The Journal of Natural Science,
Pai Chai University, Korea
Volume 7: 27~35, 1995

알고리즘 분해를 이용한 2-D DCT의 효율적인 VLSI 구현

정재길
배재대학교 물리학과

An Efficient VLSI Implementation of the 2-D DCT with the Algorithm Decomposition

Jae G. Jeong
Dept. of Physics, Pai Chai University

This paper introduces a VLSI (Very Large Scale Integrated Circuit) implementation of the 2-D Discrete Cosine Transform (DCT) with an application to image and video coding. This implementation, which is based upon a state space model, uses both algorithm and data partitioning to achieve high efficiency. With this implementation, the amount of data transfers between the processing elements (PEs) are reduced and all the data transfers are limitted to be local. This system accepts the input as a progressively scanned data stream which reduces the hardware required for the input data control module. With proper ordering of computations, a matrix transposition between two matrix by matrix multiplications, which is required in many 2-D DCT systems based upon a row-column decomposition, can be also removed. The new implementation scheme makes it feasible to implement a single 2-D DCT VLSI chip which can be easily expanded for a larger 2-D DCT by cascading these chips.

정지영상이나 동영상 코딩에 적용되는 2-D DCT의 효율적인 VLSI 구현을 위한 방법을 제시하였다. 2차원 상태공간식에 근거한 알고리즘 및 데이타 분할기법을 활용하여 다중프로세서 구조에서 문제가 되는 프로세서간의 통신량을 크게 감축시켰으며, 모든 통신을 국부적(local)이 되도록 하였다. 순차 주사 방식의 영상데이타를 입력할 수 있도록 설계하여 입력장치에 소요되는 하드웨어를 최소화하였으며, 계산의 순서를 조정함으로써 일반적인 행·열 분할 방법을 사용하는 2-D DCT에서 필요로 하는 Transposition RAM을 제거하였다. 제안된 VLSI 구조는 실시간 one-chip 2-D DCT 프로세서의 구현을 가능하게 할 뿐 아니라, 병렬 또는 직렬 연결에 의하여 보다 빠른 2-D DCT 및 보다 큰 2-D DCT로 확장될 수 있다.

Keywords: 2-D DCT, VLSI architecture, Algorithm decomposition, Algorithm partition, Data partition

Ⅰ. INTRODUCTION

The 2-D discrete cosine transform (DCT) [1] is important because of its usage in image and video data compression [2]. For most practical image, the DCT comes closest to the Karhunen-Loève transform in its energy compaction properties as compared to other transforms [1,3]. The DCT does not possess the severe blocking artifact which is often a problem with the DFT [3]. In addition to this, the DCT can be generalized to the 2-D real separable unitary tranform. Therefore, the efficient implementation of this transform has potential for being useful in many applications.

During recent years, a lot of VLSI implementations of the 2-D DCT have been presented. In order to reduce the number of required multiplications, the realization of a 2-D DCT is accomplished by the use of vector-radix type structures which requires a fewer number of multiplications [4,5]. However, the vector-radix type approach often results in an irregular architecture and requires complicated routing which may require a large silicon area and a long design time.

Recently, Sun et al. [6] presented a single chip for the 16×16 2-D DCT based upon a row-column decomposition [7]. Even though this chip can process data with a sampling rate up to 14.3 MHz, it is still slow for many video processing applications involving large frames such as 1024×1024 pixels per frame or more. Several of these chips can implement a higher speed system by using data partitioning methods and the inherent parallel structure of 2-D DCT. However, this chip can not be easily used for a larger 2-D DCT implementation because it uses an internal RAM for a matrix transposition. In contrast, a 1-D DCT chip can be easily cascaded for larger 1-D DCT. It also can be used to implement a high speed 2-D DCT, but an external transposition RAM and several 1-D DCT chips are required [8].

In this paper, I present a 2-D DCT implementation scheme which can be applied for the real-time video compression system. This scheme can be used to implement a very high speed 2-D VLSI DCT chip which can be expanded for a larger 2-D DCT. I used an algorithm decomposition technique which is appropriate for a VLSI implementation of the 2-D DCT. The implementation is based on the data being partitioned in blocks of **M** rows rather than **M**×**M** rectangular blocks.

