

Tensor Product Formulation for Hilbert Space-Filling Curves*

Shen-Yi Lin¹, Chih-Shen Chen¹, Li Liu², and Chua-Huang Huang¹

¹Department of Information Engineering and Computer Science
Feng Chia University
Taichung, Taiwan, R.O.C.

²Graduate Institute of Medical Informatics
Taipei Medical University
Taipei, Taiwan, R.O.C.

Abstract

We present a tensor product formulation for Hilbert space-filling curves. Both recursive and iterative formulas are expressed in the paper. We view a Hilbert space-filling curve as a permutation which maps two-dimensional $2^n \times 2^n$ data elements stored in the row major or column major order to the order of traversing a Hilbert curve. The tensor product formula of Hilbert space-filling curves uses several permutation operations: stride permutation, radix-2 Gray permutation, transposition, and anti-diagonal transposition. The iterative tensor product formula can be manipulated to obtain the inverse Hilbert permutation. Also, the formulas are directly translated into computer programs which can be used in various applications including image processing, VLSI component layout, and R-tree indexing, *etc.*

Keywords: tensor product, block recursive algorithm, Hilbert space-filling curve, stride permutation, Gray permutation, transposition, anti-diagonal transposition, data allocation.

1 Introduction

In 1890, G. Peano presented a space-filling curve of traversing points on a two-dimensional $3^n \times 3^n$ square grid exactly once and without crossing the path [27]. In 1891, D. Hilbert also presented a way of traversing two-dimensional $2^n \times 2^n$ space-filling curves [9]. In that paper, the Hilbert space-filling curve is viewed as an ordering function of $2^n \times 2^n$ points in the one-dimensional space. The order can be used to arrange data elements in various applications such as image pixel allocation [1, 22, 23],

VLSI component layout [26, 29], and R-tree indexing [6, 17, 16, 20, 21] to increase locality efficiency. If the data elements on a $2^n \times 2^n$ grid are initially arranged in the row (or column) major order, the ordering function of the Hilbert space-filling curve is exactly a permutation function performing data reallocation.

In this paper, we use the tensor product (also known as Kronecker product) notation [7] to formulate the permutation function of the Hilbert space-filling curve. The tensor product notation has been used to design and implement block recursive algorithms such as fast Fourier transform [12, 13], Strassen's matrix multiplication [10, 19], and parallel prefix algorithms [5].

Tensor product formulas can be directly translated to computer programs. For different architecture characteristics, such as vector processors, parallel multiprocessors, and distributed-memory multiprocessors, tensor product formulas can be manipulated using appropriate algebraic theorems and then translated to high-performance programs [4, 8]. Tensor product formulas can also be used to specify data allocation and generate efficient programs for multi-level memory hierarchy including cache memory, local memory, and external memory [18].

The Hilbert space-filling curve is viewed as a permutation function of a 2×2 block recursive structure. We express the Hilbert permutation as an algebraic formula consisting of tensor product, direct sum, and matrix product operations and some specific permutations: stride permutation, radix-2 Gray permutation, transposition, and anti-diagonal transposition. These operations and permutations can be mapped to constructs of high-level programming languages. Hence, the Hilbert space-filling curve formula can be easily translated into a computer program. The program rearranges data elements of a two-dimensional matrix from the row (or column) major order to the Hilbert space-filling curve order. Furthermore, the inverse function of the Hilbert space-filling curve formula

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

will rearrange the data elements of two-dimensional matrix from the Hilbert space-filling curve order to the row (or column) major order. This is important as employing the Hilbert permutation to compress and decompress image files.

The paper is organized as the following. Related works of the Hilbert space-filling curve is given in Section 2. In Section 3, we briefly explain the algebraic theory of tensor product and other related operations. Tensor product formulation of the Hilbert space-filling curve, both recursive form and iterative form, is derived in Section 4. We explain the derivation of the recursive formula of the Hilbert permutation step by step. Also, we prove the correctness of the iterative formula. In addition, The iterative tensor product formula of the inverse permutation of the Hilbert space-filling curve is described in Section 4. Program generation of the Hilbert space-filling curve from its tensor product formulas is explained in Section 5. We generate the program manually, although it can be done mechanically [4, 25]. Concluding remarks and future works are given in Section 6.

2 Related Works

Since D. Hilbert presented the Hilbert space-filling curve in 1981, there have been several research works about how to formally specify it using either an operational model or a functional model. Hilbert space-filling curve has been viewed as a one-to-one mapping function by Butz [2, 3]. He proposed an algorithm to compute the mapping function with bit operations. Jagadish had analyzed the clustering properties of Hilbert space-filling curve [11]. He showed that Hilbert space-filling curve achieves the best clustering, i.e., it is the best space-filling curve in minimizing the number of clusters. Moon *et al.* provided closed-form formulas of the number of clusters required by a given query region of an arbitrary shape for Hilbert space-filling curve [24]. Quinqueton and Berthod proposed an algorithm for computing all addresses of scanning path by recursive procedure [28]. Kamata *et al.* proposed a nonrecursive algorithm for N -dimensional Hilbert space-filling curve using look-up tables [14, 15]. A mathematical history of the Hilbert space-filling curves was presented by Sagan [30].

3 Overview of Tensor Product Operations

In this session, we give an overview of the algebraic operations and some of their properties used in formulating the

Hilbert space-filling curve. The operations explained include tensor product, direct sum, and stride permutation.

