantages over former matrix factorizations—referred to as *point* factorizations below, to differentiate from *block* matrix factorizations as this paper is based on.

Section II introduces the necessary background and reviews the point factorization and block factorization techniques. In Section III, based on the block triangular ERM (TERM) and single-row ERM (SERM) factorizations [11], [12], we introduce two types of PERM factorizations for the parallel integer transform. Section IV is a discussion of computational complexity. We present an efficient matrix blocking scheme and multiprocessor arrangement in Section V. Some examples are demonstrated in Section VI. Conclusions are given in Section VII.

Manuscript received November 26, 2005; accepted December 24, 2005. This work was supported in part by FANEDD of China under Grant 200038 and NKBRPC of China under Grant 2004CB318005. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. David J. Miller.



Fig. 1. A flowchart structure of the linear transform implemented by SERMs.

II. POINT AND BLOCK FACTORIZATIONS

The basic matrix factors for reversible integer transformation are called elementary reversible matrices (ERMs), including triangular ERMs (TERMs, lower or upper) and single-row ERMs (SERMs). A TERM is defined as a special triangular matrix whose diagonal elements belong to the unit group of an integral domain. For instance, on the set $\{a+bi|a, b \in Z\}$, these should be ± 1 and $\pm i$, the so-called integer factors in [9]. A SERM is a matrix with integer factors on the diagonal and only one nonzero row apart from the diagonal. Obviously, a SERM can be converted to a simple TERM by a row and a column permutation. Furthermore, a unit TERM is a unit triangular matrix and a unit SERM associated with the *i*th row can be formulated as $S_i = I + e_i s_i^T$, where e_i is an elementary vector with the *i*th element one and all others zero, and s_i is a vector whose *i*th element is zero.

The reversible integer mapping can be implemented via a series of TERMs, or a series of SERMs, equivalently. Let $A = (a_{i,j})$ be a lower TERM of size N with a diagonal of integer factors j_1, \ldots, j_N . Then the forward integer transform for y = Ax is computed downwards as follows:

$$\begin{cases} y_1 = j_1 x_1 \\ y_m = j_m x_m + \left[\sum_{n=1}^{m-1} a_{mn} x_n \right] & 2 \le m \le N \end{cases}$$
(1)

The inverse transform is computed in a recursive way, like the forward elimination

$$x_{1} = \frac{y_{1}}{j_{1}}$$

$$x_{m} = \frac{1}{j_{m}} \left(y_{m} - \left\lfloor \sum_{n=1}^{m-1} a_{mn} x_{n} \right\rfloor \right),$$

$$m = 2, \dots, N$$
(2)

where $\lfloor \rfloor$ is *any* given rounding arithmetic, which can be rounding off at bits before or after the decimal point. The computation is similar to that for an upper TERM, except that the computational ordering of the inverse is upward. It is easy to see the following characteristics of the above transform

computations: i) mapping integers to integers; ii) perfect reconstruction; and iii) in-place computation. All these are attractive for lossless data compression.

Given an $N \times N$ nonsingular matrix A, there are two important factorizations in [9].

- 1) If the leading principal minors of A are all ones, then $A = LU = S_N \cdots S_1$, named as SERM⁽⁰⁾.
- 2) If det(A) is an integer factor, then $P^T A = S_N \cdots S_1 S_0$, named as SERM⁽¹⁾, where P is a permutation matrix, S_0, S_1, \ldots, S_N are unit SERMs, and S_0 is associated with the last row (also a lower TERM).

If det(A) is nonzero, then, after a scaling modification and a few permutations, the integer transform of A of size $N \times N$ can be implemented by no more than N+1 SERMs, as illustrated in Fig. 1 for a SERM⁽¹⁾ integer transform. The number of scalar floating-point multiply-add operations is, respectively, $N^2 - N$ and N^2-1 for SERM⁽⁰⁾ and SERM⁽¹⁾ integer transforms.

Note that the algorithm of the inverse TERM integer transform is recursive, although it avoids the computational workload of evaluating the inverse matrix. For instance, an inverse upper triangular transform has to be executed stepwise from bottom to top, which makes it unsuitable for parallel computation. As for the integer transform in the form of SERM factorizations, actually only one component is changed through a unit SERM as the computation remains sequential. Ideally, we would like to have all signal components reconstructed in parallel. It is worth mentioning that in this paper, we focus on the cost of integer transformation but not that of the factorization which is assumed to be known by both encoder and decoder ends.

Observing that a unit SERM can be trivially generalized to a unit *block* SERM (for simplicity, we still use S_i): $S_i = I + e_i s_i^T$, where e_i is an elementary block matrix of which the *i*th block is I and s_i is a block matrix with the *i*th block zero, we studied block factorizations in [11] and [12]. In contrast to point SERM factorizations, block SERM factorizations make it possible that the factorizations and the transforms are carried out at block level and therefore boost the degree of parallelism. Such block approaches are more appropriate for an efficient integer implementation of large matrices, as well as those with natural block structures originating from the underlying physical backgrounds.



Fig. 2. PERM⁽⁰⁾ factorization (suppose $n^{(1)} = n^{(2)} = n^{(3)} = 2$).

For example, given a 2-by-2 block unit lower SERM $A = \begin{bmatrix} I & 0 \\ M & I \end{bmatrix}$, to reconstruct $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$ from $\mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix}$, the integer transform of $A\mathbf{x}$, we can use the block formula below instead of the one-by-one reconstruction of (2)

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 - \lfloor \boldsymbol{M} \boldsymbol{x}_1 \rfloor \end{bmatrix}$$
(3)

where | | is a rounding operator for all elements in the vector.

Generalizing point factorizations to block factorizations is not so straightforward due to the difficulty of the scaling modification and the possibility that some crucial blocks may not have full rank in factorization. In [12], in an almost arbitrary partition manner, we defined a generalized determinant matrix function "DET" (see Appendix A) and studied the block LU (BLU) factorization A = PLDU, where P is a permutation matrix, L, U are unit lower and unit upper block triangular matrices, respectively, and **D** is a block diagonal matrix. We also discussed how to convert them into the block unit SERM factorizations in [12]. In the case that all blocks are of the same size [11], we redefined the generalized determinant matrix function DET and obtained a **BLUS** factorization $A = PLDUS_0$, where S_0 is a unit block SERM associated with the last block row (thus S_0 is also a unit lower TERM) and $D = \text{diag}(I, I, \dots, I, \text{DET}(P^T A))$. We proposed a practical algorithm [11] as a generalization of point TERM factorization [9] and also proved that block SERM factorization $A = PS_n \cdots S_1 S_0$ exists if and only if $DET(P^T A)$ is a diagonal matrix and all the diagonal elements are integer factors.