Ⅱ. 2-D DCT ALGORITHM

For the 2-D DCT, the relationship between an original image pixel u(m,n) and a transformed image pixel v(m,n) is expressed by [9]:

$$v(k,l)=\frac{2}{M}\sum_{m=0}^{M-1}\sum_{n=0}^{M-1}c(k,m)\,c(l,n)\,u(m,n)$$

$$\text{for } 0\le k\le M-1 \text{ and } 0\le l\le M-1 \qquad (1)$$

$$\text{where } c(i,j)=e(i)\cos\frac{i(2j+1)\pi}{2M}$$

$$\text{and} \quad e(i)=\begin{cases}\frac{1}{\sqrt{2}} & \text{for } i=0\\ 1 & \text{for otherwise}\end{cases}$$

The above equation can be converted to the following matrix form [10].

$$\mathbf{V}=\mathbf{AUA}^{T}$$

where **U** is the original image segment which is an M×M matrix, **V** is the transformed image segment which is an M×M matrix, and

$$A=\begin{bmatrix} \sqrt{\frac{2}{M}}\,c(0,0) & \sqrt{\frac{2}{M}}\,c(0,1) & \cdots & \sqrt{\frac{2}{M}}\,c(0,M-1) \\ \sqrt{\frac{2}{M}}\,c(1,0) & \sqrt{\frac{2}{M}}\,c(1,1) & \cdots & \sqrt{\frac{2}{M}}\,c(1,M-1) \\ \vdots & \vdots & \ddots & \vdots \\ \sqrt{\frac{2}{M}}\,c(M-1,0) & \sqrt{\frac{2}{M}}\,c(M-1,1) & \cdots & \sqrt{\frac{2}{M}}\,c(M-1,M-1) \end{bmatrix} \quad (3)$$

The matrix form as given by equation (2) can be rewritten as:

$$\mathbf{V}=\mathbf{A}[\mathbf{AU}^T]^T \quad (4)$$

which means that one column of $[\mathbf{AU}^T]$ is computed from the one row of $\mathbf{U}$ and one column of $\mathbf{V}$ is computed from one column of $[\mathbf{AU}^T]^T$.

When we perform the 2-D DCT for an image coding system, which is one of the major applications, we perform the 2-D DCT in the following way instead of performing the 2-D DCT on the whole image [2, 3]: Divide a given image frame into many segments, Transform each segment independently, Apply any compression and/or coding algorithm, Transmit or store segments, Inverse transform each segment independently, and Combine all segments into one frame. This approach makes the DCT and inverse DCT computationally efficient in terms of storage and processing speed [2, 11].

Many previously developed 2-D DCT implementations are based on the input data being an M×M block. Therefore, at least M-1 rows have to be stored before starting the transformation of the first segment for a system with a progressively scanned type input. The new implementation is based on the M×N data block which is M rows of the input image frame. With this modification, we can reduce the latency time involved in the data storage as well as simplify the input controller hardware without increasing the computational complexity.

In deriving the new implementation scheme, we first define the input image frame (**F**) and output image frame (**G**) in terms of the input image segment ($\mathbf{U}_{i,j}$) and output image segment ($\mathbf{V}_{i,j}$) as follows:

$$F=\begin{bmatrix} \mathbf{U}_{0,0} & \mathbf{U}_{1,0} & \mathbf{U}_{2,0} & \cdots & \mathbf{U}_{L-1,0} \\ \mathbf{U}_{0,1} & \mathbf{U}_{1,1} & \mathbf{U}_{2,1} & \cdots & \mathbf{U}_{L-1,1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{U}_{0,L-1} & \mathbf{U}_{1,L-1} & \mathbf{U}_{2,L-1} & & \mathbf{U}_{L-1,L-1} \end{bmatrix} \quad (5)$$

$$\mathbf{G}=\begin{bmatrix} \mathbf{V}_{0,0} & \mathbf{V}_{1,0} & \mathbf{V}_{2,0} & \cdots & \mathbf{V}_{L-1,0} \\ \mathbf{V}_{0,1} & \mathbf{V}_{1,1} & \mathbf{V}_{2,1} & \cdots & \mathbf{V}_{L-1,1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{V}_{0,L-1} & \mathbf{V}_{1,L-1} & \mathbf{V}_{2,L-1} & & \mathbf{V}_{L-1,L-1} \end{bmatrix} \quad (6)$$

