

Complexity Analysis of Two Permutations Used by Fast Cosine Transform Algorithms

Sean S. B. Moore*
BBN Systems and Technologies
Cambridge, MA 02138
smoore@bbn.com

Leonard F. Wisniewski†
Department of Computer Science
Dartmouth College
Hanover, NH 03755
wisnie@cs.dartmouth.edu

Dartmouth College PCS-TR95-266

October 10, 1995

Abstract

The fast cosine transform algorithms introduced in [ST91, Ste92] require fewer operations than any other known general algorithm. Similar to related fast transform algorithms (e.g., the FFT), these algorithms permute the data before, during, or after the computation of the transform. The choice of this permutation may be an important consideration in reducing the complexity of the permutation algorithm. In this paper, we derive the complexity to generate the permutation mappings used in [ST91, Ste92] for power-of-2 data sets by representing them as linear index transformations and translating them into combinational circuits. Moreover, we show that the permutation used in [Ste92] not only allows efficient implementation, but is also self-invertible, i.e., we can use the same circuit to generate the permutation mapping for both the fast cosine transform and its inverse, like the bit-reversal permutation used by FFT algorithms. These results may be useful to designers of low-level algorithms for implementing fast cosine transforms.

1 Introduction

Fast cosine transform (FCT) algorithms are essential to efficient computation in many applications such as weather modeling, data compression/decompression and convolutions on real, symmetric data [Chi95, ER82, RY90]. Recent descriptions of fast cosine transform algorithms [ST91, Ste92] claim record minimums for the number of multiplication and addition operations needed to compute an FCT. These new algorithms have been derived using the polynomial division tree computational model.

Algorithms developed on the polynomial division tree computational model can be represented as the application of a sequence of discrete operators, or matrices, on a finite data vector of

*Supported in part by DARPA as administered by the AFOSR under contract AFOSR-90-0292 while the first author was at Dartmouth College.

†Supported in part by a Dartmouth Fellowship and in part by the National Science Foundation under Grant CCR-9308667.

length N . We formulate the *forward transform* \mathcal{T} as a sequence of operators

$$\mathcal{T} = \mathcal{S}_k \cdots \mathcal{S}_1 \mathcal{P} \mathcal{W} \tag{1}$$

where the operators \mathcal{S}_j are highly-structured matrices, the operator \mathcal{P} is a permutation matrix which effects a reordering of the data sequence, the operator \mathcal{W} is a diagonal matrix with quadrature weights along the diagonal, and $k = O(\lg N)$. We apply the operators from right to left. We can often generate the *inverse transform* by applying the transpose of each factor from left to right.¹

The polynomial division tree model provides the framework for developing algorithms to compute transforms in time asymptotically less than the $O(N^2)$ bound of a naive algorithm. The matrices \mathcal{S}_j in equation (1) must be structured such that we can apply each matrix to a vector of length N in less than $O(N^2)$ operations. For the polynomial division tree algorithm, these matrices usually require $O(N \lg^m N)$ multiplication and addition operations, where $m \geq 0$ is an integer constant. Because there are $O(\lg N)$ of these matrices, applying all of the matrices \mathcal{S}_j requires $O(N \lg^{m+1} N)$ operations. We can typically apply the matrices \mathcal{W} and \mathcal{P} in $O(N)$ operations so that the bound to compute the transform is $O(N \lg^{m+1} N)$ operations.

As an example, we consider the FFT algorithm applied to a data sequence of length N with the assumption that we generate the data sequence by evaluating, or sampling, at uniformly spaced intervals, some underlying function that has a finite Fourier series representation. In this case, the matrix \mathcal{W} is the identity matrix and the permutation matrix \mathcal{P} represents the bit-reversal permutation. Since the matrices \mathcal{S}_j have $O(N)$ non-zero entries consisting of roots of unity, we can apply all these matrices in $O(N \lg N)$ operations.

For many fast transform algorithms that use a matrix formulation like equation (1), several different permutation matrices allow us to achieve the same overall computational complexity of the transform. Appropriate rearrangement of the entries in the matrices \mathcal{S}_j keeps the algorithm's computational complexity the same. The choice of permutation matrix \mathcal{P} may not be an issue when the implementation prestores the structure of the matrix; however, it may be very important if the implementation dynamically computes the structure of the permutation matrix, i.e., the location of the non-zero elements of the matrix.

For an FFT algorithm on power-of-2 data sets, the bit-reversal permutation is a desirable choice because it only requires a constant-depth, logarithmic-width circuit to define the structure of the permutation matrix. That is, for a data sequence of length N , where $N = 2^n$ for some integer $n \geq 0$, we represent each index of the data sequence by a $\lg N$ -bit vector. A circuit to perform the bit-reversal permutation mapping uses only one wire to connect each input bit i to output bit $\lg N - i - 1$. Because the input only needs to pass through one level of wires and the output has a width of $\lg N$ bits, the circuit has constant depth and logarithmic width. If we consider each connection from input to output to be a single bit-operation, then the bit-reversal permutation mapping of N data elements requires $O(N \lg N)$ bit-operations. Furthermore, the bit-reversal permutation is *self-invertible*, that is, we reverse the effect of a bit-reversal permutation by reapplying the bit-reversal permutation ($\mathcal{P}^2 = I$). Thus, self-invertibility allows the same constant-depth circuit to be used for computing the structure of the permutation matrix for both the forward and inverse transforms.

¹Some literature refers to the forward and inverse transforms as the interpolation and evaluation cases, respectively. In the evaluation case, we do not use the quadrature matrix \mathcal{W} .

In this paper, for power-of-2 data sets, we analyze the complexity of the two permutations used in the fast cosine transform algorithms described in [ST91] and [Ste92] and show that the permutation in [Ste92] exhibits properties similar to those of the bit-reversal permutation. We use the following technique to analyze the complexity of these permutations.

Step 1 We give high-level pseudocode for a series of linear permutations which generate the structure of the permutation matrix \mathcal{P} .

Step 2 We translate the linear permutations into nonsingular, or invertible, characteristic matrix forms, and we combine them into a single characteristic matrix form.

Step 3 We translate the characteristic matrix form into a combinational circuit consisting of only XOR gates.

We shall see later that the characteristic matrix forms that generate the structure of the permutation matrices described in [ST91, Ste92] clearly illustrate their translation into constant-depth, logarithmic-width circuits. We show, however, that the circuit to perform the inverse permutation mapping used by the fast cosine transform algorithm in [ST91] has $\lg \lg N$ -depth and logarithmic width. Fortunately, we also show that the circuit used by the fast cosine transform algorithm in [Ste92] is constant-depth, logarithmic-width, and self-invertible. Thus, the permutation of [Ste92] has properties similar to the bit reversal permutation, i.e., we can use the same constant-depth circuit to compute the structure of the permutation matrix for both the forward and inverse fast cosine transforms; therefore, the permutation of [Ste92] is preferable.