In the following discussions, we assume uniform blocking, and mainly use basic block SERM forms of **BLU** and **BLUS** factorizations— $PDS_n \cdots S_1$ and $PD'S_n \cdots S_1S_0$, where Pis a permutation matrix at the element level, D is a block diagonal matrix, and D' is a block diagonal matrix with only one diagonal block not to be I (in this paper, it is supposed to be the bottom-right block¹). Throughout the rest of this paper, A is the original transform matrix in a finite dimensional space, n the number of blocks in a row or column, m the size of each block, and A_n the corresponding block matrix of A.

III. PARALLEL ERM (PERM) FACTORIZATIONS

A linear transform of an $N \times N$ block SERM, $S_i = I + e_i s_i^T$, with the *i*th block of s_i being zeros and of block size $m \times m$, can be implemented by parallel multiplications and parallel

additions. The main difficulty of applying block factorizations to parallel computing lies in the residue D (or D'). Note that row and column permutations alone are not capable of converting $\text{DET}(A_n)$ into I. We exploit recursive factorizations.

For a matrix of size N_1 , at the *k*th level, we partition the residue from the last level into $n^{(k)}$ blocks of size $m^{(k)}$ until the block size reduces to N_2 . This process is denoted as

$$N_1 = m^{(0)} \xrightarrow{n^{(1)}} m^{(1)} \xrightarrow{n^{(2)}} \cdots \xrightarrow{n^{(K)}} m^{(K)} = N_2.$$
 (4)

Take **BLU** factorization as an example. At the *k*th level, each diagonal block of $D^{(k-1)}$ is further partitioned into $n^{(k)} \times n^{(k)}$ blocks of block size $m^{(k)} \times m^{(k)}$. Then we apply BLU and block SERM factorization to factorize $D^{(k-1)}$ into $n^{(k)}$ block SERMs, formally denoted as $S_j^{(k)}$ ($1 \le j \le n^{(k)}$), and a non-ERM block diagonal matrix $D^{(k)}$. We repeat this process recursively until all the blocks are reduced to single elements (see Fig. 2 for an illustration), and finally we obtain

$$A = P^{(1)} P^{(2)} \cdots P^{(K)} D^{(K)} (L^{(K)} U^{(K)}) \cdots (L^{(1)} U^{(1)})$$

= $PD \prod_{k=K}^{1} S_{n^{(k)}}^{(k)} \cdots S_{1}^{(k)}$ (5)

where $D = \text{diag}(d_1, \ldots, d_N)$ and K is the number of factorization levels. It is not difficult to see that $\prod_{j=1}^{i} d_j$ is the *i*th leading principal minor of $P^T A$.

Similarly, successively applying **BLUS** to factorize the last diagonal block of previously remained nonidentity submatrix as shown in Fig. 3 yields a factorization

$$\boldsymbol{A} = \boldsymbol{P} \boldsymbol{D} \prod_{k=K}^{1} \boldsymbol{S}_{n^{(k)}}^{(k)} \cdots \boldsymbol{S}_{1}^{(k)} \boldsymbol{S}_{0}^{(k)}$$
(6)

where $\boldsymbol{D} = \operatorname{diag}(1, \ldots, 1, \operatorname{det}(\boldsymbol{P}^{\boldsymbol{T}}\boldsymbol{A})).$

To realize the perfect integer-reversible transform, we need to make a scaling modification to the original transform matrix before the factorization, as suggested in [9]. For the factorization formula (5), we can left-multiply A by $PD^{-1}P^{T}$, where the leftmost P is to maintain the order. Since the scaling values in each dimension are perhaps only meaningful in mathematics, (5) may be of limited use in real-world applications, although it has fewer factor matrices than (6). By contrast, in (6), we are free to choose any rows or columns for scaling, as long as the final determinant is an integer factor. This less restrictive scaling plays an important role in keeping proportions of the transform matrix and adjusting the dynamic ranges of data (see [9, Section VIII]), and so provides more flexibility and practical utility. Of

¹It is worth mentioning that if we fix the structure of D in this way, there exists a far general type of block factorization in an almost arbitrary partitioning manner [13], [14].



Fig. 5. An example of the flowchart structure of the (forward) PERM⁽¹⁾ integer transform.

course, **BLU** and **BLUS** can be combined in factorization. We can draw similar conclusions from right-permutation block factorizations.

Hereafter, the scaled formulas (5) and (6), appropriate for perfectly reversible integer transforms, are referred to as parallel ERM factorizations and are denoted by PERM⁽⁰⁾ and PERM⁽¹⁾, respectively, as a counterpart of SERM⁽⁰⁾ and SERM⁽¹⁾. From the scaling process we easily see that it is sufficient to investigate unit PERM and unit SERM factorizations. Figs. 4 and 5 are illustrations of their corresponding integer transforms. Fig. 4 is a flowchart of PERM⁽⁰⁾, where $N_1 = N = 16$, $N_2 = 1$, K = 2, $n^{(1)} = n^{(2)} = 4$. Note that there are altogether eight steps in the implementation, as compared to 16 in SERM⁽⁰⁾. The inverse transform is obtained by just moving in the reverse direction. Fig. 5 is a flowchart of PERM⁽¹⁾, where $N_1 = N = 16$, $N_2 = 1$, K = 2, $n^{(1)} = n^{(2)} = 4$. There are altogether ten transform matrices in the implementation, as compared to 17 in SERM⁽¹⁾. Similarly, the inverse transform is obtained by moving in the reverse direction.

IV. PARALLEL COMPUTATIONAL COMPLEXITY

In order to estimate the computational complexity, for the following analysis, we do not use any characteristics of an underlying parallel architecture but consider as the general case where the computation is performed by processors capable of multiplying or adding two numbers, ignoring the communication and storage aspects. Clearly, for the specific architecture, further, more refined analysis needs to be performed for more targeted and precise complexity estimates, which would involve the implications of the internal data communications and storage. The following estimates consider the transform of a single vector. When a stream of vectors is to be transformed, a particular architecture using such a pipeline affects the throughput. However, this is beyond the scope of this paper. We concentrate on how to structure the problem and study the implication for parallel implementations without a priori knowledge of the eventual architecture, which could be a general-purpose parallel computer or a hard-wired special-purpose machine, i.e., on silicon. The total number of multiplications and additions gives us a measure of complexity.

