ABSTRACT
Efficient implementation of bit-level matrix transpose operations using a new family of perfect shuffle instructions in a word-based micro-processor architecture is investigated. Using a tensor representation of the perfect shuffle index mapping, we show how bit matrix of arbitrary size may be transposed using a minimal sequence of shuffle and pack instructions, and prove the correctness of the code so generated. Specifically, a shuffle decomposition approach, and a block matrix transpose approach are proposed and the performance of these approaches are compared. Implemented with Texas Instruments C64X instruction set, significant performance enhancement is observed.

Index Terms— Bit matrix transpose, tensor, perfect shuffle

1. INTRODUCTION
With rapid proliferation of wireless communication standards, software defined radio (SDR) that leverages programmable and re-configurable processors to emulate many different wireless transceivers has been investigated intensively. A fundamental challenge of SDR development is the mismatch between the multiple data formats required in wireless standards and the single word-length in current micro-processor architecture. In particular, other than simple bit-wise logic operations, a word-based instruction set architecture has hard time to deal with bit serial operations such as channel coding, interleaving, scrambling or punctuation. Previously, we have developed an optimized approach to generate bit-parallel instruction sequence to realize a linear feedback shift register algorithm to facilitate efficient implementation of scrambler operation [1]. Theorems are provided to prove the correctness of instruction sequence generated using the proposed method. In [2], a preliminary result is reported that will implement the block interleaving operation in the new LTE-A standard [3]. However, the result is ad hoc and the method does not scale easily to different block sizes. In this work, we provide a theoretical justification of the correctness of the interleaving code so generated; as well as systematic approaches to realize block interleaving operation of arbitrary block size. We recognize the mathematical equivalent of the block interleaving of bit sequence operation is the bit matrix transpose operation. Hence, bit matrix transpose of varying matrix size will be the goal of code generation.

Although matrix transpose has been widely studied on various level of scale [4], [5], the bit matrix transpose operation poses distinct obstacles. General word-level matrix transpose can be easily indexed, and that matrix transpose amounts to address mapping. However, for bit-matrix transpose, there is no efficient way to directly access a single bit via current word-based instruction architecture. Such format mismatch between the basic processor unit, word, and the bit element often causes inefficient bit matrix transpose on a word-level processor [6], [7]. A naive solution that simply stores each bit in an ANSI char and process one bit at a time would incur a huge waste in both processing speed and data storage. To address the format mismatch, most processors have implemented instructions specially designed for bit-level operations. Some researchers have proposed new instruction set extensions for bit-level operations in encryption and media streaming [8].

This work is motivated by the availability of a family of perfect shuffling family of instructions in micro-processors designed for wireless communication applications, such as Texas Instrument C64X instruction set [9]. In C64X, four instructions SHFL, DEAL, PACK2, and PACKH2 belong to this family. The focus of our investigation in this work is on the underlying mathematics that relate the functions provided by these bit-level perfect shuffle (SHFL) and inverse perfect shuffle (DEAL) instructions to the bit-matrix transpose problem. The packing instructions PACK2 and PACKH2 are handy auxiliary instructions that facilitate efficient data format translations. These instructions generally take advantage of the regular data-accessing pattern in the bit matrix transpose. However, current design lacks the mapping methodology to the current instructions provided. This research proposed to use tensor product and perfect shuffling to represent the bit-matrix transpose. The background of tensor product and perfect shuffling is introduced in Sec. 2. The instructions supported by our targeting platform Texas Instruments’ C64X are presented in Sec. 3. We also introduce two bit-matrix transpose algorithms and discuss their implementation detail in Sec. 4. The results and the conclusion are shown in Sec. 5.
Table 1. A $S_{2,4}$-shuffle index mapping