The first M rows of **G** are obtained from the first M rows of **F**. Let a matrix with the first M rows of **F** be designated as $\mathbf{F}_1$, a matrix with the second M rows of **F** be designated as $\mathbf{F}_2$, and so on. The matrix with the first M rows of **G** be designated as $\mathbf{G}_1$, a matrix with the second M rows of **G** be designated as $\mathbf{G}_2$, and so on. Then $\mathbf{G}_1$ can be expressed as follows:

$$\begin{aligned} \mathbf{G_1} &= [[\mathbf{AU}_{0,0}\mathbf{A}^T]\,[\mathbf{AU}_{1,0}\mathbf{A}^T]\cdots[\mathbf{AU}_{L-1,0}\mathbf{A}^T]] \\ &= \mathbf{A}[[\mathbf{U}_{0,0}\mathbf{A}^T]\,[\mathbf{U}_{1,0}\mathbf{A}^T]\cdots[\mathbf{U}_{L-1,0}\mathbf{A}^T]] \quad (7) \\ &= \mathbf{A}[[\mathbf{AU}_{0,0}]^T\,[\mathbf{AU}_{1,0}]^T\cdots[\mathbf{AU}_{L-1,0}]^T] \\ &= \mathbf{AY} \end{aligned}$$

where $\mathbf{Y}=[[\mathbf{AU}_{0,0}]^T\,[\mathbf{AU}_{1,0}]^T\cdots[\mathbf{AU}_{L-1,0}]^T]$ and each $[\mathbf{AU}_{i,j}^T]^T$ is a submatrix of **Y**. $\mathbf{G}_2$ can be expressed in a similar way. Using the above representations, the 2-D DCT is based upon the use of a M×N data block rather than the use of an M×M data block.

The computation of $\mathbf{G}_1$ can be divided into two matrix by matrix multiplications. The first matrix by matrix multiplication is the computation of **Y**. In this computation, one row of $\mathbf{F}_1$ is used to compute one row of **Y**. That is, we can compute $y(m,n)$ for $0 \le m \le N-1$ as follows:

$$y(m,n)=\begin{cases}\sum_{k=0}^{M-1} a(k,m)f(k,n) & \text{for } 0\le m\le M-1\\ \sum_{k=0}^{M-1} a(k,(m)_M)f(k+M,n) & \text{for } M\le m\le 2M-1\\ \vdots & \\ \sum_{k=0}^{M-1} a(k,(m)_M)f(k+(L-1)M,n) & \text{for } (L-1)M\le m\le LM-1\end{cases} \quad (8)$$

where $(m)_M$ is m modulus **M**, $y(m-1,n-1)$ is the m-th element of the n-th row of matrix **Y**, $a(k-1,m-1)$ is the k-th element of the m-th row of the coefficient matrix **A** which is defined in equation (3), and $f(k-1,n-1)$ is the k-th pixel of the n-th row of input data.

The second matrix by matrix multiplication is the computation of **AY**. In this computation, one column of **Y** is required to compute one column of $\mathbf{G}_1$. Then, $\mathbf{G}_1$ can be expressed as the following matrix:

$$\mathbf{G}_1=\begin{bmatrix}\sum_{k=0}^{M-1}a(k,0)y(0,k) & \sum_{k=0}^{M-1}a(k,0)y(1,k) & \sum_{k=0}^{M-1}a(k,0)y(N-1,k)\\ \sum_{k=0}^{M-1}a(k-1)y(0,k) & \sum_{k=0}^{M-1}a(k-1)y(1,k) & \sum_{k=0}^{M-1}a(k-1)y(N-1,k)\\ \sum_{k=0}^{M-1}a(k,M-1)y(0,k) & \sum_{k=0}^{M-1}a(k,M-1)y(1,k) & \sum_{k=0}^{M-1}a(k,M-1)y(N-1,k)\end{bmatrix} \quad (9)$$

From equation (6), we know that each input value is used **M** times. For instance $f(0,0)$ is used to compute $y(0,0)$ through $y(m-1,0)$. Each PE can compute these computations as a group. Thus, each PE will not need to store or transfer this input value after these computations. However, each PE will need to store **M** intermediate results until **Y** is computed. We define these intermediate results as state variables and designate them as $r_k(m,n)$ for $0\le k\le M-2$. Then, the first stage of computations for a pixel is as follows:

```
if ($(m)_M \ne M-1$) {
  for k=0 to k=M-1 {
    if ($(m)_M == 0$)
      $r_k((m)_M,n)=a((m)_M,k) * f(m,n)$
    else
      $r_k((m)_M,n)= a((m)_M,k) * f(m,n)$
                $+r_k((m)_M-1,n)$
  }
}
else {
  for k=0 to k=M-1 {
    $y(m-(M-1)+k,n)=a(M-1,k) * f(m,n)$
                $+r_k(M-2,n)$
  }
}
```