Definition 3.1 (Tensor Product of Matrices) *Let A and B be two matrices of size $m \times n$ and $p \times q$, respectively. The tensor product of A and B is the block matrix obtained by replacing each element $a_{i,j}$ by $a_{i,j}B$, i.e., $A \otimes B$ is an $mp \times nq$ matrix defined as*

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

For example, if

$$A = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}, B = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}, \text{ then}$$

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 2 & 4 & 6 \end{bmatrix} \otimes \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \\ &= \begin{bmatrix} 6 & 2 & 12 & 4 & 18 & 6 \\ 4 & 8 & 8 & 16 & 12 & 24 \end{bmatrix}. \end{aligned}$$

Let F^m be the vector space of m -tuples over the field F and let $F^{m \times n}$ be the vector space of $m \times n$ matrices. The collection of elements $\{e_i^m | 0 \leq i < m\}$, where e_i^m is the vector with a one in the i -th position and zeros elsewhere, form the standard basis for F^m .

Definition 3.2 (Tensor Basis) *Let F^n be the vector space of n -tuples over the field F , a collection of elements $\{e_{i_1}^{n_1} \otimes e_{i_2}^{n_2} \otimes \cdots \otimes e_{i_k}^{n_k} | 0 \leq i_1 < n_1, 0 \leq i_2 < n_2, \dots, 0 \leq i_k < n_k\}$, is a tensor basis of $F^{n_1} \otimes F^{n_2} \otimes \cdots \otimes F^{n_k}$.*

Tensor basis can be linearized (or factorized) as below:

$$\begin{aligned} e_i^m \otimes e_j^n &= e_{in+j}^{mn} \\ e_{i_1}^{n_1} \otimes e_{i_2}^{n_2} \otimes \cdots \otimes e_{i_k}^{n_k} &= e_{i_1 n_2 \cdots n_k + \cdots + i_{k-1} n_k + i_k}^{n_1 n_2 \cdots n_k} \end{aligned}$$

Definition 3.3 (Direct Sum) *Let A and B be two matrices $m \times n$ and $p \times q$, respectively. The direct sum of A and B is an $(m+p) \times (n+q)$ matrix defined as*

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

For example, if

$$A = \begin{bmatrix} 1 & 3 & 5 \end{bmatrix}, B = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}, \text{ then}$$

$$\begin{aligned}
A \oplus B &= [1 \ 3 \ 5] \oplus \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 3 & 5 & 0 & 0 \\ 0 & 0 & 0 & 2 & 4 \\ 0 & 0 & 0 & 6 & 8 \end{bmatrix}.
\end{aligned}$$

If B is a $p \times q$ matrix, $I_n \otimes B$ is the direct sum of n copies of B , where I_n is the $n \times n$ identity matrix.

$$I_n \otimes B = \bigoplus_{k=0}^{n-1} B = \begin{bmatrix} B & & \\ & \ddots & \\ & & B \end{bmatrix}.$$

Definition 3.4 (Stride Permutation) A stride permutation $L_n^{mn}(e_i^m \otimes e_j^n)$ is defined as:

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

$L_n^{mn}(e_i^m \otimes e_j^n)$ is referred to as the stride permutation which permutes the tensor product of two vector bases. If an $m \times n$ matrix is stored in the column major order, its basis is isomorphic to $e_i^m \otimes e_j^n$. Stride permutation is exactly the transposition operation transforming the matrix from the column major ordering allocation to the row major ordering.

The followings are some properties of tensor products, direct sums, and stride permutations used in this paper. We omit the proofs of these properties. Note that, I_n is the $n \times n$ identity matrix, $\prod_{i=l}^u A_i$ is in the order from the left-hand-side to the right-hand-side as $A_u A_{u-1} \cdots A_{l+1} A_l$, and all matrix products $A_i B_i$ are legally defined.

1. $A \otimes B \otimes C = (A \otimes B) \otimes C = A \otimes (B \otimes C)$,
 $A \oplus B \oplus C = (A \oplus B) \oplus C = A \oplus (B \oplus C)$.
2. $(A_1 \otimes A_2 \otimes \cdots \otimes A_k)(B_1 \otimes B_2 \otimes \cdots \otimes B_k)$
 $= (A_1 B_1 \otimes A_2 B_2 \otimes \cdots \otimes A_k B_k)$,
 $(A_1 \oplus A_2 \oplus \cdots \oplus A_k)(B_1 \oplus B_2 \oplus \cdots \oplus B_k)$
 $= (A_1 B_1 \oplus A_2 B_2 \oplus \cdots \oplus A_k B_k)$.
3. $(A_1 \otimes B_1)(A_2 \otimes B_2) \cdots (A_k \otimes B_k)$
 $= (A_1 A_2 \cdots A_k) \otimes (B_1 B_2 \cdots B_k)$,
 $(A_1 \oplus B_1)(A_2 \oplus B_2) \cdots (A_k \oplus B_k)$
 $= (A_1 A_2 \cdots A_k) \oplus (B_1 B_2 \cdots B_k)$.
4. $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$,
 $(A \oplus B)^{-1} = A^{-1} \oplus B^{-1}$.
5. $\prod_{i=0}^{n-1} (I_n \otimes A_i) = I_n \otimes \left(\prod_{i=0}^{n-1} A_i \right)$.
6. $\prod_{i=0}^{n-1} (A_i \otimes I_n) = \left(\prod_{i=0}^{n-1} A_i \right) \otimes I_n$.
7. $(L_n^{mn})^{-1} = L_m^{mn}$, $L_n^n = I_n$.
8. $L_{rs}^{rst} = L_r^{rst} L_s^{rst}$.
9. $L_t^{rst} = (L_t^r \otimes I_s)(I_r \otimes L_t^{st})$.
10. $L_{st}^{rst} = (I_s \otimes L_t^r)(L_s^{rs} \otimes I_t)$.