<table>
<thead>
<tr>
<th>$b$</th>
<th>[2, 4]</th>
<th>[2, 4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$i_1, i_0$</td>
<td>$i_0, i_1$</td>
</tr>
<tr>
<td>0</td>
<td>0, 0</td>
<td>0, 0</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>0, 2</td>
<td>1, 0</td>
</tr>
<tr>
<td>3</td>
<td>0, 3</td>
<td>1, 1</td>
</tr>
<tr>
<td>4</td>
<td>1, 0</td>
<td>2, 0</td>
</tr>
<tr>
<td>5</td>
<td>1, 1</td>
<td>2, 1</td>
</tr>
<tr>
<td>6</td>
<td>1, 2</td>
<td>3, 0</td>
</tr>
<tr>
<td>7</td>
<td>1, 3</td>
<td>3, 1</td>
</tr>
</tbody>
</table>

2. PRELIMINARIES

In this section, we introduce the background knowledge of perfect shuffling [10] and Kronecker product [11]. The definition, representation, and a few illustrating examples are given for both concepts.

2.1. Perfect Shuffling

We first define perfect shuffling—$S(b_1, b_0)$ as the operation on $b_0 b_1$ elements that transposes a $b_1 \times b_0$ matrix into a $b_0 \times b_1$ matrix. The shuffle maps current $i$-th element to a new index $j$ after permutation; denoting the coefficients $i_1$ and $i_0$ for $b_1$ and $b_0$ respectively, one has:

$$S(b_1, b_0) : i = i_1 b_0 + i_0 \Rightarrow j = i_0 b_1 + i_1$$ (1)

If $b_1$ and $b_0$ are both powers of two and denoted as $2^{k_0}$ and $2^{k_1}$, one has:

$$S_{b_1, b_0} : i = i_1 2^{k_0} + i_0 \Rightarrow j = i_0 2^{k_1} + i_1$$ (2)

which exchanges the $k_0$ and $k_1$ bits of the index as:

$$S_{b_1, b_0} : i = i_1 |i_0 \Rightarrow j = i_0 |i_1$$ (3)

For example, $S_{2,4}$ will transform a $2 \times 4$ matrix into a $4 \times 2$ matrix. One can obtain the transposed matrix $w$ given an input vector $v = [v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7]^T$:

$$w = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix}$$ (4)

While $v$ in (4) appears to be a one-dimension vector, it can be viewed as a $2 \times 4$ matrix stored in the register by row-order; similarly, $w$ is a $4 \times 2$ matrix of the same storing methodology:

$$v = \begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \end{bmatrix}, \quad w = \begin{bmatrix} v_0 & v_4 \\ v_1 & v_5 \\ v_2 & v_6 \\ v_3 & v_7 \end{bmatrix}.$$ 

One can also verify (3) with Table 1 that the index $j$ can be obtained by exchanging the first bit and the two least significant bits.

Moreover, for two consecutive operations, $w = S_2 S_1 v$, the $S_1$ matrix adjacent to $v$ will be applied first before $S_2$. That is, the element will be permuted by $S_1$ first before $S_2$.

2.2. The Kronecker product

Definition 1 The Kronecker product $M = M_1 \otimes M_0$ of two matrices—an $r_0 \times c_0$ matrix $M_0$ and an $r_1 \times c_1$ matrix $M_1$, is a $r_1 r_0 \times c_1 c_0$ matrix. An example best illustrates the Kronecker product:

**Example 1.** Consider two $2 \times 2$ matrix $A$ and $B$ (i.e. $r_0 = c_0 = r_1 = c_1 = 2$):

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}, \quad B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

, then the Kronecker Product $C = A \otimes B$ is a $4 \times 4$ matrix:

$$C = \begin{bmatrix} a_{00} b_{00} & a_{00} b_{01} & a_{01} b_{00} & a_{01} b_{01} \\ a_{00} b_{10} & a_{00} b_{11} & a_{01} b_{10} & a_{01} b_{11} \\ a_{10} b_{00} & a_{10} b_{01} & a_{11} b_{00} & a_{11} b_{01} \\ a_{10} b_{10} & a_{10} b_{11} & a_{11} b_{10} & a_{11} b_{11} \end{bmatrix}$$