The organization of the remainder of this paper is as follows. In Section 2, we define the discrete cosine transform (DCT) and discuss in more detail the polynomial division tree model and a fast algorithm to compute the DCT. We also define the class of BMMC (bit-matrix multiply/complement) permutations which includes the linear permutations used by the fast cosine transform algorithms. In Section 3, we present the permutation of [ST91], which we call the *recursive complement*, or *RC, permutation*, and use the above methodology to analyze its complexity. In Section 4, we perform a similar analysis of the permutation of [Ste92], which we call the *odd-upper/bit-reversal*, or *OUR, permutation*, but we also show that the permutation is self-invertible. Section 5 concludes.

2 Definitions and Background

In this section, we define the discrete cosine transform (DCT) and give an overview of the fast cosine transform (FCT) algorithms of [ST91, Ste92] to compute the DCT. The permutations used by these FCT algorithms belong to the class of BMMC (bit-matrix multiply/complement) permutations. We also define the class of BMMC permutations and provide the linear index transformation framework which we use extensively in the next two sections.

The Discrete Cosine Transform (DCT)

We can define the N -point DCT $\hat{\mathbf{f}}$ of a sequence $\mathbf{f} = f(0), f(1), \dots, f(N-1)$ as

$$\hat{f}(j) := \frac{1}{N} \sum_{k=0}^{N-1} f(k) \cos\left(\frac{\pi(2k+1)j}{2N}\right) \quad (j = 0, 1, \dots, N-1) \quad (2)$$

and the inverse DCT as

$$f(k) := \sum_{j=0}^{N-1} \hat{f}(j) \cos\left(\frac{\pi(2k+1)j}{2N}\right) \quad (k = 0, 1, \dots, N-1). \quad (3)$$

The computation of the DCT in equation (2) corresponds to a matrix-vector product $\hat{\mathbf{f}} = \mathbf{C}\mathbf{f}$:

$$\begin{pmatrix} \hat{f}(0) \\ \hat{f}(1) \\ \vdots \\ \hat{f}(N-1) \end{pmatrix} = \begin{pmatrix} C(0,0) & C(0,1) & \cdots & C(0,N-1) \\ C(1,0) & C(1,1) & \cdots & C(1,N-1) \\ \vdots & \vdots & \vdots & \vdots \\ C(N-1,0) & C(N-1,1) & \cdots & C(N-1,N-1) \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ \vdots \\ f(N-1) \end{pmatrix}$$

where $C(j,k) = \cos(\frac{\pi(2k+1)j}{2N})$. Because the transformation matrix \mathbf{C} is orthogonal (when properly normalized), the matrix-vector product for the inverse DCT in equation (3) is

$$\mathbf{f} = \mathbf{C}^t \hat{\mathbf{f}}.$$

The fast inverse cosine transform algorithm of [ST91] corresponds to a sparse factorization of the matrix \mathbf{C}^t such that

$$\mathbf{C}^t = \mathbf{P}^t \mathbf{D}_0^t \mathbf{D}_1^t \cdots \mathbf{D}_{n-1}^t \quad (4)$$

where each matrix \mathbf{D}_i^t is sparse (i.e., the matrix contains $O(N)$ non-zero entries) and the matrix \mathbf{P}^t is a permutation matrix. We transpose all the matrices for the inverse FCT algorithm to obtain the forward FCT algorithm. We refer the interested reader to the original paper [ST91] for more specific details, or to an expanded discussion in [Moo94].

The fast cosine transform algorithm uses the polynomial division tree method for evaluating a polynomial function [BM75, DHR94, Knu81, Moo94]. In the *polynomial division tree model*, we represent a function as a polynomial $p(z)$. The evaluation of the function at a point z_i is equivalent to the remainder of the division of $p(z)$ by the linear polynomial $(z - z_i)$:

$$p(z_i) \equiv p(z) \bmod (z - z_i).$$

This straightforward approach does not result in an asymptotically fast algorithm; however, consider the evaluation of $p(z)$ at a set of points z_0, z_1, \dots, z_k , where $p(z)$ is a polynomial of degree $N-1$ and $k < N-1$. For any $i, j \in \{0, 1, \dots, k\}$, we have

$$\begin{aligned} p(z_i) &\equiv p(z) \bmod \left(\prod_{l=0}^k (z - z_l) \right) \bmod (z - z_i), \\ p(z_j) &\equiv p(z) \bmod \left(\prod_{l=0}^k (z - z_l) \right) \bmod (z - z_j). \end{aligned}$$

For the evaluations, a divide-and-conquer approach in which an appropriate selection of and division by *supermoduli* $\prod_l (z - z_l)$ results in a fast algorithm for evaluating $p(z)$ at a set of evaluation points. Using this method, we can derive an algorithm for the fast Fourier transform (FFT) by evaluating at the N th roots of unity (i.e., at uniformly spaced intervals) and by substituting $e^{\frac{ik2\pi}{N}}$ for z_k when evaluating $p(z_k)$.

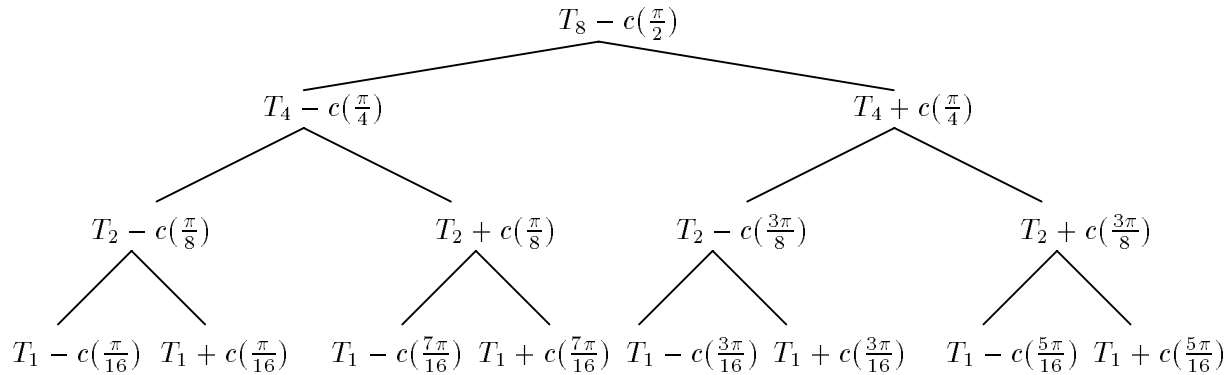


Figure 1: Chebyshev supermoduli tree. In the above tree, we use the notational simplification $c(\frac{(2k+1)\pi}{16}) = \cos(\frac{(2k+1)\pi}{16})$. Since each supermodulus ($T_j - x$) is the product of its children, we first divide the polynomial by the supermodulus ($T_8 - x$) at the top of the tree, then supply the remainder as input to a division by each child supermodulus. The computation recurses down the tree until each leaf in the division tree results in a remainder that is equivalent to $p(x_k)$.

In [ST91], the polynomial division tree derivation for the DCT proceeds by first casting the inverse DCT as the evaluation of a *Chebyshev polynomial* $p(x)$ such that

$$p(x) = \sum_{j=0}^{N-1} c_j T_j(x)$$

where we define $T_j(x) = \cos j(\arccos x) = T_j$ as the j th Chebyshev polynomial. We evaluate the Chebyshev polynomial at the zeroes of T_N , i.e., at $\cos(\frac{\pi(2k+1)}{2N})$, for $k = 0, 1, \dots, N - 1$. Since $p(x_k) \equiv (\sum_{j=0}^{N-1} c_j T_j(x)) \bmod (T_1 - x_k)$, we derive a fast algorithm by building a polynomial division tree that has sparse supermoduli with at most two non-zero coefficients. We use the same method for computing the forward DCT. Figure 1 illustrates the polynomial division tree for evaluating Chebyshev polynomials when $N = 8$.