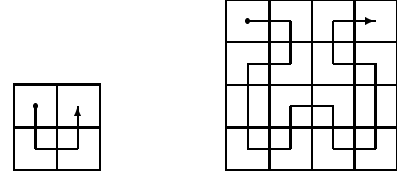


Figure 1: 2×2 and 4×4 Hilbert space-filling curves

4 Tensor Product Formulation for Hilbert Space-Filling Curves

The Hilbert space-filling curve visits all points on a continuous plane if we iterate the curve formation algorithm to the infinite. For the finite cases, the Hilbert space-filling curve is a curve on a $2^n \times 2^n$ grid and visits consecutive neighboring points without crossing the curve. Typically, a $2^n \times 2^n$ Hilbert space-filling curve is recursively constructed from $2^{n-1} \times 2^{n-1}$ Hilbert space-filling curves. For example, a 2×2 , H_1 , and a 4×4 , H_2 , Hilbert space-filling curve are shown in Figure 1. H_2 is a curve connecting four copies of H_1 in different orientations.

Suppose the points of a grid represent some data on a plane such as image pixels, VLSI components, or geometric information, *etc.* The collection of data elements must be stored in computer memory according to a given order, usually, the column major or the row major order. These location orders are natural, but they are lack of locality efficiency. If the two-dimensional data elements are stored in the Hilbert space-filling curve order, it may improve locality access and spatial structure. In this paper, we will revisit the Hilbert space-filling curve ordering problem to rearrange computer data in the column major or row major order into the Hilbert space-filling curve order. The reordering algorithm will be expressed in the tensor product notation.

4.1 Recursive Tensor Product Formulation

Suppose the points of a $2^n \times 2^n$ grid are initially stored in the column major order. For instance, the initial allocation of 8×8 points is stored in a linear order as indexed in Figure 2(a). The construction of the $2^n \times 2^n$ Hilbert space-filling curve is carried out in the following four steps:

1. Reallocate the initial column major ordering data to 2×2 blocks with each block of $2^{n-1} \times 2^{n-1}$ points. Both the blocks and the points in each block are stored in the column major order. The result of block reallocation for the 8×8 grid is shown in Figure 2(b).
2. Permute the blocks using 2×2 Gray permutation.

This permutation reorders index sequence $(0, 1, 2, 3)$ to $(0, 1, 3, 2)$ so that the Hamming distance of two adjacent indices is 1. The result of permuting the 2×2 blocks is shown in Figure 2(c). Note that, after Gray permutation, the 2×2 blocks are in the Hilbert space-filling curve order.

3. For each block, rotate and reflect the block elements according to a given orientation. As shown in Figure 1, only the upper-left and the upper-right blocks in H_2 change their orientation. The lower-left and the low-right blocks have the same orientation as H_1 . The upper-left block is rotated 90 degrees counter-clockwise along the center of the block and mirror reflected by the horizontal line through the center of the block. The upper-right block is rotated 90 degrees clockwise along the center of the block and mirror reflected by the horizontal line through the center of the block. Mathematically, rotation and reflection applying to the upper-left and upper-right blocks is exactly the transposition operation and anti-diagonal transposition operation, respectively. We show the result after rotation and reflection of the upper-left and upper-right 4×4 blocks in Figure 2(d).
4. Finally, recursively apply 4×4 Hilbert space-filling curve permutation to each of the four blocks. The recursive permutation is applied to the points in each of the 4×4 blocks indexed in a linear order resulting from the previous steps. In addition, the base case of the recursion is the 2×2 Hilbert space-filling curve H_1 in Figure 1. The final index order of 8×8 Hilbert space-filling curve is shown in Figure 2(e).

Figure 2(f) presents the 8×8 Hilbert space-filling curve by connecting the points following the sequence of indices in Figure 2(e).

We will explain the recursive tensor product formulation of the Hilbert space-filling curve following the steps in the above example. Initially, we assume the points (data) on a two-dimensional $2^n \times 2^n$ grid are stored in the column major order. That is, the index of point (i, j) , $0 \leq i, j < 2^n$, is described by tensor basis $e_i^{2^n} \otimes e_j^{2^n}$.

1. The block reallocation is defined as $B_n = I_2 \otimes L_2^{2^n} \otimes I_{2^{n-1}}$. Applying B_n to the initial column major ordering tensor basis, we obtain the following basis:

$$\begin{aligned} & (I_2 \otimes L_2^{2^n} \otimes I_{2^{n-1}})(e_i^{2^n} \otimes e_j^{2^n}) \\ &= (I_2 \otimes L_2^{2^n} \otimes I_{2^{n-1}})(e_{i_0}^2 \otimes e_{i_1}^{2^{n-1}} \otimes e_{j_0}^2 \otimes e_{j_1}^{2^{n-1}}) \\ &= e_{i_0}^2 \otimes e_{j_0}^2 \otimes e_{i_1}^{2^{n-1}} \otimes e_{j_1}^{2^{n-1}} \end{aligned}$$

where $i = i_0 2^{n-1} + i_1$ and $j = j_0 2^{n-1} + j_1$.