In this work, the two most frequently used extensions are product of $M$ with identity matrix $I$; given a size $b$ identity matrix $I_b$, one has:

$$w = (I_b \otimes M) v$$ (5)

$$w = (M \otimes I_b) v$$ (6)

Analyzing (5), the resulting matrix is a block diagonal matrix as:

$$I_b \otimes M = \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \ddots & \cdot & \cdot \\ \cdot & \cdot & \ddots & \cdot \\ \cdot & \cdot & \cdot & M \end{bmatrix}_{b \times b}$$ (7)

The main diagonal are block matrices $M$, and the off-diagonal elements are zeros.

Furthermore, assume $M$ in (5) is $S_{b_1,b_0}$ and denote the identity matrix size now as $b_2$, $(I_{b_2} \otimes M)$ can then be viewed as dividing $w$ into $b_2$ disjoint sub-vectors, and transpose each sub-vector with $M$. We therefore reformulate the index mapping in (1) for $(I_{b_2} \otimes S_{b_1,b_0})$ as:

$$I_{b_2} \otimes S_{b_1,b_0} : i = i_2 b_2 + i_1 b_0 + i_0 \Rightarrow j = i_2 b_1 b_0 + i_0 b_1 + i_1$$ (8)

Similarly, in (6), the equivalent resulting matrix, $M \otimes I_b$, from $w$ to $v$ is:

$$(M \otimes I_b) = \begin{bmatrix} m_{0,0} I_b & \cdots & m_{0,c-1} I_b \\ \vdots & \ddots & \vdots \\ m_{r-1,0} I_b & \cdots & m_{r-1,c-1} I_b \end{bmatrix}$$ (9)
Following our previous assumption, let $M = S_{b1,b0}$ and denote the identity matrix’s size as $b_2$. The resulting matrix from (9) can be viewed as a block-wise transposition with $M$; each element is now $I_{b2}$. In other words, we are performing $M$ in a block-based permutation, and each block has size equals $b_2 \times b_2$. The index mapping for $(S_{b1,b0} \otimes I_{b2})$ is then:

$$S_{b1,b0} \otimes I_{b2} : i = (i_1 b_0 + i_0) b_2 + i_2 \rightarrow j = (i_0 b_1 + i_1) b_2 + i_2$$  

(10)

Moreover, if $b_2, b_1$ and $b_0$ are power of two, and are represented as $2^{k_2}, 2^{k_1}$ and $2^{k_0}$ respectively, the mapping becomes:

$$I_{b2} \otimes S_{b1,b0} : i = i_2 2^{k_1+k_0} + i_1 2^{k_0} + i_0 
\rightarrow j = i_2 2^{k_1+k_0} + i_0 2^{k_1} + i_1$$  

(11)

$$S_{b1,b0} \otimes I_{b2} : i = i_1 2^{k_0+k_2} + i_0 2^{k_2} + i_2 
\rightarrow j = i_0 2^{k_1+k_2} + i_1 2^{k_2} + i_2$$  

(12)

One may easily verify that the indexes $i$ and $j$ are $(k_2 + k_1 + k_0)$ bits and is composed of three slots of size $k_2$, $k_1$ and $k_0$ respectively. Operation $I_{b2} \otimes S_{b1,b0}$ exchanges the $k_1$ and $k_0$ bits of the last significant bits:

$$I_{b2} \otimes S_{b1,b0} : i = i_2 |i_1|_0 \rightarrow j = i_2 |i_0|_1$$  

(13)

Similarly, $S_{b1,b0} \otimes I_{b2}$ exchanges the $k_1$ and $k_0$ bits in most significant bits:

$$S_{b1,b0} \otimes I_{b2} : i = i_1 |i_0|_2 \rightarrow j = i_0 |i_1|_2$$  

(14)

3. THE INSTRUCTIONS OF TEXAS INSTRUMENTS C64X

In this section, we formulate the instructions related to bitmatrix transpose in Texas Instrument C64x digital signal processor following the definition of tensor product and perfect shuffling. We introduce their functions and representations by tensor product and perfect shuffling. The instructions we survey in this work are SHFL, DEAL, PACK2 and PACKH2 [9]; the representation can be obtained and extended to similar instructions on various processors.