We minimize the total number of operations by ordering the leaves of the tree in such a way as to ensure that the supermoduli are sparse, i.e., each supermodulus contains only two non-zero coefficients. The ordering of leaves corresponds to a permutation of the output data sequence. Observe that many permutations on the leaves result in a fast algorithm, i.e., rotation of a subtree in the division tree does not change the evaluation of the polynomial at the N points. For example, by rotating the subtree rooted at the $T_2 - c(\frac{\pi}{8})$ in Figure 1, the swap of leaves $T_1 - c(\frac{\pi}{16})$ and $T_1 + c(\frac{\pi}{16})$ does not change the complexity of the FCT algorithm, but does change the ordering of the output sequence. We use the permutation effected by the permutation matrix \mathbf{P} to reorder the data appropriately.

When the size of the data set is a power of 2, the fast algorithms in [ST91] and [Ste92] both compute the correct evaluation, but perform different permutations to achieve it. In the next two sections, we shall examine these two permutations in detail and show that the permutation of [Ste92] has more appealing properties.

Before continuing our discussion, we provide some formal definitions to clarify the difference between the permutation represented by the permutation matrix \mathbf{P} and the characteristic matrix

which generates the structure of the permutation matrix \mathbf{P} . A *permutation* is an arbitrary mapping $\pi : \{0, 1, \dots, N - 1\} \xrightarrow{1-1} \{0, 1, \dots, N - 1\}$. For this discussion, the domain is the set of indices for the N data points evaluated by the function \mathbf{f} . We define the *permutation matrix* \mathbf{P} as a nonsingular, or invertible, $N \times N$ matrix in which each entry is an element of $\{0, 1\}$ over \mathbb{C}^N and each row and column has exactly one 1.

For power-of-2 data sets, we also define a nonsingular, or invertible, *characteristic matrix* which generates the structure of the permutation matrix \mathbf{P} . That is, the characteristic matrix defines a linear transformation of the column indices of the $N \times N$ identity matrix to become the permutation matrix \mathbf{P} . Since a characteristic matrix is nonsingular, the columns of the identity matrix map one-to-one to the permutation matrix \mathbf{P} . The class of linear transformations for power-of-2 data sets belongs to the class of *BMMC (bit-matrix multiply/complement) permutations*.

BMMC permutations

We now describe more formally the class of BMMC permutations. The class of BMMC permutations is the most general class of bit-defined permutations². These permutations are defined for all data sets that contain N elements, where $N = 2^n$ for some integer $n \geq 0$. For convenience, we shall extensively use the notation $n = \lg N$. For a BMMC permutation, we have an $n \times n$ *characteristic matrix* $A = (a_{ij})$ whose entries are drawn from $\{0, 1\}$ and is nonsingular (i.e., invertible) over $GF(2)$, and we have a *complement vector* $c = (c_0, c_1, \dots, c_{n-1})$ of length n . Treating a *source index* x as an n -bit vector, we perform matrix-vector multiplication over $GF(2)$ ³ and then form the corresponding n -bit *target index* y by complementing some subset of the resulting bits: $y = Ax \oplus c$, or

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \oplus \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}.$$

In this paper, we only examine permutations for which the complement vector is 0, so we ignore the complement vector for the remainder of the paper. Thus, the permutation π_A characterized by a matrix A is the permutation for which $\pi_A(x) = Ax$ for all source indices x , for $x = 0, 1, \dots, N - 1$.

We use the characteristic matrix to generate the structure of the permutation matrix \mathbf{P} . For example, in the case $N = 16$, if $\pi_A(10) = 13$, then column 10 in the permutation matrix \mathbf{P} is the same as the column 13 of the identity matrix, i.e., there is a 1 in location (13, 10) of the permutation matrix \mathbf{P} .

BMMC permutations include the subclass of BPC (bit-permute/complement) permutations which have characteristic matrices with exactly one 1 in each row and column. BPC permutations include many common permutations such as the bit-reversal permutation, matrix transposition, vector-reversal permutations, hypercube permutations, and matrix reblocking. BMMC permuta-

²Edelman, Heller, and Johnson [EHJ94] call BMMC permutations *affine transformations* or, if there is no complementing, *linear transformations*.

³Matrix multiplication over $GF(2)$ is like standard matrix multiplication over the reals but with all arithmetic performed modulo 2. Equivalently, multiplication is replaced by logical-and, and addition is replaced by exclusive-or.

tions also include non-BPC permutations such as the standard binary-reflected Gray code and the permutations discussed in this paper.

We use several properties of characteristic matrices extensively throughout Sections 3 and 4. The following lemma from [CSW94] shows the equivalence of multiplying characteristic matrices and composing permutations when the complement vectors are zero. For permutations π_Y and π_Z , we define the *composition* $\pi_Z \circ \pi_Y$ as $(\pi_Z \circ \pi_Y)(x) = \pi_Z(\pi_Y(x))$ for all x in the domain of π_Y .

Lemma 1 *Let Z and Y be nonsingular $n \times n$ matrices and let π_Z and π_Y be the permutations characterized by Z and Y , respectively. Then the matrix product ZY characterizes the composition $\pi_Z \circ \pi_Y$. ■*

We can decompose a characteristic matrix A into a product of several nonsingular matrices, each of which characterizes a BMMC permutation. The following corollary from [CSW94] describes the order in which we perform these permutations to effect the permutation characterized by A .

Corollary 2 *Let the $n \times n$ characteristic matrix A be factored as $A = A^{(k)} A^{(k-1)} A^{(k-2)} \dots A^{(1)}$, where each factor $A^{(i)}$ is a nonsingular $n \times n$ matrix. Then we can perform the BMMC permutation characterized by A by performing, in order, the BMMC permutations characterized by $A^{(1)}, A^{(2)}, \dots, A^{(k)}$. That is, we perform the permutations characterized by the factors of a matrix from right to left. ■*

In Sections 3 and 4, we use Corollary 2 to compose the component permutations $A^{(i)}$ used in the permutation algorithms of [ST91, Ste92] into a single permutation.

We measure the *combinational complexity* of a permutation as the number of bit-operations required to generate the structure of the permutation matrix \mathbf{P} using a combinational circuit to compute the mapping defined by the characteristic matrix A . We shall show in Sections 3 and 4 how to directly translate the matrix A into a combinational circuit. We measure the complexity of performing the characteristic matrix mapping as the product of the input size N and the depth and width of the resulting circuit. Since we translate the characteristic matrix into XOR gates, the combinational complexity measures the total number of bit-operations required to perform the matrix-vector product Ax .

We specifically distinguish the combinational complexity from the sequential complexity of a permutation. The *sequential complexity* is the complexity of generating the structure of the permutation matrix using a sequential circuit which consists of combinational circuitry and one or more registers (clocked memory elements).

