Shuffle Memory System

Kichul Kim
School of Electrical Engineering
University of Seoul
Dongdaemun-Gu, Seoul 130-743, Korea
kkim@uoscc.uos.ac.kr

Abstract

This paper proposes a new memory system called shuffle memory. The shuffle memory is a generalization of transposition memory that has been widely used in 2-D Discrete Cosine Transform and Discrete Fourier Transform. The shuffle memory is the first memory system that receives $M \times N$ inputs in row major while providing $M \times N$ outputs in column major order using only $M \times N$ memory space. Shuffle memory can provide memory efficient architectures for separable 2-D transforms and separable 2-D filter banks.

1. Introduction

Separable 2-D transforms and separable 2-D filter banks frequently appear in scientific computations and digital signal processing [1, 3, 4, 6, 7, 11]. Fourier Transform, Walsh transform, Hadamard Transform, and Discrete Cosine Transform are examples of popular separable transforms [6]. An efficient hardware implementation is known to exist for this class of transforms [6]. The hardware implementation for those transforms consists of two 1-D transform units and a memory system. The memory system is between two 1-D transform units and performs proper data reordering. For $M \times N$ 2-D separable transforms, when $M = N$, an efficient memory system, called transposition memory [5], is widely used for data reordering. When $M \neq N$, no efficient memory system is known. Double buffers are commonly used to achieve 100% utilization of 1-D transform units. The cost of a double buffer becomes very high as M and N become large.

In the rest of this section, 2-D DFT(Discrete Fourier Transform) will be used as an example of separable 2-D transform. The explanation will apply to all separable transforms and filter banks with minor changes on equations.

2-D DFT is a separable function and can be computed by consecutive two 1-D DFT computations [6]. An $M \times N$ 2-D DFT can be represented as in the following equation.

$$F(u,v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi ux/M} e^{-j2\pi vy/N} \tag{1}$$

The above equation can be represented as follows,

$$F(u,v) = \frac{1}{M} \sum_{x=0}^{M-1} F(x,v) e^{-j2\pi ux/M} \tag{2}$$

$$F(x,v) = \frac{1}{N} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi vy/N} \tag{3}$$

Equations (2) and (3) represent M-point 1-D DFT and N-point 1-D DFT respectively. By equations (2) and (3), $F(u,v)$ can be computed by application of N-point 1-D DFT on each row of the input array followed by application of M-point 1-D DFT on each column of the intermediate result. Figure 1 shows the procedure of the computation of $M \times N$ 2-D DFT.

A hardware implementation of 2-D DFT is shown in figure 2. In figure 2, input array is provided to N-point 1-D DFT in row major order. The result of N-point 1-D DFT of each row is stored in a memory. When all rows of the input array are processed, the intermediate result is provided to M-point 1-D DFT in column major order. The result of 2-D DFT is obtained from M-point 1-D DFT in column major order. In figure 2, the cost and the performance of 2-D DFT processor heavily depend on the structure and the cost of the memory. When the memory can not perform read operation (for the M-point 1-D DFT) and write operation (for the N-point 1-D DFT) at the same time, the utilization of 1-D DFT units becomes 50% because one of 1-D DFT units must idle until the memory becomes available. To avoid low utilization of 1-D DFT units, one can chose double buffers in 2-D DFT implementation, where two memories of size $M \times N$ are used. While one memory receives inputs in row major order the other memory provides outputs in column major order. When receiving an array and providing an ar-

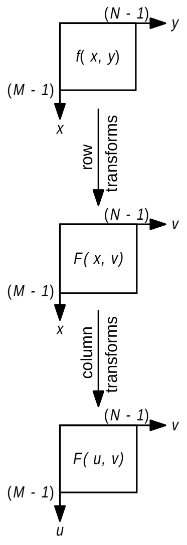


Figure 1. Computation of 2-D DFT with two consecutive 1-D DFT computations

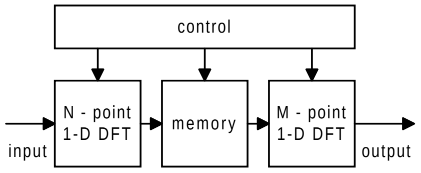
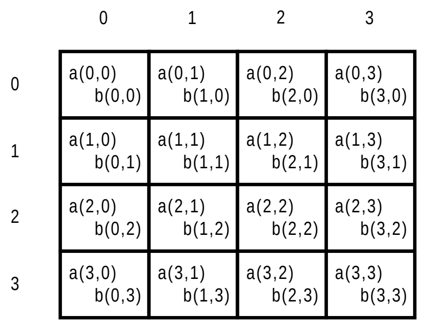