3.1. Perfect Shuffling Instructions

In TI’s DSP, the perfect shuffling instruction is SHFL, and the inverse perfect shuffling instruction is DEAL. In TI’s c64 DSP, the SHFL instructions separate the word as two half-words, and interleaved them. The DEAL instruction collects the odd bits in the first half-word and the even bits in the other. The functions of both instructions are shown in Fig. 1.

The instruction SHFL can be formulated as:

$$\text{SHFL}(s) = S_{b2,16}.$$  

(15)

The matrix $S_{2,16}$ will generate a $16 \times 2$ matrix:

$$\begin{bmatrix}
0 & 1 & 2 & \cdots & 13 & 14 & 15 
16 & 17 & 18 & \cdots & 29 & 30 & 31
\end{bmatrix}^T$$  

(16)

which is indeed the output pattern of _SHFL(). Similarly, the instruction DEAL can be formulated as:

$$\text{DEAL}() = S_{16,2}.$$  

(17)

3.2. Packing Instructions in TI C64x

The alignment instruction PACK2 has two 32-bit input vectors, and it generates one 32-bit vector from the lower half of both input vectors; similarly, PACKH2 combines the upper halves. The functions of PACK2 and PACKH2 are shown in Fig. 2. Since we do not discard any elements during matrix

$$\text{PACK2}(s) = S_{b1,b0} \otimes I_{b2}.$$  

$$\text{PACKH2}(s) = S_{b1,b0} \otimes I_{b2}.$$  

(18)

Fig. 2. Packing instructions in TI C64x DSP.
We then show that the origin elements from the two 32-bit registers are:

\[
    w = [v_0 \cdots v_{15} | v_{32} \cdots v_{47} | v_{16} \cdots v_{31} | v_{48} \cdots v_{63}]^T \tag{21}
\]

The first 32 elements, \(w[0:31]\), come from the upper halves of the registers \(B_0\) and \(B_2\), and thus they are the output of \textsc{Pack2}; in contrast, the last 32 elements come from the lower halves of the two registers \(B_1\) and \(B_3\), and are therefore the output of \textsc{Pack2}. Note the constraint of \(S_{2,2}\) is that the two input (and two output) registers need to be adjacent.

One can easily extend the permutation to more complex pattern than \(S_{2,2}\). The following example illustrates the transposition of a 32 \(\times\) 32 matrix. The elements are stored in thirty-two registers of size 32-bits and each register contain two 16-bit sub-vectors \(B\).

**Example 2.** Performs transpose \(S_{32,2}\) to \( \{B_i \mid 0 \leq i \leq 63\} \).

\[
    v = [B_0 \ B_1 \ B_2 \ \cdots \ B_{61} \ B_{62} \ B_{63}]^T \tag{22}
\]

Note \( \{B_i \mid i \in \text{odd}\} \) are the first halves (e.g. \(w[0:15]\)), and \( \{B_i \mid i \in \text{even}\} \) are the second halves (e.g. \(w[16:31]\)) of the input registers.

\[
    w = (S_{32,2} \otimes I_{16}) \ v
    = S_{32,2} [B_0 \ B_1 \ B_2 \ \cdots \ B_{61} \ B_{62} \ B_{63}] \\
    = [B_0 \ B_2 \ B_4 \ \cdots \ B_{58} \ B_{60} \ B_{62}] \\
    \tag{23}
\]

Therefore, \(S_{32,2}\) is essentially collecting all first-half-words in the first sixteen registers, and the second-half-words in the rest.