An easy way to use a sequential circuit to compute the N matrix-vector products $y^{(i)} = Ax^{(i)}$, for $i = 0, 1, \dots, N - 1$, is to compute them in Gray-code order, where $x^{(i)} = \text{gray}(i)$ computes the index in Gray-code order for a source index i . In Gray-code order, consecutive source vectors differ in only one bit. For consecutive source vectors, if we know that bit j of $x^{(i)}$ changed to form $x^{(i+1)}$, then we just add column j of the characteristic matrix A to the previous target vector $y^{(i)}$ to determine target vector $y^{(i+1)}$.

Since we can determine the next bit to change when mapping the source indices in Gray-code order in constant average time and we use $\lg N$ bit-operations to compute each target index, the sequential complexity to perform the N matrix-vector products is $O(N \lg N)$ bit operations.

The inverse of the permutation examined in Section 3 is one example in which the combinational complexity to perform the characteristic matrix-vector product is greater than the sequential complexity. We shall not further explore the sequential complexity of the permutations in the remainder of this paper. Therefore, when discussing the computational complexity of these permutations, we refer to the combinational complexity. The combinational circuit representation may be useful to designers of low-level implementations.

In Sections 3 and 4, we translate the permutation algorithms used in the FCT algorithms presented in [ST91] and [Ste92], respectively, into a single characteristic matrix form that generates the structure of the permutation matrix. In Section 3, we show that the permutation of [ST91], which we call the *recursive-complement*, or *RC, permutation*, requires a constant-depth, logarithmic-width circuit to compute its structure, and, therefore, has a complexity of $O(N \lg N)$ bit-operations. We also show that the inverse of the RC permutation requires a $\lg \lg N$ -depth, logarithmic-width circuit, resulting in a complexity of $O(N \lg N \lg \lg N)$ bit-operations. In Section 4, we show how to implement the permutation of [Ste92], which we call the *odd-upper/bit-reversal*, or *OUR, permutation*, with a constant-depth, logarithmic-width circuit which is *self-invertible*. That is, the inverse uses the same circuit and, therefore, the permutations for both the forward and inverse transforms use $O(N \lg N)$ bit-operations to compute the structure of the permutation matrix \mathbf{P} , just like the bit-reversal permutation used by the FFT algorithm.

3 Recursive-complement (RC) permutations

In this section, we describe an algorithm for performing the RC permutation used in the FCT algorithm presented in [ST91]. We translate the algorithm into a product of nonsingular matrices each of which characterizes a BMMC permutation with a special characteristic matrix form. We show that the product corresponds to the composition of the bit-reversal permutation with a Gray-code permutation.⁴ We use the properties of the characteristic matrix form for the RC permutation to design a constant-depth, logarithmic-width circuit which maps each $\lg N$ -bit source index to a $\lg N$ -bit target index. We also show that the inverse, however, maps to a logarithmic-depth, logarithmic-width circuit.

The algorithm to perform the RC permutation is as follows:

Algorithm RC-Permute(N)

```

1  J = N;
2  for i = 0, 1, ..., J - 1
3    MakeSet(i);
4  while J > 1
5    for i = 0, 1, ..., J/2 - 1
6      Join (i, J - 1 - i);
7    J = J/2;
```

Lines 2–3 create N singleton sets numbered from 0 to $N - 1$. Each singleton set contains an element with an index for a column of the identity matrix. Lines 5–6 combine each set with the

⁴This permutation is also the *tournament-pairing* permutation used for pairing teams in tournaments where the highest remaining seed plays the lowest remaining seed in each round, assuming the higher seeded teams win every game of each round.

set indexed by the bitwise complement of its index. Lines 5–6 cut the number of sets in half, assigning each new set the lower-numbered index of its component sets. We recurse until there remains one set with all N elements in their final order. Thus, we call this permutation the recursive complement (RC) permutation because, at each recursive step, we pair each set with the set indexed by the bitwise complement of its index. For example, if $N = 8$, on iteration 0, we pair set number $i = 3 = 011_2$ with set number $\bar{i} = N - 3 - 1 = 4 = 100_2$.

We reformulate **Algorithm RC-Permute** as a series of linear permutations. The input is an N -element set $\mathbf{x} = \langle x^{(0)}, x^{(1)}, \dots, x^{(N-1)} \rangle$ with the elements indexed from 0 to $N - 1$. The output is an N -element set $\mathbf{y} = \langle y^{(0)}, y^{(1)}, \dots, y^{(N-1)} \rangle$ that contains the permuted elements of the input set \mathbf{x} .

Algorithm RC-Permute($\mathbf{x}, \mathbf{y}, N$)

```

1  $J \leftarrow N$ 
2  $\mathbf{z} \leftarrow \{z^{(0)} = x^{(0)}\}, \{z^{(1)} = x^{(1)}\}, \dots, \{z^{(N-1)} = x^{(N-1)}\}$ 
3 while  $J > 1$ 
4    $\mathbf{z} \leftarrow \{z^{(0)}\}, \dots, \{z^{(J/2-1)}\}, \{z^{(J-1)}\}, \dots, \{z^{(J/2)}\}$ .
   (Reverse the order of the  $J/2$  elements with indices in the range  $J/2, J/2 + 1, \dots, J - 1$ .)
5    $\mathbf{z} \leftarrow \{z^{(0)}\}, \{z^{(J-1)}\}, \{z^{(1)}\}, \{z^{(J-2)}\}, \dots, \{z^{(J/2-1)}\}, \{z^{(J/2)}\}$ .
   (Perfect shuffle the first  $J/2$  sets with the second  $J/2$  sets.)
6    $\mathbf{z} \leftarrow \{z^{(0)}, z^{(J-1)}\}, \{z^{(1)}, z^{(J-2)}\}, \dots, \{z^{(J/2-1)}, z^{(J/2)}\}$ 
   (Group juxtapositioned sets in pairs.)
7    $J \leftarrow J/2$ 
8  $\mathbf{y} \leftarrow \mathbf{z}$ 

```

Line 2 creates N singleton sets. Lines 3–7 repeatedly perform two types of permutations on the J remaining sets. Line 4 performs what we call a *second-half reversal permutation*; that is, if we split the ordered sets into two halves, the second-half reversal permutation reverses the order of the $J/2$ highest-numbered sets. The permutation in line 5 performs a *perfect shuffle permutation* of the first $J/2$ sets with the second $J/2$ sets. Lines 6–7 combines every two consecutive sets into one set reducing the number of sets to $J/2$. The second-half reversal and perfect shuffle permutations recursively continue until there is only one set with N elements.

The composition of the second-half reversal and the perfect shuffle permutations is the same permutation as pairing the i th set with the \bar{i} th set. The second-half reversal permutation reverses the order of the second $J/2$ sets such that each set is in the same location within its half as the set indexed by its bitwise complement. The perfect shuffle permutation pairs each set with the set indexed by its bitwise complement to complete each iteration of the inner loop in **Algorithm RC-Permute**.

Second-half reversal permutations

We now translate the second-half reversal and perfect shuffle permutations into nonsingular characteristic matrix forms. The perfect shuffle is a well-known BMMC permutation and we shall see that the second-half reversal also belongs to the class of BMMC permutations. We first define characteristic matrix forms for these two types of permutations. Then we show that the product of the characteristic matrices of all the second-half reversals and perfect shuffles involved in an

RC permutation characterize the composition of the bit-reversal permutation with the Gray-code permutation.

