FAST 2-DIMENSIONAL 8 × 8 INTEGER TRANSFORM ALGORITHM DESIGN FOR H.264/AVC FIDELITY RANGE EXTENSIONS∗

Chih-Peng Fan†, Member

SUMMARY In this letter, efficient two-dimensional (2-D) fast algorithms for realizations of 8 × 8 forward and inverse integer transforms in H.264/AVC fidelity range extensions (FRExt) are proposed. Based on matrix factorizations with Kronecker product and direct sum operations, efficient fast 2-D 8 × 8 forward and inverse integer transforms can be derived from the one-dimensional (1-D) fast 8 × 8 forward and inverse integer transforms through matrix operations. The proposed fast 2-D 8 × 8 forward and inverse integer transform designs don’t require transpose memory in hardware realizations. The fast 2-D 8 × 8 integer transforms require fewer latency delays and provide a larger throughput rate than the row-column based method. With regular modularity, the proposed fast algorithms are suitable for VLSI implementations to achieve H.264/AVC FRExt high-profile signal processing.

key words: H.264, fidelity range extensions (FRExt), integer transform, fast algorithm

1. Introduction

Within video and image compression standards, such as JPEG, MPEG-1/2, and MPEG-4, the discrete cosine transform (DCT) technology has been widely used for transform coding. In ITU-T H.264/Advanced Video Coding (AVC) [1], [2] standard, unlike the popular 8 × 8 DCT utilized in previous standards, the 4 × 4 integer transform can be computed exactly in integer arithmetic to avoid inverse transform mismatch problems [3], [9]. When we combine integer transforms with quantization designs in H.264/AVC, the 4 × 4 integer transform can be computed without multiplications, just additions and shifts.

To further expand the H.264/AVC application to high definition (HD) materials, the H.264/AVC committee has recently chosen the fidelity range extensions (FRExt) for coding high resolution video sequences [4], [5]. In the recent FRExt proposal [6], [7], the authors provided new 8 × 8 forward and inverse integer transforms for improvement of coding efficiency. From experimental results, the simple coding tool with new 8 × 8 integer transforms achieves average bit-rate reductions of around 10% for the coding of progressive-scan HD resolution material [7]. When we combine integer transforms with quantization designs, the 8 × 8 integer transforms are also computed only with additions and shift operations.

In this paper, the memoryless fast 2-D 8 × 8 integer transform algorithms for H.264/AVC FRExt are developed with high throughput rate and less latency delay. The following part of this letter is organized as follows. Section 2 reviews the 2-D 8 × 8 forward and inverse integer transforms. In Sect. 3, we review several useful matrix operations and properties for clarifying the derivations of the proposed fast algorithms. In Sect. 4, the fast 1-D 8 × 8 forward and inverse integer transform algorithms are introduced with matrix factorizations. The computational complexity is described for fast 1-D 8 × 8 integer transforms. Based on the efficient 1-D fast forward and inverse integer algorithms, the novel fast 2-D 8 × 8 forward and inverse integer transform architectures are developed in Sect. 5. In Sect. 5, the computational complexity and comparisons of architectures will be discussed for fast 2-D 8 × 8 integer transforms. Finally, we give a conclusion in Sect. 6.

2. 8 × 8 Integer Transforms

In the H.264/AVC FRExt proposal, the 2-D forward integer transform can be computed in a separable way as a 1-D horizontal (row) transform followed by a 1-D vertical (column) transform, where the corresponding 1-D 8 × 8 forward transform is shown by the following $C$ matrix [6], [7]

$$C = \begin{bmatrix}
8 & 12 & 10 & 6 & 3 & -3 & -6 & -10 & -12 \\
8 & 4 & -4 & -8 & -8 & -8 & 4 & 8 & 8 \\
10 & -3 & -12 & -6 & 6 & 12 & 3 & -10 & 8 \\
8 & -8 & -8 & 8 & 8 & -8 & -8 & 8 & 8 \\
6 & -12 & 3 & 10 & -10 & -3 & 12 & -6 & 8 \\
4 & -8 & 8 & -4 & -8 & 8 & -8 & 4 & 8 \\
3 & -6 & 10 & -12 & 12 & -10 & 6 & -3 & 8
\end{bmatrix}$$

(1)

Thus, the 2-D 8 × 8 forward integer transform can be factorized to the following equivalent form [2] as

$$Y = (CXC^T) \odot E_f,$$

(2)

where “$\odot$” means that each element of $(CXC^T)$ is multiplied by the scaling factor in the same position in scaling matrix $E_f$, and “$T$” means the matrix transposition. Meanwhile, the 2-D $8 \times 8$ inverse integer transform is described as

$$X = C_i(Y \odot E_f)C_i^T = C_i\tilde{Y}C_i^T,$$

(3)

Manuscript revised July 26, 2006.