For PERM⁽⁰⁾, if $N_1 = N$, $N_2 = 1$, the number of multiplications and additions, equal to those of SERM⁽⁰⁾, is given by

$$\sum_{k} \left(\frac{m^{(k-1)}}{n^{(k)}} \cdot \frac{N}{m^{(k-1)}} \right) \cdot \left(m^{(k-1)} \left(1 - \frac{1}{n^{(k)}} \right) \right) \cdot n^{(k)} = N^2 - N.$$
(7)

For $PERM^{(1)}$, this number is also the same as $SERM^{(1)}$ and is given by

$$\sum_{k} m^{(k)} (m^{(k-1)} - m^{(k)})(n^{(k)} + 1) = N^2 - 1.$$
 (8)

Thus the computational complexity of a PERM is the same as the equivalent SERM. However, since the computation can be now organized onto blocks [see (3) for an example], the performance can be improved by using some mathematical packages, such as BLAS. Moreover, with nontrivial elements to be $m(N-m) \ge N-1$, degree of parallelism increases and more processors (up to $N^2/4$) can be involved in computing. We notice that the additional freedom of row partitioning in the two-dimensional data structure helps cut down the computation cost for parallel computing, owing to the independent reconstruction of all the intrablock rows in the inverse PERM integer transform.

Fig. 6 is an example of the transformation of a 16×16 block matrix S_1 with block size 4×4 , corresponding to a stage in Fig. 4 or 5. In Fig. 6, only the computation of the first element is explicitly depicted. In a block SERM transformation, all the multiplications at a given stage can be executed in parallel by using p processors, so that the total computational time of multiplications in parallel is [m(N-m)/p] if we take the multiplication of two numbers as a unit time. However, additions are not so simple. For each processor adding two numbers, p processors can only implement cumulative addition (summation) of n numbers in $\lceil \log_2 n \rceil$ addition steps if n < 2p. For n > 2p, the computational time of additions is $[(n-p)/p + C\log_2 p]$, where 0 < C < 1. Therefore, the computational time of parallel additions for a block SERM transform is $\lceil \log_2(N-m) \rceil$ if m(N-m)m < 2p and $[(m(N-m) - p)/p + C(\log_2 p - \log_2 m)],$ or simply $[m(N-m)/p + C(\log_2 p - \log_2 m)]$, if $m(N - m)/p + C(\log_2 p - \log_2 m)$ $m) \geq 2p.$



Fig. 6. Parallel computation of a reversible integer transform with a block SERM.

Theoretically, the multiplication time of the parallel integer transform (4) with $\text{PERM}^{(0)}$ is

$$T_{\text{PERM}^{(0)}}^{*} = \sum_{k=1}^{K} n^{(k)} \left[\frac{m^{(k)}(m^{(k-1)} - m^{(k)})}{p} \right] \frac{N_{1}}{m^{(k-1)}}$$
$$\approx \frac{N_{1}}{p} \sum_{k=1}^{K} (m^{(k-1)} - m^{(k)}) = \frac{N_{1}(N_{1} - N_{2})}{p}$$
(9)

where $n^{(k)}m^{(k)} = m^{(k-1)}$, $m^{(0)} = N_1$, $m^{(K)} = N_2$. Similarly, the multiplication time of (4) with PERM⁽¹⁾ is

$$T_{\text{PERM}^{(1)}}^{*} = \sum_{k=1}^{K} (n^{(k)} + 1) \left[\frac{m^{(k)}(m^{(k-1)} - m^{(k)})}{p} \right]$$
$$\approx \frac{1}{p} \sum_{k=1}^{K} \left((m^{(k-1)})^{2} - (m^{(k)})^{2} \right) = \frac{N_{1}^{2} - N_{2}^{2}}{p} \quad (10)$$

where $n^{(k)}m^{(k)} = m^{(k-1)}, m^{(0)} = N_1, m^{(K)} = N_2.$

From (9) and (10), we see that, theoretically, the multiplication time has nothing to do with n—the block size for PERM⁽⁰⁾ and PERM⁽¹⁾.

If all $n^{(k)}$ are equal to n, then $m^{(k)} = m^{(k-1)}/n = m^{(0)}/n^k = N_1/n^k$, and we have $m^{(K)} = N_1/n^K = N_2$ or $K = \log_n(N_1/N_2)$.

As shown in Fig. 6, the additions cannot all be done in parallel, so the addition time is theoretically more complicated than the multiplication time.

For PERM⁽⁰⁾, if there are p processors and as many processors as possible are used in computation, the parallel addition time can be estimated as shown in (11) at the bottom of the next page.

For $PERM^{(1)}$, the parallel addition time can be estimated as shown in (12) at the bottom of the next page.

The above time estimations are related to a turning point K_p , where $m^{(K_p)}(m^{(K_p-1)}-m^{(K_p)})$ should be a number closest to but less than 2p. As the level k increases, the problem size decreases to such an extent that the speedup reaches its limit and cannot be improved further. In order to minimize the computational time, we can split the whole task into several phases and use a different processor allocation scheme in each phase.

However, PERM factorizations are not perfect. Suppose all $n^{(k)}$ are equal to n. The total number of rounding operations of PERM⁽¹⁾ is

$$\sum_{k=1}^{K} (n^{(k)} + 1)m^{(k)} = (n+1)N_1 \sum_{k=1}^{K} \frac{1}{n^k}$$
$$= (n+1)N_1 \frac{n^K - 1}{n^K (n-1)} = (N_1 - N_2) \frac{n+1}{n-1}$$
(13)

which is a decreasing function of n and achieves its minimum when $n = N_1/N_2$ and K = 1. For PERM⁽⁰⁾, the total number of rounding operations is KN. Hence, as the block size or the number of factorization levels grows, the number of the rounding operations also increases, possibly resulting in higher transform error though integer reversibility is still guaranteed. However, the transform error can be reduced to *any* extent by using a proper rounding operator. See Appendix B for details.