Let us first define the $n \times n$ characteristic matrix form for the second-half reversal permutation of the initial N singleton sets as

$$R_0^{(n)} = \left(\begin{array}{c|c} & \begin{matrix} n-1 & 1 \end{matrix} \\ \hline & \begin{matrix} 1 \\ \vdots \\ 1 \end{matrix} \\ \hline I & \begin{matrix} 1 \\ \vdots \\ 1 \end{matrix} \\ \hline 0 & \cdots & 0 & | & 1 \end{array} \right) \begin{matrix} n-1 \\ \\ \\ 1 \end{matrix} .$$

In this matrix, we augment a leading $(n-1) \times (n-1)$ identity matrix with 1s in every position of the rightmost column. If we number the singleton sets from 0 to $N-1$, a set with a 0 as the most significant bit of its index is among the first $N/2$ sets. In this case, the rightmost column has no effect on the mapping, maintaining the first $N/2$ sets in their current locations. Otherwise, the most significant bit of the source index is 1 and therefore, the set is among the second $N/2$ sets. The rightmost column of 1s in the characteristic matrix has the effect of complementing the least significant $n-1$ bits of the source index for the set and thus, reverses the order of the second $N/2$ sets. Because there is only a single one in the rightmost location of the $(n-1)$ st row, all sets remain in the half in which they initially resided.

Before the k th iteration, $k = 0, 1, \dots, \lg N - 1$, of lines 3–7 of **Algorithm RC-Permute**, there are $N/2^k$ sets. The most significant $n-k$ bits of each source index represent the *set number* of the set that contains that element. The least significant k bits represent the index of an element within the set of 2^k elements to which it belongs. Because the order of the elements within a set does not change during the k th iteration, the least significant k bits do not change on that iteration.

On the k th iteration of line 4 in **Algorithm RC-Permute**, we perform a second-half reversal on N elements evenly split among $N/2^k$ sets. The second-half reversal reverses the ordering of the second $N/2^{k+1}$ sets. We denote the second-half reversal of N elements on the k th iteration as $R_k^{(n)}$. Thus, we characterize the k th second half reversal permutation with the $n \times n$ matrix

$$R_k^{(n)} = \left(\begin{array}{c|c} & \begin{matrix} k & n-k \end{matrix} \\ \hline & \begin{matrix} 0 & \cdots & 0 \end{matrix} \\ \hline 0 & \begin{matrix} R_0^{(n-k)} \end{matrix} \\ \hline \vdots & \\ \hline 0 & \end{array} \right) \begin{matrix} k \\ n-k \end{matrix} .$$

The leading $k \times k$ identity submatrix maintains the ordering of the elements within each set. The trailing $(n-k) \times (n-k)$ submatrix performs the same second-half reversal on the remaining $N/2^k$ sets as the second-half reversal permutation characterized by $R_0^{(n-k)}$. Thus, all characteristic matrices for second-half reversal permutations are unit upper triangular. Because all unit upper triangular matrices are nonsingular, second-half reversal permutations belong to the class of BMCC permutations.

Perfect shuffle permutations

We can easily define the perfect shuffle permutation in the same manner. The perfect shuffle permutation rotates the bits of the source index to the left by one bit. Therefore, we characterize the perfect shuffle of N sets by the $n \times n$ characteristic matrix

$$S_0^{(n)} = \left(\begin{array}{c|c} & \begin{matrix} n-1 & 1 \end{matrix} \\ \hline \begin{matrix} 0 & \cdots & 0 \end{matrix} & \begin{matrix} 1 \\ 0 \end{matrix} \\ \hline & \begin{matrix} \vdots \\ 0 \end{matrix} \\ \hline & \begin{matrix} n-1 \end{matrix} \end{array} \right) .$$

On the k th iteration of line 5 of **Algorithm RC-Permute**, we perform a perfect shuffle of the $N/2^k$ sets each of which contains 2^k elements. The perfect shuffle must not change the ordering of the elements within a set. Thus, we perform a perfect shuffle on the set numbers represented by the most significant $n - k$ bits of the source index. We characterize the perfect shuffle on the k th iteration by the $n \times n$ matrix

$$S_k^{(n)} = \left(\begin{array}{c|c} & \begin{matrix} k & n-k \end{matrix} \\ \hline \begin{matrix} I & 0 & \cdots & 0 \end{matrix} & \begin{matrix} k \\ n-k \end{matrix} \\ \hline \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} & \begin{matrix} S_0^{(n-k)} \end{matrix} \\ \hline & \end{array} \right) .$$

The leading $k \times k$ identity submatrix retains the ordering of the elements within each set. The trailing $(n - k) \times (n - k)$ submatrix performs the same perfect shuffle on the remaining $N/2^k$ sets as the perfect shuffle characterized by $S_0^{(n-k)}$ performs on the initial $N/2^k$ singleton sets on iteration 0. Because the characteristic matrix of a perfect shuffle on the k th iteration always has a single 1 in each row and column, it is nonsingular. Therefore, the perfect shuffle permutations are among the class of BMCC permutations.

Translating the RC permutation algorithm into a single characteristic matrix

We translate **Algorithm RC-Permute** into a product of second-half reversal and perfect shuffle permutations. For the k th iteration of the loop in lines 3–7 of **Algorithm RC-Permute**, we construct the appropriate matrices, $R_k^{(n)}$ and $S_k^{(n)}$, that characterize the second-half reversal and perfect shuffle permutations, respectively. Because we perform the second-half reversal first, by Lemma 1, the characteristic matrices $R_k^{(n)}$ are always to the right of their corresponding factors $S_k^{(n)}$.

We perform $\lg N$ iterations of the loop in lines 3–7 of **Algorithm RC-Permute**. The factors $R_{n-1}^{(n)}$ and $S_{n-1}^{(n)}$ are always equal to the identity matrix and we ignore them for the remainder of this section. Thus, the entire execution of the loop in **Algorithm RC-Permute** corresponds to performing the permutation characterized by the product

$$G^{(n)} = S_{n-2}^{(n)} R_{n-2}^{(n)} S_{n-3}^{(n)} R_{n-3}^{(n)} \cdots S_1^{(n)} R_1^{(n)} S_0^{(n)} R_0^{(n)} . \quad (5)$$

In the remainder of this proof, we assume that any submatrix denoted by a single 1 indicates that its row and column dimensions are exactly 1; all other dimensions have size $n - 1$. The product of the leftmost $n - 1$ pairs of perfect shuffles and second-half reversals in the matrix product $G^{(n+1)}$ is

$$\prod_{k=1}^{n-1} S_k^{(n+1)} R_k^{(n+1)} = \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & \prod_{k=0}^{n-2} S_k^{(n)} R_k^{(n)} & \\ 0 & & & \end{array} \right) = \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & G^{(n)} & \\ 0 & & & \end{array} \right)$$