†The author is with the Department of Electrical Engineering, National Chung Hsing University, 250 Kuo-Kuang Road, Tai-chung, Taiwan, R.O.C.

‡This work was supported by the National Science Council, Taiwan, R.O.C., under Grant NSC95-2220-E-005-006.

a) E-mail: cpfan@dragon.nchu.edu.tw

DOI: 10.1093/ietisy/e89-d.12.3006

Copyright © 2006 The Institute of Electronics, Information and Communication Engineers
where \( C_i \) is the corresponding 1-D \( 8 \times 8 \) inverse transform matrix and we define \( C_i = C_i^T \) \[6,7\]. Since the scaling matrices \( E_i \) in (2) and \( E_i \) in (3) can be merged to the quantization and pre-scaling processes respectively, then the core forward transform \((CXC_i^T)\) in (2) and core inverse transform \((C_iYC_i^T)\) in (3) become 2-D \( 8 \times 8 \) forward and inverse integer transforms respectively. In the following sections, we will concentrate on the derivations of the fast core forward and inverse integer transforms in (2) and (3).

3. Review of Matrix Operations

The Kronecker product, which is denoted as “\( \otimes \)”, is defined as follows \[8\]

\[
A \otimes B = \begin{bmatrix}
  a_{11}B & \cdots & a_{1j}B \\
  \vdots & \ddots & \vdots \\
  a_{mj}B & \cdots & a_{mj}B
\end{bmatrix},
\]

where \( A = [a_{ij}] \in R^{m \times r} \) and \( B = [b_{ij}] \in R^{n \times s} \). After the Kronecker product, the size of the matrix becomes \( (A \otimes B) \in R^{mn \times rs} \), which is product-expanded. The direct sum, which is denoted as “\( \oplus \)”, is treated as the diagonal layout of two matrices as follows \[8\],

\[
(A \oplus B) = \begin{bmatrix}
  A & 0 \\
  0 & B
\end{bmatrix}.
\]

After the direct sum operation, the size of the matrix is sum-expanded as \( (A \oplus B) \in R^{(m+n)(r+s)} \). A useful matrix formula for the derivations of this letter is denoted in the following,

\[
\begin{bmatrix}
  A & B \\
  A & -B
\end{bmatrix} = (H_2 \otimes I)(A \oplus B),
\]

where \( H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \), and \( I \) is an identity matrix. The Kronecker-product and direct-sum transposition properties are described as

\[
(A \oplus B)^T = (A^T \oplus B^T),
\]

and

\[
(A \otimes B)^T = (A^T \otimes B^T).
\]

The Kronecker product has the factorization property, which is stated as follows \[8\]

\[
AC \otimes BD = (A \otimes B)(C \otimes D).
\]

The distribution law related to direct-sum and Kronecker product operations is given by \[8\],

\[
(A_1 \oplus A_2) \otimes (B_1 \oplus B_2) = [A_1 \otimes (B_1 \oplus B_2)] \oplus [A_2 \otimes (B_1 \oplus B_2)].
\]

4. Fast 1-Dimensional \( 8 \times 8 \) Forward and Inverse Integer Transforms

According to the fast butterfly operations in \[6,7\], the fast \( 1-D 8 \times 8 \) forward integer transform can be factorized into the three-stage matrix computations as follows

\[
C = C_3 \cdot C_2 \cdot C_1,
\]

where

\[
C_1 = \begin{bmatrix}
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
  0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
  1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
  0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\
  0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\
  0 & 0 & 0 & 1 & -1 & 0 & 0 & 0
\end{bmatrix},
\]

and

\[
C_3 = \begin{bmatrix}
  1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
  0 & 0 & 1 & \frac{1}{2} & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 1 & \frac{1}{2} & 0 & 0 \\
  1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & 0 \\
  0 & 0 & 1 & \frac{1}{2} & -1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & -1
\end{bmatrix},
\]