2. The Gray permutation for 2×2 grid is the mapping of $(0, 1, 2, 3)$ to $(0, 1, 3, 2)$. We define this permutation

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

(a)

0	4	8	12	32	36	40	44
1	5	9	13	33	37	41	45
2	6	10	14	34	38	42	46
3	7	11	15	35	39	43	47
16	20	24	28	48	52	56	60
17	21	25	29	49	53	57	61
18	22	26	30	50	54	58	62
19	23	27	31	51	55	59	63

(b)

0	4	8	12	48	52	56	60
1	5	9	13	49	53	57	61
2	6	10	14	50	54	58	62
3	7	11	15	51	55	59	63
16	20	24	28	32	36	40	44
17	21	25	29	33	37	41	45
18	22	26	30	34	38	42	46
19	23	27	31	35	39	43	47

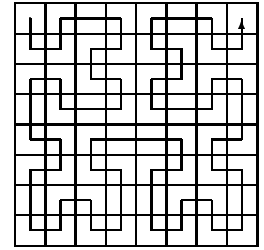
(c)

0	1	2	3	63	62	61	60
4	5	6	7	59	58	57	56
8	9	10	11	55	54	53	52
12	13	14	15	51	50	49	48
16	20	24	28	32	36	40	44
17	21	25	29	33	37	41	45
18	22	26	30	34	38	42	46
19	23	27	31	35	39	43	47

(d)

0	3	4	5	58	59	60	63
1	2	7	6	57	56	61	62
14	13	8	9	54	55	50	49
15	12	11	10	53	52	51	48
16	17	30	31	32	33	46	47
19	18	29	28	35	34	45	44
20	23	24	27	36	39	40	43
21	22	25	26	37	38	41	42

(e)



(f)

Figure 2: Construction of 8×8 Hilbert space-filling curves.

as $G_2 = I_2 \oplus J_2$, where

$$J_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \text{ and then } G_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The Gray permutation of blocks is specified as $G_n = G_2 \otimes I_{2^{2(n-1)}}$. Applying G_n to the resulting tensor basis of B_n , we obtain:

$$\begin{aligned} & (G_2 \otimes I_{2^{2(n-1)}})(e_{i_0}^2 \otimes e_{j_0}^2 \otimes e_{i_1}^{2^{n-1}} \otimes e_{j_1}^{2^{n-1}}) \\ &= e_{i_0}^2 \otimes e_{j_0}^2 \otimes e_{i_1}^{2^{n-1}} \otimes e_{j_1}^{2^{n-1}} \end{aligned}$$

where (i_0, j_0) is mapped to (i_0', j_0') in the following way: $(0, 0) \rightarrow (0, 0)$, $(0, 1) \rightarrow (0, 1)$, $(1, 0) \rightarrow (1, 1)$, and $(1, 1) \rightarrow (1, 0)$.

3. We use T_{n-1} and \bar{T}_{n-1} to denote transposition and anti-diagonal transposition operation of $2^{n-1} \times 2^{n-1}$ blocks, respectively. The rotation and reflection operation is expressed as $R_n = T_{n-1} \oplus I_{2^{2(n-1)}} \oplus I_{2^{2(n-1)}} \oplus \bar{T}_{n-1}$. The effect of T_{n-1} and \bar{T}_{n-1} on tensor basis $e_{i_1}^{2^{n-1}} \otimes e_{j_1}^{2^{n-1}}$ is as below:

$$\begin{aligned} & T_{n-1}(e_{i_1}^{2^{n-1}} \otimes e_{j_1}^{2^{n-1}}) = e_{j_1}^{2^{n-1}} \otimes e_{i_1}^{2^{n-1}}, \\ & \bar{T}_{n-1}(e_{i_1}^{2^{n-1}} \otimes e_{j_1}^{2^{n-1}}) = e_{2^{n-1}-1-j_1}^{2^{n-1}} \otimes e_{2^{n-1}-1-i_1}^{2^{n-1}} \end{aligned}$$

Note that, the transposition operation T_{n-1} is exactly the stride permutation $L_{2^{n-1}}^{2^{2(n-1)}}$.

- Finally, the recursive application of Hilbert space-filling curve permutation to the four resulting blocks is expressed as $I_4 \otimes H_{n-1}$.

We summarize the recursive tensor product formula of Hilbert space-filling curve permutation of $2^n \times 2^n$ grid below:

Definition 4.1 (Recursive Tensor Product Formula)

$$\begin{aligned} H_1 &= G_2. \\ n > 1 : \\ H_n &= (I_4 \otimes H_{n-1})R_nG_nB_n \\ &= (I_4 \otimes H_{n-1}) \\ &\quad (T_{n-1} \oplus I_{2^{2(n-1)}} \oplus I_{2^{2(n-1)}} \oplus \overline{T}_{n-1}) \\ &\quad (G_2 \otimes I_{2^{2(n-1)}})(I_2 \otimes L_2^{2^n} \otimes I_{2^{n-1}}). \end{aligned}$$

If vector X of size 4^n is a data collection, such as image pixels, stored in the column major order, then $Y = H_n X$ is the same data collection stored in the Hilbert space-filling curve order.

4.2 Iterative Tensor Product Formulation

The recursive tensor product formula of the Hilbert space-filling curve permutation can be expanded repeatedly to derive the iterative tensor product formula as in Theorem 4.1. We will prove the theorem using mathematical induction.