This product multiplied by $S_0^{(n+1)}$ and $R_0^{(n+1)}$ gives us our final product

$$\begin{aligned} G^{(n+1)} &= \prod_{k=0}^{n-1} S_k^{(n+1)} R_k^{(n+1)} = \left(\prod_{k=1}^{n-1} S_k^{(n+1)} R_k^{(n+1)} \right) S_0^{(n+1)} R_0^{(n+1)} \\ &= \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & G^{(n)} & \\ 0 & & & \end{array} \right) \left(\begin{array}{ccc|c} 0 & \cdots & 0 & 1 \\ \hline & & & 0 \\ & & I & \vdots \\ & & & 0 \end{array} \right) \left(\begin{array}{c|ccc} & & & 1 \\ \hline & & I & \vdots \\ & & & 1 \\ 0 & \cdots & 0 & 1 \end{array} \right) \\ &= \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & G^{(n)} & \\ 0 & & & \end{array} \right) \left(\begin{array}{ccc|c} 0 & \cdots & 0 & 1 \\ \hline & & & 1 \\ & & I & \vdots \\ & & & 1 \end{array} \right) \\ &= \left(\begin{array}{ccc|c} 0 & \cdots & 0 & 1 \\ \hline & & & 1 \\ & & G^{(n)} & \vdots \\ & & & 0 \end{array} \right) . \quad \blacksquare \end{aligned}$$

We use the lower bidiagonal matrix $G^{(n)}$ to design a combinational circuit with constant depth and logarithmic width. Each row in the matrix $G^{(n)}$ determines one bit of the target index $y_i = \bigoplus_{j \in T_i} x_j$ where T_i is some set of source bits. Each element in T_i corresponds to a non-zero entry in row i . Since the matrix $G^{(n)}$ is reverse lower bidiagonal, each set T_i has no more than 2 elements. Thus, we use at most one XOR gate to compute each bit of the target index and a total of $\lg N - 1$ XOR gates to compute the entire target index. The target index has $\lg N$ bits, resulting in a circuit with logarithmic width. Since the circuit has $O(\lg N)$ width, constant depth, and the number of source indices is N , by our definition, the complexity of performing the permutation mapping to generate the structure of the permutation matrix \mathbf{P} is $O(N \lg N)$ bit-operations. An example circuit for $n = 8$ is shown in Figure 2.

The inverse of the RC permutation

Many applications of the FCT, such as data compression/decompression and convolutions on real, symmetric sequences, use both the forward and inverse transforms. Hence, we would like to

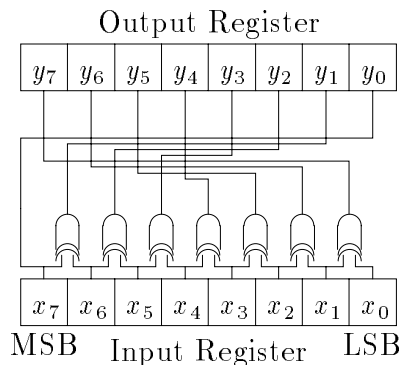


Figure 2: Circuit for mapping source indices for the RC permutation where $n = \lg N = 8$. This circuit has only one layer of 7 XOR gates. Thus, the circuit has constant depth and logarithmic width.

have an efficient method for generating the structure of the permutation matrix for the inverse transform, or equivalently, a simple index mapping circuit corresponding to the inverse of $G^{(n)}$.

We now show that the circuit corresponding to the inverse of the characteristic matrix $G^{(n)}$ has logarithmic depth.⁵ The $n \times n$ inverse of the matrix $G^{(n)}$ is

$$\text{inv}(G^{(n)}) = \begin{pmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & \cdots & 1 & \\ \vdots & \vdots & \ddots & & \\ 1 & 1 & & & \\ 1 & & & & \end{pmatrix}. \quad (7)$$

Although this characteristic matrix has $O(\lg^2 N)$ nonzero entries and a row has as many as $\lg N$ nonzero entries, we can use a circuit that computes partial sums to perform the index mapping of the inverse RC permutation. For each target index bit i , we compute the partial sum of the source index bits: $y_i = \bigoplus_{j=0}^{n-i-1} x_j$. This circuit has $\lg \lg N$ depth and $\lg N$ width. Figure 3 shows the circuit for the inverse RC permutation when $n = 8$. By our definition of complexity, the inverse RC permutation requires $O(N \lg N \lg \lg N)$ bit-operations.

Unfortunately, the inverse RC permutation has several undesirable properties. First, a (combinational) software algorithm to generate the inverse RC permutation mapping would require $O(N \lg N \lg \lg N)$ bit operations. Second, two different algorithms, or circuits, must be employed for the forward and inverse fast cosine transforms. Third, the complexity for the inverse RC permutation is greater than the complexity for the forward RC permutation. The next section shows that the permutation algorithm used in [Ste92] does not have these limitations.

4 Odd-upper/bit-reversal (OUR) permutations

In this section, we show that the algorithm for performing the permutation used by the FCT algorithm in [Ste92], which we call the OUR permutation, also maps to a constant-depth circuit.

⁵We denote the inverse of a matrix A by the function $\text{inv}(A)$.

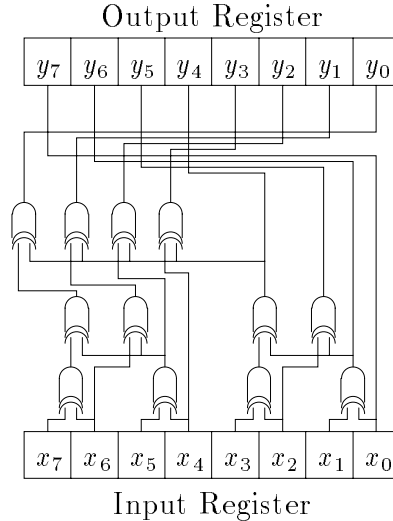


Figure 3: Circuit for mapping the inverse RC permutation where $n = \lg N = 8$. This circuit has depth $\lg \lg N = 3$ and logarithmic width.

The OUR permutation is a better choice than the RC permutation because it is self-invertible; that is, we use the same constant-depth circuit for computing the structure of the permutation matrices of both the forward and inverse fast cosine transforms.

The derivation of the *odd-upper/bit-reversal (OUR)* permutation arises naturally when we consider the computation of a fast cosine transform as a fast Fourier transform, or more generally as a fast discrete monomial transform [DHR94, MHR93]. It is shown in [MHR93] and elsewhere that for a real-valued sequence $\mathbf{f} = f(0), f(1), \dots, f(N-1)$ generated by evaluating a function at N points, we can compute the discrete cosine transform \hat{f}_{\cos} of \mathbf{f} as the real part of the discrete Fourier transform \hat{f}_{FT} of \mathbf{f} , i.e.,

$$\hat{f}_{\cos} = \Re(\hat{f}_{FT})$$

where $\Re(z)$ denotes the real part of a complex number z . This fact follows from

$$\Re(e^{im\theta}) = \cos(m\theta), \tag{8}$$

or equivalently $\cos(m\theta) = \frac{e^{im\theta} + e^{-im\theta}}{2}$. Although the DCT is a real-valued transform, we can use an FFT algorithm to compute the DCT in $O(N \lg N)$ operations if we employ an appropriate mapping from points in the real interval $[-1, 1]$ to the unit circle S^1 .