The matrix computation in (12) can be further decomposed by

\[
C_1 = (H_2 \otimes I_4) \cdot P_c,
\]

where \( I_4 \) is \( 4 \times 4 \) identity matrix. The permutation matrix \( P_c \) is defined by

\[
P_c = \begin{bmatrix}
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}.
\]

Thus, the computational complexity in (15) needs 8 additions. By using (6), the matrix computation in (13) can be
further factorized and denoted by
\[ C_2 = [(H_2 \otimes I_2)P_4] \oplus Q_1, \]  
(17)
where
\[ P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad Q_1 = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \\ 0 \\ 0 \\ -3 \\ -2 \\ -1 \\ -3 \\ 0 \\ 1 \\ -2 \\ 0 \\ 1 \\ -1 \\ 3 \\ 2 \\ -2 \\ -1 \end{bmatrix}. \]  
(18)

In (18), the computation of \( 3/2x \) can be replaced by \( x + (x \gg 1) \), where \( (x \gg 1) \) means \( x \) right shifts 1 bit [6], [7]. Thus, the computational complexity in (17) requires 4 shift operations and 16 additions. The matrix computation in (14) can be further decomposed by
\[ C_3 = P_r \cdot \tilde{C}_3, \]  
(19)
where
\[ P_r = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \]  
(20)
and
\[ \tilde{C}_3 = (H_2 \oplus Q_2) \oplus Q_3. \]  
(21)

In (21), we define that
\[ Q_2 = \begin{bmatrix} 1 \\ 1/2 \\ -1 \end{bmatrix} \quad \text{and} \quad Q_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1/4 \\ 0 \\ 1/4 \\ 0 \\ -1/4 \\ 1/4 \\ 0 \\ 0 \\ -1 \end{bmatrix}. \]

In (21), the computation of \( 1/2x \) can be replaced by \( (x \gg 1) \). Then the computation of \( 1/4x \) can be replaced by \( (x \gg 2) \), where \( (x \gg 2) \) means \( x \) right shifts 2 bits [6], [7]. Thus, the computational complexity in (19) needs 6 shift operations and 8 additions.

By replacing (11) with (15), (17) and (19), we can rewrite \( C \), which is stated in (11), in the following.
\[ C = C_3 \cdot C_2 \cdot C_1 = P_r \cdot [(H_2 \oplus Q_2) \oplus Q_3] \cdot \{(H_2 \otimes I_2)P_4] \oplus Q_1\} \cdot (H_2 \otimes I_4) \cdot P_r. \]  
(22)
Equation (22) is the proposed structure for the fast 1-D \( 8 \times 8 \) forward integer transform. Similarly, the fast 1-D \( 8 \times 8 \) inverse integer transform can be written as (23), where we apply the Kronecker-product and direct-sum transposition properties in (7) and (8).
\[ C_i = C_i^T = C_i^T \cdot C_i^T \cdot C_i^T = \left[ P_r^c (H_2 \otimes I_4) \cdot \left[ (P_4 (H_2 \otimes I_2)] \oplus Q_1^T \right] \right] \cdot \{(H_2 \oplus Q_2) \oplus Q_3^T\} \cdot P_r^c. \]  
(23)

In Eq. (22), the middle stage permutation \( P_4 \), the first stage permutation \( P_r \), and the last stage permutation \( P_r^c \) need no arithmetic computations, which are implemented by pure hard-wired connections. Table 1 shows the comparisons of computational complexity for fast 1-D \( 8 \times 8 \) forward or inverse integer transforms. The computational complexity of proposed 1-D fast methods in (22) and (23) is equivalent to that of the fast computations in [6], [7]. In (22) and (23), when we define that the latency delay of one addition or one addition/shift operation requires one clock cycle, the computational latency delays in (22) occupy 4 clock cycles, where the 4 pipelined phases are noted in the data flow diagram of (22) shown in Fig. 1. In Fig. 1, the computations of \( Q_1 \) need two pipelined phases, where parallel computations of partial-sum additions and addition/shift operations are processed in the first pipelined phase, and pure additions for outputs of \( Q_1 \) are processed in the second pipelined phase. Based on the similar structure in (22), the computational latency delays of the fast 1-D inverse \( 8 \times 8 \) integer transform in (23) also occupy 4 clock cycles. The data flow diagram and the pipelined phases of (23) are shown in Fig. 2.