Theorem 4.1 (Iterative Tensor Product Formula) For $n \geq 1$,

$$\begin{aligned} H_n &= \prod_{i=0}^{n-1} I_{4^i} \otimes \\ &\quad \left[(T_{n-i-1} \oplus I_{2^{2(n-i-1)}} \oplus I_{2^{2(n-i-1)}} \oplus \overline{T}_{n-i-1}) \right. \\ &\quad \left. (G_2 \otimes I_{2^{2(n-i-1)}})(I_2 \otimes L_2^{2^{n-i}} \otimes I_{2^{n-i-1}}) \right], \end{aligned}$$

where T_{i-1} is the transposition operation $L_{2^{i-1}}^{2^{2(i-1)}}$ and \overline{T}_{i-1} is the anti-diagonal transposition operation of $2^{i-1} \times 2^{i-1}$ matrix.

Proof:

Base case:

$$\begin{aligned} H_1 &= \prod_{i=0}^0 I_{4^i} \otimes \left[(T_{-i} \oplus I_{2^{2(-i)}} \oplus I_{2^{2(-i)}} \oplus \overline{T}_{-i}) \right. \\ &\quad \left. (G_2 \otimes I_{2^{2(-i)}})(I_2 \otimes L_2^{2^{1-i}} \otimes I_{2^{-i}}) \right] \\ &= I_{4^0} \otimes \left[(T_0 \oplus I_{2^0} \oplus I_{2^0} \oplus \overline{T}_0)(G_2 \otimes I_{2^0}) \right. \\ &\quad \left. (I_2 \otimes L_2^2 \otimes I_{2^0}) \right] \\ &= G_2 \end{aligned}$$

Induction step:

For $k \geq 1$, assume the following induction hypothesis holds

$$\begin{aligned} H_k &= \prod_{i=0}^{k-1} I_{4^i} \otimes \\ &\quad \left[(T_{k-i-1} \oplus I_{2^{2(k-i-1)}} \oplus I_{2^{2(k-i-1)}} \oplus \overline{T}_{k-i-1}) \right. \\ &\quad \left. (G_2 \otimes I_{2^{2(k-i-1)}})(I_2 \otimes L_2^{2^{k-i}} \otimes I_{2^{k-i-1}}) \right]. \end{aligned}$$

We obtain the following result:

$$\begin{aligned} H_{k+1} &= (I_4 \otimes H_{(k+1)-1}) \\ &\quad (T_{(k+1)-1} \oplus I_{2^{2((k+1)-1)}} \oplus I_{2^{2((k+1)-1)}} \oplus \overline{T}_{(k+1)-1}) \\ &\quad (G_2 \otimes I_{2^{2((k+1)-1)}})(I_2 \otimes L_2^{2^{(k+1)}} \otimes I_{2^{(k+1)-1}}) \\ &= \left[I_4 \otimes \left[\prod_{i=0}^{k-1} I_{4^i} \otimes \left[(T_{k-i-1} \oplus I_{2^{2(k-i-1)}} \oplus \right. \right. \right. \\ &\quad \left. \left. \left. I_{2^{2(k-i-1)}} \oplus \overline{T}_{k-i-1}) \right] \right] \right. \\ &\quad \left. (G_2 \otimes I_{2^{2(k-i-1)}})(I_2 \otimes L_2^{2^{k-i}} \otimes I_{2^{k-i-1}}) \right] \\ &\quad (T_{(k+1)-1} \oplus I_{2^{2((k+1)-1)}} \oplus I_{2^{2((k+1)-1)}} \oplus \overline{T}_{(k+1)-1}) \\ &\quad (G_2 \otimes I_{2^{2((k+1)-1)}})(I_2 \otimes L_2^{2^{(k+1)}} \otimes I_{2^{(k+1)-1}}) \\ &= \left[\prod_{i=0}^{k-1} I_{4^{i+1}} \otimes \left[(T_{k-i-1} \oplus I_{2^{2(k-i-1)}} \oplus \right. \right. \\ &\quad \left. \left. I_{2^{2(k-i-1)}} \oplus \overline{T}_{k-i-1}) \right] \right. \\ &\quad \left. (G_2 \otimes I_{2^{2(k-i-1)}})(I_2 \otimes L_2^{2^{k-i}} \otimes I_{2^{k-i-1}}) \right] \\ &\quad (T_{(k+1)-1} \oplus I_{2^{2((k+1)-1)}} \oplus I_{2^{2((k+1)-1)}} \oplus \overline{T}_{(k+1)-1}) \\ &\quad (G_2 \otimes I_{2^{2((k+1)-1)}})(I_2 \otimes L_2^{2^{(k+1)}} \otimes I_{2^{(k+1)-1}}) \\ &= \left[\prod_{i=1}^{(k+1)-1} I_{4^i} \otimes \left[(T_{(k+1)-i-1} \oplus I_{2^{2((k+1)-i-1)}} \oplus \right. \right. \\ &\quad \left. \left. I_{2^{2((k+1)-i-1)}} \oplus \overline{T}_{(k+1)-i-1}) \right] \right. \\ &\quad \left. (G_2 \otimes I_{2^{2((k+1)-i-1)}})(I_2 \otimes L_2^{2^{k-i}} \otimes I_{2^{k-i-1}}) \right] \\ &\quad \left[I_{4^0} \otimes \left[(T_{(k+1)-1} \oplus I_{2^{2((k+1)-1)}} \oplus \right. \right. \\ &\quad \left. \left. I_{2^{2((k+1)-1)}} \oplus \overline{T}_{(k+1)-1}) \right] \right. \\ &\quad \left. (G_2 \otimes I_{2^{2((k+1)-1)}})(I_2 \otimes L_2^{2^{(k+1)}} \otimes I_{2^{(k+1)-1}}) \right] \\ &= \prod_{i=0}^{(k+1)-1} I_{4^i} \otimes \left[(T_{(k+1)-i-1} \oplus I_{2^{2((k+1)-i-1)}} \oplus \right. \\ &\quad \left. I_{2^{2((k+1)-i-1)}} \oplus \overline{T}_{(k+1)-i-1}) \right. \\ &\quad \left. (G_2 \otimes I_{2^{2((k+1)-i-1)}})(I_2 \otimes L_2^{2^{k-i}} \otimes I_{2^{k-i-1}}) \right]. \end{aligned}$$