To define such a mapping, we first consider the two requirements necessary to achieve an $O(N \lg N)$ FFT algorithm: uniform sampling and application of the bit-reversal permutation (or a closely related permutation) to the data sequence. The standard FFT algorithm computes the N -point discrete Fourier transform of a sequence $\mathbf{f} = f(0), f(1), \dots, f(N-1)$ in $O(N \lg N)$ operations under the assumption that we sample the underlying function at the N th roots of unity or at a coset of the N th roots of unity. The set of N th roots of unity is the set of the N uniformly-spaced points on the unit circle S^1 , including the identity. The standard FFT algorithm in matrix form applies the bit-reversal permutation matrix and a series of sparse matrices to the

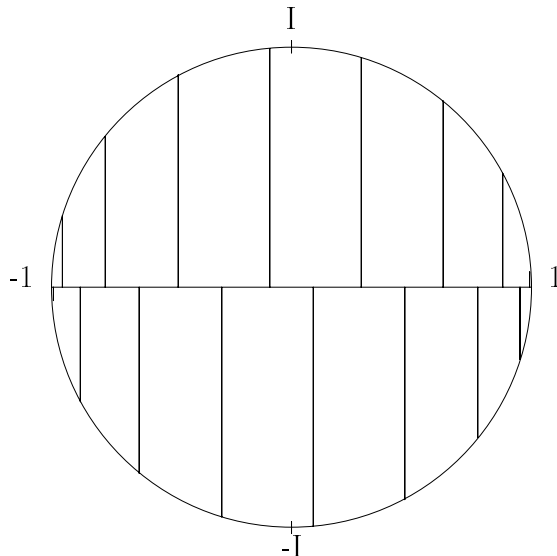


Figure 4: Mapping of $\cos(\frac{\pi(2k+1)}{2N})$, $k = 0, 1, \dots, N - 1$ to the unit circle for $N = 16$. The points on the unit circle are uniformly spaced, separated by $\frac{\pi}{8}$ radians.

sequence \mathbf{f} in $O(N \lg N)$ operations. If the evaluation points on the unit circle are not uniformly spaced or we do not apply an appropriate permutation matrix, then the fast Fourier transform uses $O(N \lg^2 N)$ operations [MHR93]. If we wish to derive an $O(N \lg N)$ DCT algorithm using the FFT algorithm and the identity of equation (8), then we must sample the underlying function at the points $\cos(\theta_k)$ on the interval $[-1, 1]$ that are uniformly spaced in θ , which is not the same as uniform sampling on $[-1, 1]$. Standard fast cosine transform algorithms perform this type of sampling, usually at the points $\cos(\frac{\pi(2k+1)}{2N})$, where $k = 0, 1, \dots, N - 1$.

If the sequence \mathbf{f} represents samples of a function evaluated at the sequence of points $\mathbf{x} = x^{(0)}, x^{(1)}, \dots, x^{(N-1)}$ such that $x^{(k)} = \cos(\frac{\pi(2k+1)}{2N})$, for $k = 0, \dots, N - 1$, then we map the sequence \mathbf{x} to the sequence ω by the function $U : [-1, 1] \times \mathbb{N} \rightarrow S^1$ such that

$$U(x, k) = e^{(-1)^k i \cos^{-1} x^{(k)}} = e^{(-1)^k i \frac{\pi(2k+1)}{2N}} = \omega^{(k)} .$$

If we sort the points $\omega^{(k)}$ into angular order, then the resulting sequence has the property that consecutive points are separated by exactly $\frac{2\pi}{N}$ radians, which is a uniform sampling of points on the unit circle. We illustrate an example of this mapping in Figure 4.

We now derive a permutation π on \mathbf{x} such that after performing the mapping U on the sequence $\pi(\mathbf{x})$, the resulting sequence $U(\pi(\mathbf{x}))$ is sorted by increasing angular order. Note that if we apply the function U to every element in \mathbf{x} , then the resulting sequence $\omega = \omega^{(0)}, \omega^{(1)}, \dots, \omega^{(N-1)}$ is not sorted by increasing angular order. To permute the sequence \mathbf{x} such that the sequence $U(\pi(\mathbf{x}))$ is sorted by increasing angular order, we first partition \mathbf{x} into its even-indexed and odd-indexed elements. We then map the even-indexed elements into the first $N/2$ locations and the odd-indexed elements into the last $N/2$ locations. Additionally, we reverse the order of the odd-indexed

elements. Thus, for the N -point sequence \mathbf{x} , the permutation is

$$\pi(\mathbf{x}) = x^{(0)}, x^{(2)}, \dots, x^{(N-2)}, x^{(N-1)}, x^{(N-3)}, \dots, x^{(3)}, x^{(1)} .$$

If we compose the bit-reversal permutation with $\pi(\mathbf{x})^6$ and apply U , [Moo94] shows that the ordering of the elements on the unit circle produces the sparse matrices needed to compute a fast discrete Fourier transform and its associated discrete cosine transform using $O(N \lg N)$ operations.

An example for $N = 8$ may be helpful. We start with

$$\mathbf{x} = \cos\left(\frac{\pi}{16}\right), \cos\left(\frac{3\pi}{16}\right), \cos\left(\frac{5\pi}{16}\right), \cos\left(\frac{7\pi}{16}\right), \cos\left(\frac{9\pi}{16}\right), \cos\left(\frac{11\pi}{16}\right), \cos\left(\frac{13\pi}{16}\right), \cos\left(\frac{15\pi}{16}\right).$$

Partitioning into even- and odd-indexed elements and mapping even-indexed elements to locations 0–3 and odd-indexed elements to locations 4–7 results in the sequence

$$\cos\left(\frac{\pi}{16}\right), \cos\left(\frac{5\pi}{16}\right), \cos\left(\frac{9\pi}{16}\right), \cos\left(\frac{13\pi}{16}\right), \cos\left(\frac{3\pi}{16}\right), \cos\left(\frac{7\pi}{16}\right), \cos\left(\frac{11\pi}{16}\right), \cos\left(\frac{15\pi}{16}\right).$$

Reversing the last 4 elements of the sequence gives

$$\cos\left(\frac{\pi}{16}\right), \cos\left(\frac{5\pi}{16}\right), \cos\left(\frac{9\pi}{16}\right), \cos\left(\frac{13\pi}{16}\right), \cos\left(\frac{15\pi}{16}\right), \cos\left(\frac{11\pi}{16}\right), \cos\left(\frac{7\pi}{16}\right), \cos\left(\frac{3\pi}{16}\right),$$

effecting the permutation π defined above. If one were to apply the mapping U at this point, the resulting sequence of points on the unit circle would be sorted in angular order. Applying a bit-reversal permutation produces

$$\cos\left(\frac{\pi}{16}\right), \cos\left(\frac{15\pi}{16}\right), \cos\left(\frac{9\pi}{16}\right), \cos\left(\frac{7\pi}{16}\right), \cos\left(\frac{5\pi}{16}\right), \cos\left(\frac{11\pi}{16}\right), \cos\left(\frac{13\pi}{16}\right), \cos\left(\frac{3\pi}{16}\right).$$

Mapping to the unit circle using $U(x, k)$ gives the sequence

$$e^{\frac{i\pi}{16}}, e^{-\frac{i15\pi}{16}}, e^{\frac{i9\pi}{16}}, e^{-\frac{i7\pi}{16}}, e^{\frac{i5\pi}{16}}, e^{-\frac{i11\pi}{16}}, e^{\frac{i13\pi}{16}}, e^{-\frac{i3\pi}{16}},$$

which is a coset of the set of 8th roots of unity in bit-reversal order.