Figure 2. The structure of 2-D DFT processor

ray are finished at the same time, two memories exchange their roles. This way, two 1-D DFT units can work without waiting, achieving 100% utilization. When M and N become large, the cost of memory in a double buffer becomes very high.

When $M = N$, an efficient memory system called transposition memory can be used in 2-D DFT implementations [5]. Because of it's low memory requirement and simple structure, transposition memory has been widely used in 2-D Discrete Cosine Transform implementations. Figure 3 shows the storage scheme of 4×4 transposition memory. In figure 3, 1-D memory space is viewed as 2-dimensional address space. The digits appearing on left of each row represent the most significant digits of memory addresses. The digits appearing on top of each column represent the least significant digits of memory addresses. Array element $a(1, 2)$ is stored in address $12_{(4)}$, which is $6_{(10)}$. Two different storage schemes are used in figure 3. Array A is stored in row major order, and array B is stored in column



	0	1	2	3
0	a(0,0) b(0,0)	a(0,1) b(1,0)	a(0,2) b(2,0)	a(0,3) b(3,0)
1	a(1,0) b(0,1)	a(1,1) b(1,1)	a(1,2) b(2,1)	a(1,3) b(3,1)
2	a(2,0) b(0,2)	a(2,1) b(1,2)	a(2,2) b(2,2)	a(2,3) b(3,2)
3	a(3,0) b(0,3)	a(3,1) b(1,3)	a(3,2) b(2,3)	a(3,3) b(3,3)

Figure 3. Storage method for 4x4 transposition memory

major order. Note that array A is stored in row major order but it can be retrieved in column major order. We use row/column major order to specify both received/provided order and storage scheme (how an array is stored in address space). On the same line, array B can be received in row major order while it is stored in column major order.

Let's assume array A enters 2-D DFT unit first and array B follows array A. Transposition memory operates as follows:

1. The result of the first 1-D DFT on array A is stored in transposition memory in row major order (Array A is received in row major order by transposition memory).

2. When whole array A is stored in the memory, array A is provided to the second 1-D DFT unit in column major order. At the same time, the result of the first 1-D DFT on array B is stored in transposition memory in column major order.

Notice that the address sequence needed for retrieving array A in column major order is (0, 4, 8, 12, ..., 15) since array A is stored in row major order. The address sequence needed for storing array B in column major order is (0, 4, 8, 12, ..., 15). Since the two sequences are identical, an element of A is read in an address and an element of B can be written at the same address. This operation is called 'read-then-write at the same address operation'. 'Read-then-write at the same address operation' is the reason for low memory requirement of transposition memories. After all elements of A are read and all elements of B are written, new array can be written into transposition memory using the same storage scheme for array A.

Figure 4 shows a 4×4 transposition memory. The size of the memory used in a transposition memory is the half of a double buffer ($N \times N$). The structure of a transposition memory is very simple leading to easy implementation and

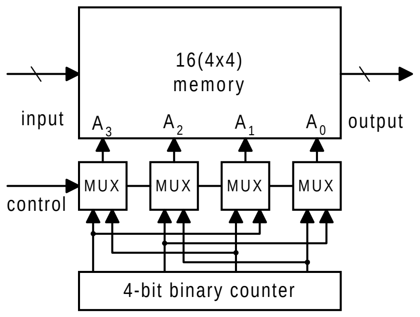


Figure 4. A 4x4 transposition memory

popular use. However, transposition memory can be used only when M and N are equal. It is highly desirable to have a memory system that provides 'read-then-write at the same address operation' even when M and N are not equal.

This paper proposes new memory system called shuffle memory system. Shuffle memory receives $M \times N$ inputs in row major while providing $M \times N$ outputs in column major order using only $M \times N$ memory space. The shuffle memory system can provide memory efficient architectures for separable 2-D transforms and separable 2-D filter banks.