V. STRATEGY FOR MATRIX BLOCK PARTITIONING

How to partition a matrix and allocate the data to processors is a practical problem in applying PERM factorizations, for it determines the parallel complexity of the corresponding integer transform. Generally speaking, an appropriate block partitioning strategy is made according to some specific optimization principles. Ignoring other factors like the communications and the multiprocessor architecture, we have simply considered the computation time of parallel multiplications and parallel additions as the complexity metrics to evaluate the block structure of PERM⁽¹⁾ as follows.

Since there exists a turning point in the performance of a parallel implementation, it is necessary to consider the block structure (submatrix size) in the case of limited or abundant processors. Besides, if only a few processors are available, row distribution should be given first priority for it leads to a higher degree of parallelism for addition. From the above discussion, we propose a three-phase strategy for determining the level of block partitioning and block sizes.

- i) If N ≥ 2p, we factorize the matrix recursively in the first phase until the block size is reduced to p, i.e., N → ... → p. In this phase data are allocated in rows to the multiprocessor. To minimize the transform error, we can employ immediate one-level block factorization of N/p blocks.
- ii) If 2√p ≤ N < 2p, then perform N → ··· → √p in this phase. In mapping the data onto processors, we still give priority to row distribution. Again, a straightforward factorization is reasonable with the block size √p.
- iii) If $N \leq 2\sqrt{p}$, then $N \to \cdots \to 1$. Processors are excessive in this phase. To minimize the parallel cost of multiplication or, equivalently, the number of matrices, we have

$$\sum_{k=1}^{K} (n^{(k)} + 1) = (n+1)K = (n+1)\log_n N$$
(14)

where

$$n = n^{(1)} = \frac{m^{(0)}}{m^{(1)}} = n^{(2)} = \frac{m^{(1)}}{m^{(2)}} = \dots = n^{(K)} = \frac{m^{(K-1)}}{m^{(K)}}.$$

It follows that the minimum value can be obtained at n = 4, i.e., partitioning into four blocks at each level is the best solution. In such case, the parallel computation time of additions and multiplications is

$$T_{\text{PERM}^{(1)}}^* = \sum_{k=1}^{K} (n^{(k)} + 1) = 5 \log_4 N$$
(15)

$$T_{\text{PERM}^{(0)}}^{+} = \sum_{k=1}^{K_{p}} n^{(k)} \left[\frac{\left(m^{(k)} (m^{(k-1)} - m^{(k)}) - p \right)}{p} + C \left(\log_{2} p - \log_{2} m^{(k)} \right) \right] \frac{N_{1}}{m^{(k-1)}} + \sum_{k=K_{p}}^{K} n^{(k)} \left[\log_{2} (m^{(k-1)} - m^{(k)}) \right] \frac{N_{1}}{m^{(k-1)}}$$
(11)

$$T_{\text{PERM}^{(1)}}^{+} = \sum_{k=1}^{K_{p}} (n^{(k)} + 1) \left[\frac{(m^{(k)}(m^{(k-1)} - m^{(k)}) - p)}{p} + C\left(\log_{2} p - \log_{2} m^{(k)}\right) \right] + \sum_{k=K_{p}}^{K} (n^{(k)} + 1) \left[\log_{2}(m^{(k-1)} - m^{(k)})\right]$$
(12)

TABLE I TIME COMPLEXITY COMPARISON

operations	p	O(N)	$O(N^2)$
Multiplications	SERM ⁽¹⁾	O(N)	O(N)
Multiplications	PERM ⁽¹⁾	O(N)	$O(\log N)$
Additions	SERM ⁽¹⁾	$O(N \log N)$	O(NlogN)
Additions	PERM ⁽¹⁾	O(N)	$O(\log^2 N)$

$$T_{\text{PERM}^{(1)}}^{+} = \sum_{k=1}^{K} (n^{(k)} + 1) \log_2 \left((n^{(k)} - 1)m^{(k)} \right)$$
$$= \sum_{k=1}^{K} 5 \log_2 \frac{3N}{4^k} = 5 \log_4 N \left(\log_4 9N - 1 \right). \quad (16)$$

We now draw a comparison on the computation time between the above block partitioning scheme and the direct parallelization of SERM⁽¹⁾. Let $T_{\rm PERM^{(1)}}^{*/+}$ and $T_{\rm pSERM^{(1)}}^{*/+}$ denote the time complexity of parallel addition/multiplication with PERM factorization and SERM factorization, respectively. For above block partitioning strategy, we have (17)–(20) shown at the bottom of the page.

First, the number of effective processors can be up to $(1/4)N^2$ for PERM integer transform. From (17) and (18), it is easy to show that the costs of multiplication and addition are both O(N)when p = O(N) and are $O(\log N)$ and $O(\log^2 N)$, respectively, when $p = O(N^2)$. By contrast, the number of effective processors cannot exceed N for SERM⁽¹⁾ transform and, therefore, for p = O(N) or $p = O(N^2)$, the time of multiplication and addition remains O(N) and $O(N \log N)$, respectively.

Table I lists the order of multiplications and additions for SERM⁽¹⁾ and PERM⁽¹⁾ for processor number O(N) and $O(N^2)$. Generally speaking, PERM⁽¹⁾ is at least not worse than SERM⁽¹⁾ for the processor number of order O(N). When the processor number becomes $O(N^2)$, there is no change in SERM⁽¹⁾ while the performance of PERM⁽¹⁾ improves still.

Figs. 7 and 8 are examples for N = 64 and C = 1, comparing the performance of PERM⁽¹⁾ and SERM⁽¹⁾ with respect to the processor number. As can be seen in Fig. 7, the SERM



Fig. 7. Number of parallel operations $(T^{*/+})$ of PERM⁽¹⁾ and parallel SERM⁽¹⁾ transforms (N = 64, C = 1).



Fig. 8. Speedup of PERM⁽¹⁾ over parallel SERM⁽¹⁾ integer transforms (N = 64, C = 1).

complexity remains the same after the processors are more than 64, whereas PERM complexity continues to decrease.

Of course, since communications and other overheads have been ignored, the above block partitioning strategy is only an illustration of what can be done. In practice, the blocking can be flexible to satisfy different requirements. For instance, to accommodate as many processors for parallel computing as possible, we could use multilevel binary partitioning.