### 4. MATRIX TRANSPOSE IMPLEMENTATION

In this section, we derive the tensor representation and map them to the instructions of TI c64 DSP [9]. The first algorithm is the perfect shuffling algorithm in [1], and the second is the block-based transformation.

#### 4.1. Factorization of Shuffles

The mapping process of shuffle is not straightforward. In general, the target bit by bit-matrix is much larger than what one instruction can handle. Thus, one needs to de-factor the targeting transpose into a few supported shuffling instructions.

**Property 1 (Paraphrase Theorem 5 in [11])** A perfect shuffle of a large column size can be de-factored into a sequence of two shuffles of smaller column size:

\[
    S_{b_2,b_1,b_0} = S_{b_2,b_0,b_1} \cdot S_{b_2,b_1,b_0} = S_{b_2,b_0,b_1} \cdot S_{b_2,b_0,b_1} \tag{24}
\]

For example, \(S_{2,4} = S_{4,2}S_{4,2}\) according to (24). Transposed matrix \(w\) given an input vector \(S_{2,4}v\) in Sec. 2.1:

\[
    S_{2,4}v = S_{4,2}S_{4,2}v = S_{4,2} \begin{bmatrix} v_0 & v_2 \\ v_4 & v_6 \\ v_1 & v_3 \\ v_5 & v_7 \end{bmatrix} = \begin{bmatrix} v_0 & v_4 & v_1 & v_5 \\ v_2 & v_6 & v_3 & v_7 \end{bmatrix}
\]

**Property 2 (Paraphrase Theorem 5 in [11])** A perfect shuffle can be de-factored into two Kronecker products of smaller shuffles and identity matrices:

\[
    S_{b_2,b_0 \otimes I_{b_1}} \cdot (I_{b_2} \otimes S_{b_1,b_0})
\]

A simple illustrating example that de-factorizes \(S_{4,2}\):

\[
    S_{4,2} = (S_{2,2} \otimes I_{b_2}) \cdot (I_{b_2} \otimes S_{2,2})
\]

\[
    (S_{2,2} \otimes I_{b_2}) \begin{bmatrix} v_0 & v_2 \\ v_4 & v_6 \\ v_1 & v_3 \\ v_5 & v_7 \end{bmatrix} = \begin{bmatrix} v_0 & v_2 \\ v_4 & v_6 \\ v_1 & v_3 \\ v_5 & v_7 \end{bmatrix}
\]

4.1.1. Perfect Shuffling Implementation

Consider the case of transposing 1024 bits which was stored in thirty-two 32-bit registers; the resulting permutation matrix is \(S_{32,32}\). Following (24), one can factor \(S_{32,32}\) as:

\[
    S_{32,32} = S_{32,16 \times 2} = S_{64,16} \cdot S_{32,2} \tag{27}
\]

In (27), one can substitute the first term iteratively with (24) as:

\[
    S_{64,16} = S_{64,4 \times 4} = S_{256,4} \cdot S_{256,4} = S_{256,2 \times 2} \cdot S_{256,2 \times 2} = S_{512,2} \cdot S_{512,2} \cdot S_{512,2} \cdot S_{512,2} \tag{28}
\]

We then factorize \(S_{512,2}\) with (24) as:

\[
    S_{512,2} = S_{32,16 \times 2} = (S_{32,2} \otimes I_{16}) \cdot (I_{32} \otimes S_{16,2}) \tag{29}
\]

Therefore, substituting (28) and (29) into (27), one can reformulate (27) as:

\[
    S_{32,32} = ((S_{32,2} \otimes I_{16}) \cdot (I_{32} \otimes S_{16,2}))^5 \tag{30}
\]