The rest of the paper is organized as follows: Section 2 introduces shuffle memory. Section 3 introduces an efficient structure for shuffle memories of size $2^m \times 2^n$. Section 4 shows shuffle memories can provide memory efficient architectures for DFT processors. Finally, section 5 gives the conclusion of the paper.

2. Shuffle memories

A q-shuffle on qc objects is denoted by S_{qc} and is defined as follows [7]:

$$S_{qc}(i) = \left(qi + \left\lfloor \frac{i}{c} \right\rfloor\right) \bmod qc \qquad (4)$$

A 3-shuffle on 12 objects is shown in figure 5 as an example.

A shuffle memory of size $M \times N$ has the following characteristics:

1. A shuffle memory receives an $M \times N$ array in row major order while providing an $M \times N$ array in column major order.

2. A shuffle memory supports 'read-then-write at the same address operation'

'Read-then-write at the same address operation' is essential for the minimal use of memory. It can be easily shown that the permutation provided by shuffle memory is a q-shuffle

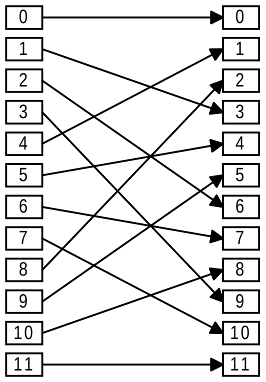


Figure 5. 3-shuffle on 12 objects

from input to output. For example, suppose a shuffle memory of size 3×4 accepts a 3×4 array in row major order and outputs the array in column major order. An element $a(i, j)$ is the $(4i + j)$-th element that enters the shuffle memory and the $(i + 3j)$-th element that exits the shuffle memory. The permutation provided by the shuffle memory is:

$$\begin{aligned} P_{SM}(4i + j) &= i + 3j \\ &= \left((4i + j) * 3 + \left\lfloor \frac{4i + j}{4} \right\rfloor\right) \bmod 12 \\ &= S_{3*4}(4i + j) \end{aligned}$$

In general, the permutation generated by a shuffle memory of size $M \times N$ is a M-shuffle.

It can be shown that the permutation generated by a transposition memory is a special case of q-shuffle. Assume that a transposition memory of size $N \times N$ accepts an $N \times N$ array in row major order and outputs the array in column major order. Let P_{TM} denote the permutation from input to output. An element $a(i, j)$ is the $(iN + j)$-th element that enters the transposition memory in row major order and it is also the $(i + jN)$-th element that exits the transposition memory in column major order. P_{TM} can be represented as follows:

$$\begin{aligned} P_{TM}(iN + j) &= i + jN \\ &= \left((iN + j)N + \left\lfloor \frac{iN + j}{N} \right\rfloor\right) \bmod N^2 \\ &= S_{N*N}(iN + j) \end{aligned}$$

The above equation shows that P_{TM} is an N-shuffle on N^2 objects. Hence, shuffle memory is a generalization of transposition memory.

A shuffle memory consists of a data RAM and an address generator. Let $G_0, G_1, \ldots, G_n$ be address sequences provided by the address generator when $M \times N$ arrays $A_0, A_1, \ldots, A_n$ enter shuffle memory. G_i is the address sequence used when A_i enters shuffle memory in row major

G_0	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
G_1	(0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11)
G_2	(0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6, 11)
G_3	(0, 9, 7, 5, 3, 1, 10, 8, 6, 4, 2, 11)
G_4	(0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 11)

Figure 6. Address sequences used in 3x4 shuffle memory

order. Let g_i^{kN+l} be the $(kN+l)$-th element of G_i. Then g_i^{kN+l} is the address where $a_i(k,l)$ is stored. Since $a_i(k,l)$ is the $(k+lM)$-th element that exits shuffle memory in column major order, the following relation holds:

$$g_{i+1}^{k+lM} = g_i^{kN+l} \tag{5}$$

The above equation shows that shuffle memory uses q-shuffle in address generation as well as it uses q-shuffle for the permutation from input to output. Figure 6 shows the address sequences used in 3×4 shuffle memory. Sequence (0, 1, 2, ..., 11) is used as G_0. Sequence G_5 is identical to sequence G_0. It can be easily shown that G_i can be obtained by applying 3-shuffle on $G_{(i-1) \bmod 5}$.