5. Fast 2-Dimensional \( 8 \times 8 \) Forward and Inverse Integer Transforms

Generally, the linear transformation of matrix \( X \) shown as
\[ AXB = Y \]  
(24)
can be expressed by the transformation of the column-wise stacking vector of \( X \) as [8]
\[ (B^T \otimes A) \cdot \text{vec}(X) = \text{vec}(Y). \]  
(25)
Then the 2-D \( 8 \times 8 \) core forward integer transform in (2) can be expressed by
\[ (C \otimes C) \cdot \text{vec}(X) = \text{vec}(Y), \]  
(26)
where \( \text{vec}(X) \) and \( \text{vec}(Y) \) are \( 64 \times 1 \) vector data. Based on (22), the 2-D \( 8 \times 8 \) core forward integer transform in (26) can be written as

### Table 1 Comparisons of computational complexity for 1-D \( 8 \times 8 \) forward or inverse integer transform.

<table>
<thead>
<tr>
<th>Shift operations</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Fig. 1  Data flow diagram of fast 1-D forward $8 \times 8$ integer transform with 4 pipelined phases.

\[ C \otimes C = (P_r \otimes P_r) \]
\[ \cdot \left[ \left[ (H_2 \oplus Q_2) \oplus Q_1 \right] \otimes \left[ (H_2 \oplus Q_2) \oplus Q_1 \right] \right] \]
\[ \cdot \left[ \left[ (H_2 \otimes I_4) P_4 \right] \otimes \left[ (H_2 \otimes I_4) P_4 \right] \right] \]
\[ \cdot \left[ (H_2 \otimes I_4 \otimes H_2 \otimes I_4) \right] \cdot \left( P_r \otimes P_r \right), \quad (27) \]

and then we define $\tilde{P}_r = (P_r \otimes P_r)$ and $\tilde{P}_r = (P_r \otimes P_r)$. The first stage permutation $\tilde{P}_r$ and the last stage permutation $\tilde{P}_r$ also do not need arithmetic computation by using pure hard-wired connections. By using the matrix equality in (9), the last middle term in (27) can be expressed by

\[ (H_2 \otimes I_4 \otimes H_2 \otimes I_4) \]
\[ = (H_2 \otimes I_4 \otimes I_2 \otimes I_4)(I_2 \otimes I_4 \otimes H_2 \otimes I_4) \]
\[ = (H_2 \otimes I_3)(I_8 \otimes H_2 \otimes I_4), \quad (28) \]

where the two-stage parallel modules are realized with butterfly structures. The computation in (28) needs 128 additions. By the matrix equality in (9) and (10), the first middle term in (27) can be rewritten as

\[ \left[ (H_2 \oplus Q_2) \oplus Q_1 \right] \otimes \left[ (H_2 \oplus Q_2) \oplus Q_1 \right] \]
\[ = \left[ I_4 \otimes \left[ (H_2 \oplus Q_2) \oplus Q_1 \right] \right] \cdot \left[ (H_2 \oplus Q_2) \oplus I_8 \right] \]
\[ \oplus \left[ I_4 \otimes \left[ (H_2 \oplus Q_2) \oplus Q_1 \right] \right] \cdot \left[ Q_3 \otimes I_8 \right]. \quad (29) \]

The computation in (29) needs 96 shift operations and 128 additions. By the matrix equality in (9) and (10), the second
middle term in (27) can be rewritten as
\[
[(H_2 \otimes I_2)P_4] \oplus Q_1 \oplus [(H_2 \otimes I_2)P_4] \oplus Q_1
\]
\[
= [I_4 \otimes [(H_2 \otimes I_2)P_4] \oplus Q_1] \oplus [(H_2 \otimes I_2)P_4] \oplus I_8 \\
+ [I_4 \otimes [(H_2 \otimes I_2)P_4] \oplus Q_1] \oplus Q_1 \oplus I_8.
\]  
(30)