Although the total problem size of PERM⁽⁰⁾, distinct from that of PERM⁽¹⁾, also drops (yet slower) as the level k increases, the number of effective rows in each matrix can remain unchanged: at level k, there are altogether $N/n^{(k)}$ components

$$T_{\text{PERM}^{(1)}}^{*}(N,p) = \begin{cases} f_{1}^{*}(N,p) = \left(\frac{N}{p}+1\right) \cdot \left(\frac{N}{p}-1\right) \cdot p + f_{2}^{*}(p,p), & p \leq \frac{N}{2} \\ f_{2}^{*}(N,p) = \left(\frac{N}{\sqrt{p}}+1\right) \cdot \left(\frac{N}{\sqrt{p}}-1\right) + f_{3}^{*}(\sqrt{p},p), & \frac{N}{2} (17)$$

$$T_{\text{PERM}^{(1)}}^+(N,p) = \begin{cases} f_1^+(N,p) = \binom{N}{p} + 1 \cdot \binom{N}{p} - 1 \cdot p + f_2^+(p,p), & p \le \frac{N}{2} \\ f_2^+(N,p) = \binom{N}{\sqrt{p}} + 1 \cdot \binom{N}{\sqrt{p}} - 1 + C\log_2\sqrt{p} + f_3^+(\sqrt{p},p), & \frac{N}{2} (18)$$

$$\int_{2}^{2} (N,p) = \left(\frac{\sqrt{p}}{\sqrt{p}} + 1\right) \cdot \left(\frac{\sqrt{p}}{\sqrt{p}} + 1 + C \log_2 \sqrt{p}\right) + \int_{3}^{2} (\sqrt{p},p), \quad \frac{1}{2}
$$\int_{3}^{4} (N,p) = 5 \log_4 N \left(\log_4 9N - 1\right), \qquad p \ge \frac{N^2}{4}$$$$

while
$$T^*_{\text{pSERM}^{(1)}}(N,p) = (N+1)\frac{N-1}{p}$$
 (19)

$$T_{\rm pSERM^{(1)}}^+(N,p) = (N+1)\left(\frac{N-1}{p} + C\log_2 p\right)$$
(20)

updated in a single step, whereas the number for PERM⁽¹⁾ is $N/(n^{(1)} \cdots n^{(k)})$. This trait is conducive to row allocation to efficiently utilize processor resources. For instance, assuming each $N/n^{(k)}$ is a multiple of p, the parallel complexity of multiplication and addition for both cases is $(N^2 - N)/p$.

VI. EXAMPLES

We demonstrate our PERM factorizations by a commonly used transform: discrete Fourier transform (DFT).

A. Example: 9-by-9 DFT Matrix

The *N*-by-*N* DFT matrix is defined by $A = [\exp(-j \cdot 2\pi i k/N)]_{i,k}$; see the first equation at the bottom of the page. Without the loss of generality, suppose n = 3 and K = 2. Since $\det(A) = 3^9 = 19683$, *A* should be scaled to implement perfectly reversible integer transform.

1) $PERM^{(0)}$ Factorization: From (5), $A = PD(L^{(2)}U^{(2)})(L^{(1)}U^{(1)})$, where we have

Now scale
$$A$$
 by $A' = PD^{-1}P^T A$, where we have

$$PD^{-1}P^{T} = \text{Diag}(1, -0.5 + 0.2887i, 0.0193 - 0.1094i, -0.5 + 0.2887i, -0.1895 - 0.1590i, 0.1466 + 0.2539i, -0.1667 + 0.2887i, -0.2155 + 0.0380i, -0.0887 + 0.2437i).$$

Then we can obtain PERM⁽⁰⁾ factorization, $A' = PS_3^{(2)}S_2^{(2)}S_1^{(2)}S_3^{(1)}S_2^{(1)}S_1^{(1)}$, where all PERMs from right to left can be filled one by one into a matrix of the same size as A; see the equation at the bottom of the page. We call this matrix an integer transform matrix of A'.

2) *PERM*⁽¹⁾ *Factorization:* In order to keep the physical equivalents of the original transform, we scale A in a proportional way: A'' = (1/3)A. From (6), we have $A'' = PDS_3^{(2)}S_2^{(2)}S_1^{(2)}S_0^{(2)}S_3^{(1)}S_2^{(1)}S_1^{(1)}S_0^{(1)}$, where

	Γ0	0	0	1	0	0	0	0	[0	
	1	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	1	0	
	0	0	1	0	0	0	0	0	0	
P =	0	0	0	0	1	0	0	0	0	
	0	0	0	0	0	0	0	0	1	
	0	1	0	0	0	0	0	0	0	
	0	0	0	0	0	1	0	0	0	
	$\lfloor 0 \rfloor$	0	0	0	0	0	1	0	0	
D = diag(1, 1, 1, 1, 1, 1, 1, 1, -1).										

Likewise, we can store all PERMs in a 9-by-9 matrix as we did in Section VI-A1, except that the last block-row of S_0 is written along the diagonal downwards. Finally, after overwriting the bottom right element with the last one of D, i.e., det($P^T A''$), we get an integer transform matrix of A'' as shown in the equation at the bottom of the next page. Clearly, the storage of PERM

	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
	1.0000	0.7660 - 0.6428i	0.1736 - 0.9848i	-0.5000 - 0.8660i	-0.9397 - 0.3420i	-0.9397 + 0.3420i	-0.5000 + 0.8660i	0.1736 + 0.9848i	0.7660 + 0.6428i
	1.0000	0.1736 - 0.9848i	-0.9397 - 0.3420i	-0.5000 + 0.8660i	0.7660 + 0.6428i	0.7660 - 0.6428i	-0.5000 - 0.8660i	-0.9397 + 0.3420i	0.1736 + 0.9848i
4	1.0000	-0.5000 - 0.8660i	-0.5000 + 0.8660i	1.0000	-0.5000 - 0.8660i	-0.5000 + 0.8660i	1.0000	-0.5000 - 0.8660i	-0.5000 + 0.8660i
<i>A</i> =	1.0000	-0.9397 + 0.3420i	0.7660 - 0.6428i	-0.5000 + 0.8660i	0.1736 - 0.9848i	$0.1736 \pm 0.9848i$	-0.5000 - 0.8660i	0.7660 + 0.6428i	-0.9397 - 0.3420i
	1.0000	-0.5000 + 0.8660i	-0.5000 - 0.8660i	1.0000	-0.5000 + 0.8660i	-0.5000 - 0.8660i	1.0000	-0.5000 + 0.8660i	-0.5000 - 0.8660i
	1.0000	0.1736 + 0.9848i	-0.9397 + 0.3420i	-0.5000 - 0.8660i	0.7660 - 0.6428i	0.7660 + 0.6428i	-0.5000 + 0.8660i	-0.9397 - 0.3420i	0.1736 - 0.9848i
	1.0000	0.7660 + 0.6428i	0.1736 + 0.9848i	-0.5000 + 0.8660i	-0.9397 + 0.3420i	-0.9397 - 0.3420i	-0.5000 - 0.8660i	0.1736 - 0.9848i	0.7660 - 0.6428i