Address generator of a shuffle memory should have a mechanism to perform q-shuffle on address sequences. A simple way to perform q-shuffle on address sequences is using address RAMs. Figure 7 shows the structure of a 3×4 shuffle memory. There are one data RAM and two address RAMs. Data RAM receives and provides intermediate data. In this example, we assume data size is 64-bit. Address RAMs provide addresses to data RAM alternatively. Address RAM ARAM0 is initialized to sequence G_0, *ie*, address i holds data i. When an address is provided to data RAM from ARAM0, the address is stored in ARAM1. A 12×4 ROM is used to provide the address for ARAM1. ROM is initialized to address sequence (0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 11), which is inverse 3-shuffle (4-shuffle). This way address sequence G_1 will be stored in ARAM1. When all 12 addresses are provided to data RAM, two address RAMs exchange their roles.

3. Shuffle memories of size $2^m \times 2^n$

When M and N are powers of two, there exists a more efficient structure for shuffle memories. Let $a_{m+n-1}a_{m+n-2}\cdots a_0$ represent the position of an address in an address sequence. 2^m-shuffle is expressed as follows:

$$\begin{aligned}&S(a_{m+n-1}a_{m+n-2}\cdots a_0)\\&= a_{n-1}a_{n-2}\cdots a_0 a_{m+n-1}a_{m+n-2}\cdots a_n\end{aligned}$$

The above equation shows 2^m-shuffle can be obtained by cyclicly shifting address bits. A $2^m \times 2^n$-shuffle

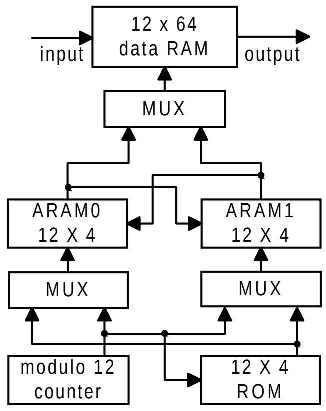


Figure 7. A 3x4 shuffle memory

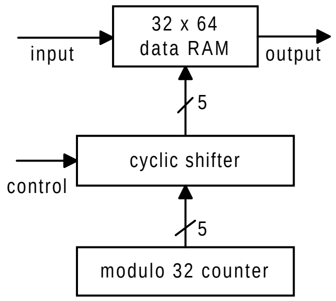


Figure 8. A 4x8 shuffle memory

memory can have at most m+n address sequences. Let $G_0, G_1, \ldots, G_{m+n-1}$ represent those address sequences. It is convenient to define G_0 as identity permutation. Addresses in address sequence G_i is obtained from addresses in G_0 by performing q-shuffle $(i-1)$ times. Hence we need general cyclic shifter to generate all required address sequences.

Figure 8 shows the structure of a 4×8 shuffle memory. It has a 32×64 data RAM and address generator. Address generator consists of a 5-bit binary counter and a general cyclic shifter. Control signal determines the amount of cyclic shift. Notice the simplicity of address generator. When the size of data RAM is large, the cost advantage of a shuffle memory over a double buffer is tremendous.

4. Application of shuffle memory on DFT

Shuffle memory can provide memory efficiency in DFT implementations. Figure 9 shows the structure of 3×4 2-D DFT with shuffle memory. By using shuffle memory, use of a double buffer is avoided.

European standard DVB-T(Digital Video Broadcasting-

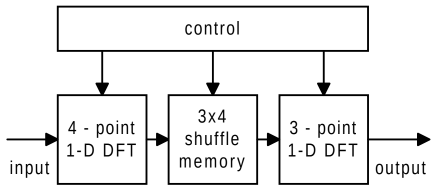


Figure 9. 3x4 DFT with shuffle memory

Terrestrial) uses OFDM(Orthogonal Frequency Division Multiplexing). 2K/8K DFT is the key element in DVB-T OFDM. In implementing 2K/8K DFT processor, memory requirement plays an important role. For example, 8K-point DFT can be economically decomposed into 4K-point DFT and 2-point DFT. Since 4096 is the square of 64, 4K-point DFT can be implemented very efficiently with two 64-point DFTs and a 64×64 transposition memory [11]. 64-point DFT can also be decomposed into two 8-point DFTs. In summary, 8K-point DFT can be implemented with four 8-point DFTs and one 2-point DFT. In this case, the same module can be used repeatedly and a VLSI design can benefit from modularity and regularity. Using transposition memories also greatly reduces memory requirement. However, this architecture can not be used in 2K/8K DFT since it can not provide 2K-point DFT operation. A new memory efficient architecture is strongly desired for 2K/8K DFT.