Q.E.D.

The recursive and iterative tensor product formulas can be directly translated into high-level programming language programs. We will explain program generation in Section 5.

4.3 Inverse Hilbert Space-Filling Curve Permutation

Hilbert space-filling curve permutation can be applied in various problem domain such as image processing. In image processing, it is important to perform both compression and decompression operations. Hence, inverse Hilbert space-filling curve permutation is needed in the decompression operation.

Given the iterative tensor product formula H_n as in Theorem 4.1, we obtain the inverse Hilbert space-filling curve permutation as the theorem below.

Theorem 4.2 (Inverse Tensor Product Formula) For $n \geq 1$,

$$H_n^{-1} = \prod_{i=0}^{n-1} I_{4^{n-i-1}} \otimes \left[(I_2 \otimes L_2^{2^{i+1}} \otimes I_2) (G_2 \otimes I_{2^{2i}}) (T_i \oplus I_{2^{2i}} \oplus I_{2^{2i}} \oplus \overline{T}_i) \right].$$

Proof:

$$\begin{aligned} H_n^{-1} &= \left[\prod_{i=0}^{n-1} I_{4^i} \otimes \left[(T_{n-i-1} \oplus I_{2^{2(n-i-1)}} \oplus I_{2^{2(n-i-1)}} \oplus \overline{T}_{n-i-1}) (G_2 \otimes I_{2^{2(n-i-1)}}) (I_2 \otimes L_2^{2^{n-i}} \otimes I_{2^{n-i-1}}) \right] \right]^{-1} \\ &= \prod_{i=n-1}^0 I_{4^i} \otimes \left[(I_2 \otimes L_2^{2^{n-i}} \otimes I_{2^{n-i-1}})^{-1} (G_2 \otimes I_{2^{2(n-i-1)}})^{-1} (T_{n-i-1} \oplus I_{2^{2(n-i-1)}} \oplus I_{2^{2(n-i-1)}} \oplus \overline{T}_{n-i-1})^{-1} \right] \\ &= \prod_{i=n-1}^0 I_{4^i} \otimes \left[(I_2 \otimes L_2^{2^{n-i}})^{-1} \otimes I_{2^{n-i-1}} (G_2^{-1} \otimes I_{2^{2(n-i-1)}}) (T_{n-i-1}^{-1} \oplus I_{2^{2(n-i-1)}} \oplus I_{2^{2(n-i-1)}} \oplus \overline{T}_{n-i-1}^{-1}) \right] \\ &= \prod_{i=n-1}^0 I_{4^i} \otimes \left[(I_2 \otimes L_2^{2^{n-i}} \otimes I_{2^{n-i-1}}) (G_2 \otimes I_{2^{2(n-i-1)}}) (T_{n-i-1} \oplus I_{2^{2(n-i-1)}} \oplus I_{2^{2(n-i-1)}} \oplus \overline{T}_{n-i-1}) \right] \\ &= \prod_{i=0}^{n-1} I_{4^{n-i-1}} \otimes \left[(I_2 \otimes L_2^{2^{i+1}} \otimes I_2) (G_2 \otimes I_{2^{2i}}) (T_i \oplus I_{2^{2i}} \oplus I_{2^{2i}} \oplus \overline{T}_i) \right]. \end{aligned}$$

Q.E.D.

5 Program Generation

Both the recursive and iterative tensor product formulas can be used to generate programs for Hilbert space-filling curves. We use the syntax of C language to illustrate the generated programs and assume the data elements are of type `int`, though it can be any of other types.