Creating a matrix product for the OUR permutation

We now translate the permutation algorithm for π into characteristic matrix factors. Then we compose the characteristic matrix factors with a factor that characterizes the bit-reversal permutation and show that the resulting characteristic matrix maps to a constant-depth, self-invertible circuit.

The following algorithm describes more formally the mapping π of the N -element sequence $\mathbf{x} = \langle x^{(0)}, x^{(1)}, \dots, x^{(N-1)} \rangle$ to the N -element sequence $\mathbf{y} = \pi(\mathbf{x})$:

Algorithm π -map(\mathbf{x} , \mathbf{y} , N)

```

1  for  $j = 0$  to  $N - 1$ 
2    if  $(j \bmod 2 \equiv 1)$  then
3       $y^{(N-(\lfloor j/2 \rfloor + 1))} = x^{(j)}$ 
4    else
5       $y^{(j/2)} = x^{(j)}$ 

```

⁶Thus, we complete the derivation of the OUR acronym in the description “if $x^{(0)}$ is Odd, then complement the Upper bits; bit-Reverse”.

We reformulate **Algorithm π -map** into the composition of the inverse of a perfect shuffle and a second-half reversal. Then we multiply the matrix product corresponding to **Algorithm π -map** with the characteristic matrix form corresponding to the bit-reversal permutation. The product of the three matrices results in the characteristic matrix form for the OUR permutation.

Algorithm π -map($\mathbf{x}, \mathbf{y}, N$)

- 1 $\mathbf{z} = \langle x^{(0)}, x^{(2)}, \dots, x^{(N-2)}, x^{(1)}, x^{(3)}, \dots, x^{(N-1)} \rangle$
(Perform the inverse perfect shuffle.)
- 2 $\mathbf{z} = \langle x^{(0)}, x^{(2)}, \dots, x^{(N-2)}, x^{(N-1)}, x^{(N-3)}, \dots, x^{(3)}, x^{(1)} \rangle$
(Perform a second-half reversal.)
- 3 $\mathbf{y} = \mathbf{z}$

The characteristic matrix form for the inverse of a perfect shuffle permutation is

$$\text{inv}(S_0^{(n)}) = \begin{pmatrix} 1 & n-1 \\ \hline 0 & I \\ \hline 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \end{pmatrix} \begin{matrix} n-1 \\ \\ \\ 1 \end{matrix} .$$

This characteristic matrix form represents a right cyclic rotation of the source index bits by one bit. That is, any source index with a 0 in its least significant bit (even indices) maps to one of the lower $N/2$ locations, and any source index with a 1 in its least significant bit (odd indices) maps to one of the upper $N/2$ locations. The order among the even- and odd-indexed elements remains the same.

Thus, the characteristic matrix form corresponding to **Algorithm π -map** is the matrix product

$$Q^{(n)} = R_0^{(n)} \text{inv}(S_0^{(n)}) = \begin{pmatrix} n-1 & 1 & 1 & n-1 \\ \hline I & \vdots & 1 \\ \hline 0 & \dots & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & n-1 \\ \hline \vdots & I \\ \hline 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \end{pmatrix} = \begin{pmatrix} 1 & n-1 \\ \hline \vdots & I \\ \hline 1 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \end{pmatrix} \begin{matrix} n-1 \\ \\ \\ 1 \end{matrix} .$$

To complete the derivation of the final characteristic matrix form for the OUR permutation, we multiply the matrix product $Q^{(n)}$ on the left by the matrix form $B^{(n)}$ that characterizes a bit-reversal permutation of N elements resulting in the matrix product

$$H^{(n)} = B^{(n)} Q^{(n)} = \begin{pmatrix} n & 1 & n-1 \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & n-1 \\ \hline \vdots & I \\ \hline 1 & 0 & \dots & 0 \end{pmatrix} = \begin{pmatrix} 1 & n-1 \\ \hline 1 & 0 & \dots & 0 \\ \hline \vdots & 0 & \dots & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \begin{matrix} 1 \\ \\ n-1 \\ \end{matrix} .$$

5 Conclusion

We have translated the two permutations used by the fast cosine transforms of [ST91, Ste92] into a product of familiar linear index transformation matrices. Both of these products result in linear index transformation matrices which we show how to implement using constant-depth, logarithmic-width circuits. By our definition, both permutations have a complexity of $O(N \lg N)$ bit-operations. We have also shown that the permutation of [Ste92] is a better permutation choice because it is self-invertible, which allows the same constant-depth circuit to be used for both the forward and the inverse FCT algorithms.

We ask whether other applications use permutations that are in the class of BMMC permutations. If so, we could apply the techniques of the linear transformation analysis in this paper to determine their complexity. This method of analysis may allow us derive more permutations with properties, such as self-invertibility, that make them superior to the permutations in common use.

References

- [BM75] A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems*. Elsevier, New York, 1975.
- [Chi95] P. Chintrakulchai. *Speeding Up Fractal Image Compression*. PhD thesis, Department of Computer Science, Dartmouth College, July 1995.
- [CSW94] T.H. Cormen, T. Sundquist, and L.F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report Technical Report PCS-TR94-223, Dartmouth College Department of Computer Science, July 1994. Preliminary version appeared in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*.
- [DHR94] J.R. Driscoll, D. Healy, and D. Rockmore. Fast spherical transforms for distance transitive graphs. Technical Report Technical Report PCS-TR94-223, Dartmouth College Department of Mathematics and Computer Science, 1994.
- [EHJ94] A. Edelman, S. Heller, and S.L. Johnson. Index transformation algorithms in a linear algebra framework. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1302–1309, December 1994.
- [ER82] D.F. Elliot and K.R. Rao. *Fast Transforms: Algorithms, Analyses, Applications*. Academic Press, Orlando, 1982.
- [Knu81] D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1981.
- [MHR93] S.S.B. Moore, D.M. Healy, and D.N. Rockmore. Symmetry stabilization for fast discrete monomial transforms and polynomial evaluation. *Linear Algebra and its Applications - Special Issue on Computational Linear Algebra in Algebraic and Related Problems*, 192:249–299, October 1993.

- [Moo94] S. S. B. Moore. *Efficient Stabilization Methods for Fast Polynomial Transforms*. PhD thesis, Department of Mathematics and Computer Science, Dartmouth College, June 1994.
- [RY90] K.R. Rao and P. Yip. *Discrete Cosine Transforms: Algorithms, Advantages, Applications*. Academic Press, San Diego, 1990.
- [ST91] G. Steidl and M. Tasche. A polynomial approach to fast algorithms for discrete Fourier-cosine and Fourier-sine transforms. *Mathematics of Computation*, 56(193):281–296, 1991.
- [Ste92] G. Steidl. Fast radix- p discrete cosine transform. In *Applicable Algebra in Engineering, Communication and Computing*, pages 39–46. Springer-Verlag, 1992.