Figure 10 shows the architecture of a 2K/8K DFT using shuffle memories. 8K-point DFT is decomposed into 4-point DFT and 2K-point DFT. 2K-point DFT is again decomposed into 64-point DFT and 32-point DFT. In summary, the architecture uses three 8-point DFTs and two 4-point DFTs. It uses several kinds of shuffle memories. It should be noticed that memory requirement has been greatly reduced by using shuffle memory.

5. Conclusion

This paper proposed a new memory system called shuffle memory. The shuffle memory receives $M \times N$ input arrays in row major order and provides $M \times N$ output arrays in column major order without using a double buffer. The shuffle memory is a generalization of transposition memory. Shuffle memory can provide memory efficiency in implementing 2-D DCTs and 2-D DFTs. A design of 2K/8K DFT has been presented to show the usefulness of shuffle memories.

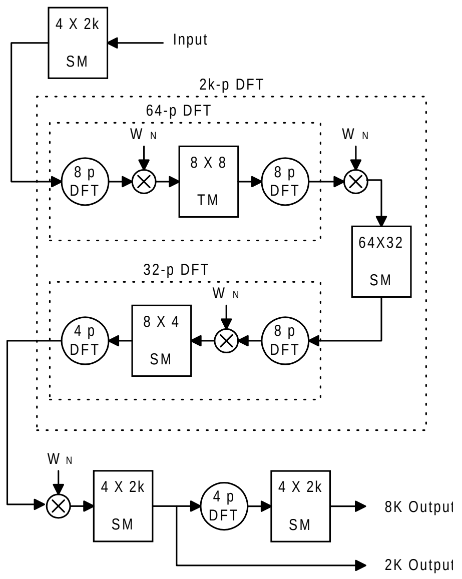


Figure 10. 2K/8K DFT for OFDM systems

References

[1] A. Artieri, S. Kritter, F. Jutand, and N. Demassieux. A one chip vlsi for real time two-dimensional discrete cosine transform. In *Intl. Symposium on Circuits and Systems*, pages 701–704, 1988.

[2] P. Budnik and D. J. Kuck. The organization and use of parallel memories. *IEEE Trans. Computers*, C-20(12):1566–1569, 1971.

[3] J. Carlach, P. Penard, and J. Sicre. Tcad: a 27 mhz 8x8 discrete cosine transform chip. In *IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 2429–2432, 1989.

[4] N. Demassieux, G. Concordel, J.-P. Durandeau, and F. Jutand. An optimized vlsi architecture for a multiformat discrete cosine transform. In *IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 547–550, 1987.

[5] L. J. D'Luna, W. A. Cook, R. M. Gudash, G. W. Brown, T. J. Tredwell, J. R. Fisher, and T. Tarn. An 8 x 8 discrete cosine transform chip with pixel rate clocks. In *The 3rd Annual IEEE ASIC Seminar and Exhibit*, pages P7–5.1 – P7–5.4, 1990.

[6] R. C. Gonzalez and P. Wintz. *Digital Image Processing*. Addison-Wesley Publishing Company, 2nd edition, 1987.

[7] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[8] F. Jutand, Z. J. Mou, and N. Demassieux. Dct architectures for hdtv. In *Intl. Symposium on Circuits and Systems*, pages 196–199, 1991.

[9] K. Kim and V. K. Prasanna. Latin squares for parallel array access. *IEEE Trans. Parallel and Distributed Systems*, 4(4):361–370, 1993.

[10] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. Computers*, C-24(12):1145–1155, 1975.

[11] A. V. Oppenfeim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice-Hall, 1989.