Recall the tensor representation of instructions \textsc{Deal} and \textsc{Pack2+Pack2} from (17) and (18):

\[
    \text{DEAL()} = S_{16,2} \\
    (\text{PACK2} + \text{PACK2})() = S_{2,2} \otimes I_{16}
\]

One can easily found in (30) that the first term reorders the half-words with \textsc{Pack2+Pack2} as in (23) and the second term performs 16 set of \textsc{Deal}. Thus, the equivalent operations of (30) can be represented in pseudo code as:
for i:=0 to i:=4{
    for j:=0 to j:=31
        DEAL(reg [j])
for k:=0 to k:=16{
    reg [k]=PACK2(reg [2k], reg [2k+1])
    reg [k+1]=PACK2(reg [2k], reg [2k+1])
}
}

Note the operations orders of the tensor product is right-to-left; therefore, DEAL is applied before PACK2+PACKH2 in (30).

4.2. Block-based Transpose

A common trick in matrix transpose is block-based transpose. Let the matrix be partition into \( r \times c \) blocks and each block size is \( r_0 \times c_0 \) (i.e. the original matrix is \( r_0 \times c_0 \) in size.) The block-based transpose can be illustrated as follows:

\[
B_{ij} = \begin{bmatrix}
B_{0,0} & \cdots & B_{0,c-1} \\
B_{1,0} & \cdots & B_{1,c-1} \\
\vdots & \ddots & \vdots \\
B_{r-1,0} & \cdots & B_{r-1,c-1}
\end{bmatrix}
\]

where \( B_{ij} \) is applied before \( i; j \).

Example 3, transpose a \( 32 \times 32 \) matrix \( A \) with four \( 16 \times 16 \) sub-blocks.

\[
A_T = \begin{bmatrix}
B_{00} & B_{01} \\
B_{10} & B_{11}
\end{bmatrix}
\]

Proof: Applying the index mapping equations in (13) and (14), we consider the index as \( 2k+2 \) bits going through the four operations.

\[
(J_2 \otimes S_{2^k,2} \otimes I_{2^k}) \text{ exchanges the } k\text{-bit and } 1\text{-bit slots (under-scored) in the middle:}
\]

\[
i = 1|1|1|k \rightarrow j = 1|1|1|k
\]

\[
(4) \otimes S_{2^k,2^k} \text{ exchanges the two } k\text{-bit slots in the least significant bits:}
\]

\[
i = 2|1|k|k \rightarrow j = 2|1|k|k
\]

\[
(S_{2^k,2^k} \otimes I_{2^k}) \text{ exchanges the two } 1\text{-bit slots:}
\]

\[
i = 1|1|1|k \rightarrow j = 1|1|1|k
\]

\[
(S_{2^k,2} \otimes I_{2^k}) \text{ exchanges the } k\text{-bit and } 1\text{-bit slots in the middle:}
\]

\[
i = 1|1|1|k \rightarrow j = 1|1|1|k
\]

Combining (35) to (38), one has:

\[
i = 1|1|1|k \rightarrow j = 1|1|1|k
\]
5. SIMULATION RESULTS AND DISCUSSION

The naïve, perfect shuffling and block-based transpose methods for $8 \times 8$ to $64 \times 64$ power-of-two square matrices transpose are implemented. The codes were written in ANSI C and the intrinsics supported as introduced in Sec. 3. We then built and emulated them on Texas Instruments Inc.’s cycle true simulator [12]. The simulation results are shown in Table 2. We compare the cycles required by the three methods, and show the speed-up compared with the naïve method.

Obviously, the naïve implementation is the slowest because it takes multiple cycles to process one bit. The naïve method is at least 9 times slower than the other methods. Compared with the block-based transpose, the perfect-shuffling implementation is 72% faster. The efficient mapped perfect-shuffling algorithm fully take advantage of the device and thus outperforms the general method.