ſ	[1]	[1.0000][1.0000]	[1.0000		0		0	7	1.0000	0		0]
	[-0.5000 ·	+ 0.2887i]	[1][0.5000	- 0.8660i]	0		1.0000		0		0	1.0000		0
	[-0.3333]	0	- 0.5774i][1]	0		0		1.0000		0	0	1.000	0
	-0.3534 -	- 0.1140i	0.2245 -	+ 0.0396i	0.1975 -	- 0.1140i]	[1][0.7660 -	0.6428i][0.1736 - 0.98	848i]][0.0603 + 0.3420	i -0.6736	- 0.1188i -0.592	24 + 0.3420i]
	-0.1466 ·	+ 0.1747i	-0.5000 -	0.0607i	0.1466 +	0.1747i	[-0.4397	- 0.2539i][1][-0.1736 - 0.30	008i]	0.4397 - 0.5240	0.5000 -	+ 0.1820i -0.439	97 - 0.5240i
	-0.1975 -	- 0.1140i	-0.2245 -	+ 0.0396i	-0.6466	- 0.1140i	[0.1975 -	- 0.3420i][0.5686 -	0.1003i][1][0.5924 + 0.3420	i 0.6736 -	0.1188i 0.939	07 + 0.3420i
	-0.3333		0		0]	0 -	+ 0.1925i	-0.3132	- 0.1808i	-0.2553 + 0.1	474i][[1][-0.3473][1.000) [[c
	0		-0.3333		0		0.3132 -	0.1808i	0 +	+ 0.1925i	-0.3132 - 0.18	308i	[0.3711 - 0.4423i][1][-0.173	6 + 0.9848i]
	0		0		-0.3333		0.2553 -	+ 0.1474i	0.3132	- 0.1808i	0 + 0.19	925i 📗	[-0.4220 + 0.1536i][0+	0.5077i][1]

integer transform does not increase in comparison with that of the original DFT transform.

VII. CONCLUDING REMARKS

In the above discussions, we have presented PERM factorizations for parallel reversible integer transforms based on block factorizations. Compared with SERM factorizations, they offer improved parallel performance. Particularly, they increase the degree of parallelism and thus accommodate more processors. Since the PERM factorization and the corresponding integer transforms can all be calculated at the block level, we also expect increased efficiency in sequential computation with some matrix computation software (such as BLAS) speeding block operations. Consequently, PERM factorizations are attractive for large matrix integer transforms. Considering the flexibility of the scaling modification, $PERM^{(1)}$ could be more promising in real-world applications.

One problem is that the problem size gradually drops when level k increases, which will probably reduce the availability of processors. This cannot be ignored, especially when $PERM^{(1)}$ is employed with relatively more processors.

The key to applying PERM factorizations is the proper choice of the block partitioning strategy. Including other necessary factors such as the communications, our future work will study this problem systematically and test the performance by further experimentation.

APPENDIX A DEFINITION AND PROPERTIES OF DET

In this Appendix, we give a simple introduction of the **DET** used in BLUS factorization. Interested readers may refer to [11] and [12] for details. We shall use

$$oldsymbol{A}_n = egin{bmatrix} oldsymbol{A}_{11} \cdots oldsymbol{A}_{1n} \ \cdots \ oldsymbol{A}_{nn} \cdots oldsymbol{A}_{nn} \ oldsymbol{A}_{nn} \end{bmatrix}$$

to denote a square block matrix with n rows and n columns of blocks. Assume $A_{i,j}$ are of the same size. Before defining **DET**, we first define an auxiliary matrix function $W^{(n)}$.

Definition 1: Given a block matrix $A_n, W^{(n)}(A_n)$ is recursively defined as follows.

i) For
$$n = 1$$
, $W^{(1)}(A_1) := A_{11}$.

ii) For n = 2, if A_{12} is invertible, then $W^{(2)}(A_2) :=$ $A_{22}A_{12}^{-1}A_{11} - A_{21}.$

iii) For $n \ge 3$, if $\boldsymbol{W}^{(n-1)}(\boldsymbol{A}_n(1,\ldots,n-1;1\ldots,n-2,n))$ exists and is invertible, then $\boldsymbol{W}^{(n)}(\boldsymbol{A}_n)$ is defined by $W^{(n)}(A_n) := D^{(n)}_{n,n-1} - D^{(n)}_{n,n-2}(D^{(n)}_{n-1,n-2})^{-1}D^{(n)}_{n-1,n-1}$, where $D^{(n)}_{i,j}$ is a notation for $W^{(n-1)}(A_n(1,\ldots n-2,i;1,\ldots n-3,j,n))$ with $n-1 \le i \le n$ and $n-2 \le j \le n-1$.

Definition 2: Given a block matrix A_n , $DET(A_n)$ is recursively defined as follows.

- i) For n = 1, **DET** $(A_1) := A_{11}$.
- ii) For $n \ge 2$, if $W^{(n)}(A_n)$ is defined, then $DET(A_n)$ exists and is defined by

 $\mathbf{DET}(\mathbf{A}_n)$:

$$= \boldsymbol{W}^{(n)}(\boldsymbol{A}_n) \cdot \mathbf{DET}(\boldsymbol{B}_{n-1})$$

= $\boldsymbol{W}^{(n)}(\boldsymbol{A}_n) \cdot \mathbf{DET}(\boldsymbol{A}_n(1,\ldots,n-1;1,\ldots,n-2,n)).$

Some important properties are listed as follows.