5.1 Recursive Program

We specify the interface of the recursive function for the tensor product formula as `void hilbert(int n, int *a)`, where `n` is the parameter denoting the problem size $2^n \times 2^n$ and `a` is a pointer pointing to the starting address of the input array of $2^n \times 2^n$ elements. When the function returns, the result is stored in the array pointed by `a`. The recursive program is as below:

```
01 void hilbert(int n, int *a) {
02   int i1, j1, i;
03   int b[(int)pow(4,n)];
04   int tmp;
05   if (n==1) {
06     tmp = a[2];
07     a[2] = a[3];
08     a[3] = tmp; }
```

```
09   else {
10     for (i1=0; i1<(int)pow(2,n-1); i1++) {
11       for (j1=0; j1<(int)pow(2,n-1); j1++) {
12         // i0=0, j0=0
13         b[j1*(int)(pow(2,n-1))+i1]=
14           a[i1*(int)(pow(2,n))+j1];
15         // i0=0, j0=1
16         b[i1*(int)(pow(2,n-1))+j1+
17           (int)pow(4,n-1)]=
18           a[i1*(int)(pow(2,n))+j1+(int)pow(2,n-1)];
19         // i0=1, j0=0
20         b[((int)pow(2,n-1)-1-j1)*
21           (int)(pow(2,n-1))+
22           ((int)pow(2,n-1)-1-i1)+
23           3*(int)pow(4,n-1)]=
24           a[i1*(int)(pow(2,n))+j1+
25             (int)pow(2,2*n-1)];
26         // i0=1, j0=1
27         b[i1*(int)(pow(2,n-1))+j1+
28           2*(int)pow(4,n-1)]=
29           a[i1*(int)(pow(2,n))+j1+
30             (int)pow(2,2*n-1)+(int)pow(2,n-1)];
31       } }
32   for (i=0; i<(int)pow(4,n); i++) a[i] = b[i];
33   hilbert(n-1, &a[0]);
34   hilbert(n-1, &a[(int)pow(4,n-1)]);
35   hilbert(n-1, &a[2*(int)pow(4,n-1)]);
36   hilbert(n-1, &a[3*(int)pow(4,n-1)]); }
```

Lines 6 to 8 are corresponding to the base case H_1 which accepts an array of four elements and performs G_2 permutation swapping the third and the fourth elements. The tensor basis $e_i^{2^n} \otimes e_j^{2^n}$ of the input data for the recursive formula H_n is factorized to $e_{i_0}^{2^n} \otimes e_{i_1}^{2^{n-1}} \otimes e_{j_0}^{2^n} \otimes e_{j_1}^{2^{n-1}}$. The `for` loops in lines 10 and 11 are corresponding to $e_{i_1}^{2^{n-1}}$ and $e_{j_1}^{2^{n-1}}$. The bases $e_{i_0}^{2^n}$ and $e_{j_0}^{2^n}$ are implemented as four cases (0, 0), (0, 1), (1, 0), and (1, 1) in lines 12, 13, 14, and 15, respectively. In line 16, the intermediate result in array `b` is copied to array `a`. Finally, $I_4 \otimes H_{n-1}$ is translated into the four recursive calls to `hilbert` in lines 17 to 20.

5.2 Iterative Program

Similarly to the recursive program, we specify the interface of the recursive function for the tensor product formula as `void hilbert(int n, int *a)`, where `n` is the parameter denoting the problem size $2^n \times 2^n$ and `a` is a pointer pointing to the starting address of the input array of $2^n \times 2^n$ elements. When the function returns, the result is stored in the array pointed by `a`. The iterative program is as below:

```
01 void hilbert(int n, int *a) {
02   int i, k, i1, j1;
03   int b[(int)pow(4,n)];
04   for (i=0; i<n; i++) {
05     for (k=0; k<(int)pow(4,i); k++) {
06       for (i1=0; i1<(int)pow(2,n-i-1); i1++) {
07         for (j1=0; j1<(int)pow(2,n-i-1); j1++) {
08           // i0=0, j0=0
```

```

08     b[k*(int)pow(4,n-i)+
        j1*(int)(pow(2,n-i-1))+i1]=
a[k*(int)pow(4,n-i)+
    i1*(int)(pow(2,n-i))+j1];
// i0=0, j0=1
09     b[k*(int)pow(4,n-i)+
        i1*(int)(pow(2,n-i-1))+j1
        +(int)pow(4,n-i-1)]=
a[k*(int)pow(4,n-i)+
    i1*(int)(pow(2,n-i))+j1+
    (int)pow(2,n-i-1)];
// i0=1, j0=0
10     b[k*(int)pow(4,n-i)
        +((int)pow(2,n-i-1)-1-j1)*
        (int)(pow(2,n-i-1))
        +((int)pow(2,n-i-1)-1-i1)+
        3*(int)pow(4,n-i-1)]=
a[k*(int)pow(4,n-i)+
    i1*(int)(pow(2,n-i))+j1
    +(int)pow(2,2*(n-i)-1)];
// i0=1, j0=1
11     b[k*(int)pow(4,n-i)+
        i1*(int)(pow(2,n-i-1))+j1
        +2*(int)pow(4,n-i-1)]=
a[k*(int)pow(4,n-i)+
    i1*(int)(pow(2,n-i))+j1
    +(int)pow(2,2*(n-i)-1)+
    (int)pow(2,n-i-1)];
    } } }
12     for (k=0; k<(int)pow(4,k); k++)
        a[k] = b[k];
    } }

```

The `for` loop in line 4 is the iteration corresponding to matrix product $\prod_{i=0}^{n-1}$ in the iterative formula. The tensor basis $e_i^{2^n} \otimes e_j^{2^n}$ of the input data for the iterative formula H_n is factorized to $e_k^{4^i} \otimes e_{i_0}^2 \otimes e_{i_1}^{2^{n-k-1}} \otimes e_{j_0}^2 \otimes e_{j_1}^{2^{n-k-1}}$. The `for` loops in lines 5, 6, and 7 are corresponding to $e_k^{4^i}$, $e_{i_1}^{2^{n-k-1}}$ and $e_{j_1}^{2^{n-k-1}}$. The bases $e_{i_0}^2$ and $e_{j_0}^2$ are implemented as four cases (0, 0), (0, 1), (1, 0), and (1, 1) in lines 8, 9, 10, and 11, respectively. In line 12, the intermediate result in array `b` is copied to array `a`.