Recall $32 \times 32$ transpose for perfect shuffling in (30):

$$S_{32,32} = \left( (S_{16,2} \otimes I_{16}) \cdot (I_{32} \otimes S_{16,2}) \right)^5.$$

From the equation, we can tell there are five outer loops, and each loop performs 32 DEAL and 32 sets of PACK+PACK2. Therefore, the estimated complexity is $5 \times 32 \times (1 + 2) = 480$ instruction cycles plus looping cost and other overhead.

The block-based transpose matrix in (43) is:

$$(I_2 \otimes S_{16,2} \otimes I_{16}) (S_{2,2} \otimes I_{256}) (I_4 \otimes S_{16,16}) (I_2 \otimes S_{16,2} \otimes I_{16}).$$

The operation required 2 × 32 sets of PACK+PACK2, and 32 times of register re-ordering, and 4 loops of $16 \times 16$ sub-block transpose. Estimated with a similar method, the $16 \times 16$ matrix transpose can be performed in 16 packing instruction pairs, 8 register re-ordering, and 4 loops of $8 \times 8$ matrix transpose. The $8 \times 8$ matrix transpose is a fundamental transpose and can be performed in 18 instructions.

<table>
<thead>
<tr>
<th>^</th>
<th>Naive cycle</th>
<th>Shuffle cycle</th>
<th>Speed-up (X)</th>
<th>Block-based cycle</th>
<th>Speed-up (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 × 8</td>
<td>299</td>
<td>32</td>
<td>9.37</td>
<td>32</td>
<td>9.75</td>
</tr>
<tr>
<td>16 × 16</td>
<td>1833</td>
<td>202</td>
<td>9.07</td>
<td>631</td>
<td>3.12</td>
</tr>
<tr>
<td>32 × 32</td>
<td>36868</td>
<td>1749</td>
<td>21.49</td>
<td>4591</td>
<td>8.03</td>
</tr>
<tr>
<td>64 × 64</td>
<td>159066</td>
<td>15767</td>
<td>10.1</td>
<td>29923</td>
<td>5.3</td>
</tr>
</tbody>
</table>

Table 2. Matrix Transpose Performance on TI-C6416

The operation required 2 × 32 sets of PACK+PACK2, and 32 times of register re-ordering, and 4 loops of $16 \times 16$ sub-block transpose. Estimated with a similar method, the $16 \times 16$ matrix transpose can be performed in 16 packing instruction pairs, 8 register re-ordering, and 4 loops of $8 \times 8$ matrix transpose. The $8 \times 8$ matrix transpose is a fundamental transpose and can be performed in 18 instructions. Thus, the total estimated instructions count is: $32 \times (2 \times 2 + 1) + 4 \times (16 \times (2 \times 2 + 1) + 4 \times 18) = 608$ instruction cycles plus looping cost and other overhead. The estimation shows the block-based implementation is 26% more instruction cycles than perfect shuffling counter part. In [2], the authors applied an in-house tool to emulate the behavior of bit-level instructions to verify the implementation; the correct results are then fed into the emulator for performance comparison. With our method, one can then easily verify the correctness and justify the design decision based on the tensor product representation.

The emulated results show the performance difference between the two methods is larger. This deviation comes mainly from the high overhead of the block-base matrix transpose; the looping and sub-function call produce serious overhead. In contrast, the perfect shuffling implementation is simple and therefore compact in code size. One can also recognize this fact easily based on the representation of (30) and (43).

In this work, we propose a method to design the square matrix transpose algorithm for any devices supports shuffling operations. Our method converges the targeting transpose and the operations supported into perfect shuffling and tensor product representation. The target transpose is then de-factored into the operations supported (both in tensor product).

We found the method efficient in both algorithm verification and performance estimating. Our results demonstrates the powerfulness of the algorithm-architecture mapping design. This work currently considers only square matrices of sizes power of two; some future works on rectangular matrix transpose will be an interesting next step.

6. REFERENCES