Triangular Property: Assume $DET(A_n)$ is defined. If $A_{11} = I$ and $A_{21} = \cdots = A_{n,1} = 0$, then $\mathbf{DET}(\mathbf{A}_n) = \mathbf{DET}(\mathbf{A}_n(2,\ldots,n;2,\ldots,n)), \forall n \ge 2$ Scaling Property:

$$\begin{aligned} \mathbf{DET} \left(\begin{bmatrix} \boldsymbol{X}_1 & & \\ & \ddots & \\ & & \boldsymbol{X}_n \end{bmatrix} \begin{bmatrix} \boldsymbol{A}_{11} & \cdots & \boldsymbol{A}_{1n} \\ \cdots & \cdots & \cdots \\ \boldsymbol{A}_{n1} & \cdots & \boldsymbol{A}_{nn} \end{bmatrix} \right) \\ &= \boldsymbol{X}_n \cdot \boldsymbol{W}^{(n)}(\boldsymbol{A}_n) \cdots \boldsymbol{X}_1 \cdot \boldsymbol{W}^{(1)}(\boldsymbol{A}_1) \\ &\mathbf{DET} \left(\begin{bmatrix} \boldsymbol{A}_{11} & \cdots & \boldsymbol{A}_{1n} \\ \cdots & \cdots & \cdots \\ \boldsymbol{A}_{n1} & \cdots & \boldsymbol{A}_{nn} \end{bmatrix} \begin{bmatrix} \boldsymbol{Y}_1 & & \\ & \ddots & & \\ & & \boldsymbol{Y}_n \end{bmatrix} \right) \\ &= \boldsymbol{W}^{(n)}(\boldsymbol{A}_n) \cdot \boldsymbol{Y}_{n-1} \cdots \boldsymbol{W}^{(2)}(\boldsymbol{A}_2) \cdot \boldsymbol{Y}_1 \cdot \boldsymbol{W}^{(1)}(\boldsymbol{A}_1) \cdot \boldsymbol{Y}_n \end{aligned}$$

where X_i $(1 \le i \le n-1)$ and Y_j $(1 \le j \le n, j \ne n-1)$ should be invertible.

This property, as a counterpart of that of matrix determinant DET, is an important guarantee of the flexibility and practicability of the scaling modification.

APPENDIX B TRANSFORM-ERROR CONTROL

As mentioned in Section IV, one disadvantage of PERM factorizations is that larger block size and more factorization levels result in more rounding operations and possibly higher transform error. However, noticing that | in (1) and (2) can actually be any nonlinear operator, we may keep more bits after the decimal point (e.g., rounding to hundredths or thousandths) to

Γ	0.5603 + 0.5240i	-1.4187 - 0.0413i	-1.9397 + 0.5796i	-0.0000 - 0.5774i	-0.5000 - 0.2887i	-0.5000 - 0.2887i][-0.1667 + 0.2887i	$0.0579 \pm 0.3283i$	0.2553 + 0.2143i
	-0.4397 + 0.5240i	1.0813 + 0.8248i	-0.4397 - 0.2864i	-0.5000 + 0.2887i	1.0686 - 0.7658i	-0.3711 + 0.4423i	0.3333 - 0.0000i	-0.1667 + 0.2887i	-0.1667 - 0.2887i
L	-0.4397 + 0.5240i	0.0813 - 0.9073i	0.5603 + 1.4456i	-0.5000 + 0.2887i	-0.5686 + 0.1003i	0.8711 - 0.4237i	0.3333 - 0.0000i	-0.1667 - 0.2887i	-0.1667 + 0.2887i
Γ	0.4397 - 0.5240i	$0.4187 \pm 0.0413i$	0.9397 - 0.5796i	1.8264 - 0.9848i	-0.1318 + 0.3833i	-0.5282 + 0.5796i	-0.1975 - 0.0348i	0.5415 - 0.0533i	0.1560 - 0.2005i
	0.5000 - 0.0882i	-1.0924 + 0.0163i	0.6133 + 1.2387i	-0.6736 - 0.1188i	$2.3682 \pm 0.8028i$	-0.5282 - 0.4684i	0.0509 - 0.2887i	-0.2352 + 0.3283i	-0.1868 - 0.2539i
L	0.2340 - 0.1351i	0.3473 + 1.1918i	-1.0813 + 0.7310i	-0.6736 - 0.1188i	-0.1318 - 2.0328i	1.9718 + 1.6832i	0.0780 - 0.4423i	-0.3604 - 0.2749i	-0.2861 + 0.3889i
Γ	0.4269 + 0.5968i	-1.5627 + 0.6686i	-2.0345 + 0.1390i]	0.2106 - 0.3468i	0.0247 - 0.9177i	-0.4515 - 1.4111i	[1.3392 + 0.4386i]	[-0.3467+0.3075i][-0.7599 - 0.1424i]]
	0.4954 - 0.1786i	0.2636 + 1.5743i	1.6391 - 1.5535i	-0.7894 - 0.4260i	-0.1728 + 0.8312i	-0.3226 - 0.6526i	[-0.1092 + 1.3345i]	[-0.0873 + 1.0636i][-0.4196 + 0.5959i]
L	0.1243 + 0.2912i	2.7033 - 1.1378i	-0.8006 + 0.9306i	0.7894 - 0.1980i	0.8958 - 1.0988i	-1.0201 + 0.8076i	[2.1058 - 1.1169i]	[0.7393 + 0.6162i	[-1.0000 - 0.0000i]

effectively reduce the transform error. This can be justified by the error bounds given in [9] and [12].

Given a factorization $A = PV_M \cdots V_1$, we introduce M diagonal matrices E_{V_i} to label the positions of possible rounding error in the transformation with V_i . For example, $E_L = \text{diag}(0, 1, \dots, 1)$, $E_U = \text{diag}(1, \dots, 1, 0)$, $E_{S_N} = \text{diag}(0, \dots, 0, 1)$. Let ε be the final error vector. If we consider the largest possible error, the error bound [9, Section VII] can be written as

$$\|\boldsymbol{\varepsilon}\|_{\infty} \leq u \cdot \left(\left\| \boldsymbol{E}_{\boldsymbol{V}_{M}} \right\|_{\infty}^{2} + \left\| \boldsymbol{V}_{M} \boldsymbol{E}_{\boldsymbol{V}_{M-1}} \right\|_{\infty}^{2} + \left\| \boldsymbol{V}_{M} \boldsymbol{V}_{M-1} \boldsymbol{E}_{\boldsymbol{V}_{M-2}} \right\|_{\infty}^{2} + \dots + \left\| \boldsymbol{V}_{M} \boldsymbol{V}_{M-1} \cdots \boldsymbol{V}_{2} \boldsymbol{E}_{\boldsymbol{V}_{1}} \right\|_{\infty}^{2} \right)$$

$$(21)$$

where u is the unit roundoff, defined as the largest error that can occur in one rounding operation.