6 Conclusions

We use a tensor product based algebraic theory to model the Hilbert space-filling curves. The Hilbert space-filling curve permutation algorithm is expressed in both recursive and iterative tensor product formulas. In addition, the iterative tensor product formula for the inverse Hilbert space-filling curve permutation is derived. These formulas are used to generate recursive and iterative programs of C programming language.

The tensor product theory is also suitable for expressing other space-filling curves such as two-dimensional Peano, Moore, and Wunderlich space-filling curves and three-dimensional Hilbert space-filling curves. These tensor product formulas of space-filling curves consist of block recursive permutation, radix-2 and radix-3 Gray permutations for two digits and three digits, and permutations that

transform the coordinate system. We will develop tensor product formulas of these space-filling curves in the future.

Space-filling curves are used in various applications such as image compression, VLSI component layout, and R-tree indexing. For the example of image compression, data collection can be rearranged according to 2D or 3D space-filling curve permutations to enhance locality relationship and improve compression ratio. Furthermore, the inverse space-filling curve permutations can be used in image decompression. In future work, we will investigate the application of space-filling curves to various problem domains.

References

- [1] S. Biswas. Hilbert scan and image compression. In *15th International Conference on Pattern Recognition*, volume 3, pages 207–210, 2000.
- [2] A. R. Butz. Space filling curves and mathematical programming. *Information and Control*, 12(4):314–330, 1968.
- [3] A. R. Butz. Convergence with Hilbert’s space filling curve. *Journal of Computer and System Sciences*, 3(2):128–146, 1969.
- [4] D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan, and R. W. Johnson. EXTENT: A portable programming environment for designing and implementing high-performance block-recursive algorithms. In *Proceedings of Supercomputing ’94*, pages 49–58, 1994.
- [5] M.-H. Fan, C.-H. Huang, Y.-C. Chung, J.-S. Liu, and J.-Z. Lee. A programming methodology for designing parallel prefix algorithms. In *Proceedings of the 2001 International Conference on Parallel Processing*, pages 463–470, 2001.
- [6] D. M. Gavrilu. R-tree index optimization. In *Sixth International Symposium on Spatial Data Handling*, volume 2, pages 771–791, Edinburgh, Scotland, 1994.
- [7] A. Graham. *Kronecker Products and Matrix Calculus: With Applications*. Ellis Horwood Limited, 1981.
- [8] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A framework for generating distributed-memory parallel programs for block recursive algorithms. *Journal of Parallel and Distributed Computing*, 34(2):137–153, 1996.

- [9] D. Hilbert. Über die stetige abbildung einer linie auf Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.
- [10] C.-H. Huang, J. R. Johnson, and R. W. Johnson. A tensor product formulation of Strassen’s matrix multiplication algorithm. *Appl. Math Letters*, 3(3):104–108, 1990.
- [11] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342. ACM Press, 1990.
- [12] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying and implementing Fourier transform algorithms on various architectures. *Circuits Systems Signal Process*, 9(4):450–500, 1990.
- [13] R. W. Johnson, C.-H. Huang, and J. R. Johnson. Multilinear algebra and parallel programming. *The Journal of Supercomputing*, 5(2–3):189–217, 1991.
- [14] S. Kamata, R. O. Eason, and Y. Bandou. A new algorithm for N-dimensional Hilbert scanning. *IEEE Transactions on Image Processing*, 8(7):964–973, 1999.
- [15] S. Kamata, M. Niimi, R. O. Eason, and E. Kawaguchi. An implementation of an N-dimensional Hilbert scanning algorithm. In *Proceedings of the 9th Scandinavian Conference on Image Analysis*, pages 431–440, 1995.
- [16] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 500–509, Santiago, Chile, 1994.
- [17] Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management, Washington, DC, USA, November 1-5, 1993*, pages 490–499. ACM, 1993.
- [18] B. Kumar, C.-H. Huang, P. Sadayappan, and R. W. Johnson. An algebraic approach to cache memory characterization for block recursive algorithms. In *1994 International Computer Symposium*, pages 336–342, 1994.
- [19] B. Kumar, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A tensor product formulation of Strassen’s matrix multiplication algorithm with memory reduction. *Scientific Programming*, 4(4):275–289, 1995.
- [20] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, pages 497–506. IEEE Computer Society, 1997.
- [21] S. T. Leutenegger and M. A. Lopez. The effect of buffering on the performance of R-trees. *Knowledge and Data Engineering*, 12(1):33–44, 2000.
- [22] G. Melnikov and A. K. Katsaggelos. A non-uniform segmentation optimal hybrid fractal/DCT image compression algorithm. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 2573–2576, 1998.
- [23] N. Memon, D. L. Neuhoff, and S. Shende. An analysis of some common scanning techniques for lossless image coding. *IEEE Transactions on Image Processing*, 9:1837–1848, 2000.
- [24] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141, 2001.
- [25] J. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Puschel, and M. Veloso. Spiral: Automatic implementation of signal processing algorithms. In *Proc. HPEC 2000*. MIT Lincoln Laboratories (on CD-Rom), 2000.
- [26] B. O’Sullivan. Applying partial evaluation to VLSI design rule checking, 1995.
- [27] G. Peano. Sur une courbe qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890.
- [28] J. Quinqueton and M. Berthod. A locally adaptive Peano scanning algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(4):403–412, 1981.
- [29] S. Rovetta and R. Zunino. VLSI circuits with fractal layout for spatial image decorrelation. In *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, volume 4, pages 110–113, 1999.
- [30] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.