The computation in (30) needs 64 shift operations and 256 additions. By replacing (27) with (28), (29) and (30), the fast 2-D 8 × 8 core forward transform becomes
\[
C \otimes C = \tilde{P}_F \cdot [I_4 \otimes [(H_2 \otimes Q_2) \oplus Q_1]] \cdot [(H_2 \otimes Q_2) \otimes I_8] \\
+ [I_4 \otimes [(H_2 \otimes Q_2) \oplus Q_1]] \cdot [(H_2 \otimes Q_2) \otimes I_8] \\
+ [I_4 \otimes [(H_2 \otimes Q_2) \oplus Q_1]] \cdot (Q_1 \otimes I_8).
\]  
(31)

Equation (31) is the proposed fast 2-D 8 × 8 forward integer transform algorithm. Similarly, the 2-D 8 × 8 core inverse integer transform in (3) can be expressed by
\[
(C_i \otimes C_i) \cdot \text{vec}(\tilde{Y}) = \text{vec}(X).
\]  
(32)

Then the 2-D 8 × 8 core inverse integer transform in (32) can be written as
\[
C_i \otimes C_i = C_i^T \otimes C_i = (C \otimes C)^T.
\]  
(33)

By using the matrix properties in (7) and (8), the (33) becomes
\[
(C \otimes C)^T = \tilde{P}_F \cdot [I_4 \otimes H_2 \otimes I_4] \cdot (H_2 \otimes I_{32}) \\
\cdot \left\{ \begin{array}{l}
[[P_4(H_2 \otimes I_2)] \otimes I_8] \cdot [I_4 \otimes [P_4(H_2 \otimes I_2) \oplus Q_1^T]] \\
\oplus [Q_1^T \otimes I_8] \cdot [I_4 \otimes [P_4(H_2 \otimes I_2) \oplus Q_1^T]]
\end{array} \right\}
\]  
(34)

Equation (34) is the proposed fast 2-D 8 × 8 inverse integer transform algorithm. Table 2 shows the comparisons of computational complexity for fast 2-D 8 × 8 forward or inverse integer transform. The computational complexity of proposed fast forward and inverse 2-D 8 × 8 transform methods is equivalent to that of the fast computations in [6], [7] with row-column based methods.

Figure 3 shows the symbolic data flow diagram and pipelined phases of the proposed fast 2-D 8 × 8 forward transform in (31). In Fig. 3, the computational latency delays occupy 8 clock cycles, where the 1st stage needs one pipelined phase, the 2nd stage needs one pipelined phase, the 3rd stage needs 4 pipelined phases, and the 4th stage needs 2 pipelined phases. In the 3rd stage, two-level cascaded Q_1 computations are required, where each Q_1 computation needs two pipelined phases, which are similar to the 1-D case in Fig. 1. Based on the similar structure in (31), the latency delays of the fast 2-D 8 × 8 inverse integer transform in (34) also occupy 8 clock cycles. Figure 4 shows the symbolic data flow diagram and pipelined phases for the realization of (34). In architecture comparisons, the proposed fast 2-D integer transform designs require no transpose memory. The latency delay of our design is less than the previous row-column based architecture. Meanwhile, the throughput rate of the proposed architecture is 8 times increase, compared with that of the row-column based scheme. Table 3 shows the architecture comparisons of the 2-D 8 × 8 integer transforms.

**Table 2** Comparisons of computational complexity for 2-D 8 × 8 forward or inverse integer transform.

<table>
<thead>
<tr>
<th>2-D 8×8</th>
<th>Row-Column Method</th>
<th>Row-Column Method</th>
<th>Proposed Fast Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Without Fast Algorithm</td>
<td>With Fast Algorithm [6,7]</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>896</td>
<td>160</td>
<td>512</td>
</tr>
</tbody>
</table>

**Fig. 3** Symbolic data flow diagram of fast 2-D forward 8 × 8 integer transform with 8 pipelined phases.

**6. Conclusion**

The fast 2-D 8 × 8 forward and inverse integer transform algorithms are developed by utilizing matrix factorizations and operations. With Kronecker product and direct sum operations, the fast 2-D 8 × 8 integer transforms can be derived from the fast 1-D 8 × 8 integer transforms through matrix utilizations. In comparison with the row-column based architecture, the developed fast 2-D 8 × 8 forward and in-
in VLSI implementation, the proposed fast algorithms can achieve real-time H.264/AVC FRExt video signal processing.

Acknowledgement

The author would like to thank the anonymous reviewers, whose careful reviews and detailed comments help to improve the readability of this letter. This work was supported by the National Science Council, Taiwan, R.O.C., under Grant NSC95-2220-E-005-006.

References