Alternatively, if we care about the mean error, by characterizing the rounding errors as independent random noises, it can be formulated as [12, Section 2.4]

$$E(\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon}) = N\sigma^{2} \cdot \left(\left\| \boldsymbol{E}_{\boldsymbol{V}_{M}} \right\|_{F}^{2} + \left\| \boldsymbol{V}_{M}\boldsymbol{E}_{\boldsymbol{V}_{M-1}} \right\|_{F}^{2} + \left\| \boldsymbol{V}_{M}\boldsymbol{V}_{M-1}\boldsymbol{E}_{\boldsymbol{V}_{M-2}} \right\|_{F}^{2} + \dots + \left\| \boldsymbol{V}_{M}\boldsymbol{V}_{M-1}\cdots\boldsymbol{V}_{2}\boldsymbol{E}_{\boldsymbol{V}_{1}} \right\|_{F}^{2} \right)$$

$$(22)$$

where σ^2 is the variance of noise in one dimension and $\|\cdot\|_F$ is the Frobenius-norm.

These results also apply to block matrices, and the only difference is that the position matrix E_V for error propagation becomes more specific.

It is theoretically hard to find the optimal factorization minimizing (21) or (22). However, we may lower u or σ to reduce the transform error, since we can flexibly choose an arbitrary operator $| \ |$ as used in (1) and (2).

In summary, with quantitative error control formulas (21) and (22), we can determine the appropriate transform precision by choosing a proper rounding operator, thus achieving both perfect reversibility and very good transform-error control. (Interested readers may refer to [12] for details.)

REFERENCES

- H. Blume and A. Fand, "Reversible and irreversible image data compression using the S-transform and Lempel-Ziv coding," in *Proc. SPIE*, 1989, vol. 1091, pp. 2–18.
- [2] A. Zandi, J. D. Allen, E. L. Schwartz, and M. Boliek, J. A. Storer and M. Cohn, Eds., "CREW: Compression with reversible embedded wavelets," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, 1995, pp. 212–221.
- [3] A. Said and W. A. Pearlman, "An image multiresolution representation for lossless and lossy compression," *IEEE Trans. Image Process.*, vol. 5, pp. 1303–1310, 1996.
- [4] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," J. Appl. Comput. Harmon. Anal., vol. 3, no. 2, pp. 186–200, 1996.
- [5] F. A. M. L. Bruekers and A. W. M. van den Enden, "New networks for perfect inversion and perfect reconstruction," *IEEE J. Sel. Areas Commun.*, vol. 10, pp. 130–137, 1992.
- [6] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," J. Fourier Anal. Applicat., vol. 4, no. 3, pp. 247–269, 1998.

- [7] A. R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, "Wavelet transform that map integers to integers," J. Appl. Comput. Harmon. Anal., vol. 5, no. 3, pp. 332–369, 1998.
- [8] P. Hao and Q. Shi, "Invertible linear transforms implemented by integer mapping," (in Chinese) *Sci. China*, ser. E, vol. 30, no. 2, pp. 132–141, 2000.
- [9] —, "Matrix factorizations for reversible integer mapping," *IEEE Trans. Signal Process.*, vol. 49, pp. 2314–2324, 2001.
- [10] —, "Proposal of reversible integer implementation for multiple component transforms," in *ISO/IEC JTC1/SC29/WG1N1720*, Arles, France, Jul. 3–7, 2000.
- [11] Y. She and P. Hao, "Block TERM factorization of uniform block matrices," *Sci. China*, ser. F, vol. 47, no. 4, pp. 421–436, 2004.
- [12] Y. She, "Matrix factorizations for efficient implementation of linear transforms," (in Chinese) Master's thesis, Peking Univ., Beijing, China, 2003.
- [13] P. Hao, "Customizable triangular factorizations of matrices," *Linear Algebra Applicat.*, vol. 382, pp. 135–154, 2004.
- [14] Y. She and P. Hao, "On the necessity and sufficiency of PLUS factorizations," *Linear Algebra Applicat.*, vol. 400, pp. 193–202, 2005.
- [15] G. Plonka, "A global method for invertible integer DCT and integer wavelet algorithms," *Appl. Comput. Harmon. Anal.*, vol. 16, pp. 90–110, 2004.



Yiyuan She was born in China in 1978. He received the B.Sc. degree in mathematics and the M.Sc. degree in electrical engineering from Peking University, Beijing, China, in 2000 and 2003, respectively. He is currently pursuing the Ph.D. degree in statistics at Stanford University, Stanford, CA.

He was a Visiting Graduate Student at Queen Mary, University of London, London, U.K., in 2003. His research interests include signal processing, machine learning, statistical computing, and bioinformatics.



Pengwei Hao (M'98) was born in the north of Shaanxi Province, China, in 1966. He received the B.Sc. degree in computer science and the M.Sc. degree in computer graphics from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1994, respectively, and the Ph.D. degree in image processing from the Institute of Remote Sensing Applications, Chinese Academy of Sciences, Beijing, China, in 1997.

From 1997 to 1999, he was a Lecturer at the Center for Information Science, Peking University, Beijing.

In 2000, he was a Visiting Scientist for three months with the Centre for Vision, Speech, and Signal Processing, University of Surrey, Surrey, U.K. In 2002, he became a Lecturer at Queen Mary, University of London, London, U.K. He is also currently an Associate Professor with the Center for Information Science, Peking University. His research interests include data and image compression, data hiding, signal sampling and reconstruction, and computer graphics.



Yakup Paker (M'74) received the bachelor's degree in electrical engineering from Istanbul Technical University, Turkey, in 1958 and the M.S. and Ph.D. degrees from Columbia University, New York, in 1961 and 1965, respectively.

He is an Emeritus Professor of parallel computing at Queen Mary, University of London, London, U.K. He held academic positions at Middle East Technical University, Ankara, Turkey; the University of Westminster, London, U.K.; and Rennes University, Rennes, France, before joining Queen

Mary as a full Professor in 1990. His research interests have included computer systems architectures and parallel computers, real-time architectures for video processing, virtual studios, three-dimensional TV, and digital broadcasting convergence. He has undertaken a range of collaborative research projects, funded by the European Union and the British government, which have included developing a multiprocessor architecture and systems software for real-time video processing, a customized TV terminal, an advanced set-top box based interactive TV, and systems architecture for three-dimensional TV. He has published extensively and is the author/editor of a number of books and proceedings.