We observe in equation (9) that one column of **Y** is required to compute each element of $\mathbf{G}_1$. However, each column of **Y** requires more than one row of the input. Therefore, the intermediate results have to be stored in a buffer and they have to be retrieved during the processing of the next row of input data. Let's define these intermediate results as state variables and designate them as $q_k(m,n)$. Then, the computations for the second stage for **M** rows are as follows:

```
if ($(n)_M \ne M-1$) {
   for k=0 to k=M-1 {
      if ($(n)_M == 0$)
         $q_k(m,n)=a((n)_M,k) * y(m,n)$
      else
         $q_k(m,n)=a((n)_M,k) * y(m,n)$
               $+q_k(m,n-1)$
   }
}
else {
   for k=0 to k=M-1 {
      $g(m,n-(M-1)+k)=$
            $a(M-1,k) * y(m,n)$
               $+q_k(m,n-1)$
   }
}
```

If we assume that each processor in the VLSI system can compute one multiplication and one addition in a single cycle, each PE needs **2M** cycles for the processing of each pixel. It needs **M** cycles for the computation of the first stage and **M** cycles for the computation of the second stage. Since each row of an image frame has **N** pixels, each PE needs a total of 2NM cycles to process one row of input. Therefore, the throughput with a single PE is **1/2M** outputs per cycle.

Ⅲ. ALGORITHM DECOMPOSITION

In this section, an asynchronous architecture, which can provide high performance only with the low speed data communication channels, is presented. The order of the computations and the order in which the results are used are the same for all of the state variables. The state variables are the only data to be transferred between PEs. Therefore all data communications between PEs can be implemented with FIFOs, which will greatly reduce the handshaking overhead. Also, the amount of data to be transferred between the PEs is much smaller than that is required for systolic arrays whose PEs have to receive and transmit at least one data item per cycle. These advantages are obtained through utilizing both data partitioning and algorithm partitioning for the computational structure. With this approach, we can achieve real-time processing with relatively slow processors.

1. Data Partitioning

In this architecture, two levels of data partitioning are used to implement a 2-D DCT algorithm for image coding. We define the first level data partition to be one row of data and the second level data partition to be **M** rows of data. The first level data partitioning is possible through the use of equation (8). According to this equation, we need one row of data to compute one row of **Y** and the computations for different rows are independent. Therefore, processings of several rows can be executed in parallel with an appropriate number of processors. This can be applied to all 2-D DCT implementations.

The second level data partitioning is due to the characteristics of the image coding system, that is, the 2-D DCT for the first **M** rows of input is independent from the 2-D DCT of the next **M** rows of input. Therefore, several **M×M** 2-D DCTs can be processed in parallel.

In this implementation, we assign one second level data partition, which is M rows of data, to a group of PEs which has M PEs. Then, we assign one first level data partition, which is one row of data, to each PE in a group. Then, we obtain an M×M DCT as shown in figure 1. In this figure, PE 1 through PE M process one second level data partition which is M rows of input. PE M+1 through PE 2M process a next second level data partition, which is the next M rows of input. Each PE processes one row of data which is the first level data partition.

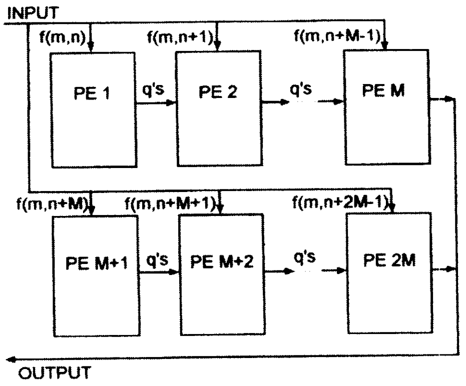

Figure 1: An implementation of the 2-D DCT with data partitioning

With only the first level of data partitioning, we can utilize at most **M** PEs because only **M** rows of data are available for the $\mathbf{M}\times\mathbf{M}$ 2-D DCT. Since **2NM** cycles are needed to process one row of data (N pixels per row) by a single PE, the throughput with **M** PEs is only one half of the PE speed which is one output for two cycles. Therefore, we can achieve real-time processing only if the PE speed is at least twice as fast as the input data rate. However, we can achieve higher throughput with the second level of data partitioning with a large number of relatively slow processors. If each processor can compute one multiplication and one addition in an input data interval, we can achieve real-time processing with two $\mathbf{M}\times\mathbf{M}$ 2-D DCTs in parallel as shown in figure 1. In this figure, an M PE group represents one $\mathbf{M}\times\mathbf{M}$ 2-D DCT without the second level of data partitioning. This is possible for the image coding system because the 2-D DCT for the M rows of data is independent from the 2-D DCT for the next M rows of data. Therefore, we can use many relatively slow speed processors to achieve real-time processing with two levels of data partitioning.

2. Algorithm partitioning

A 2-D DCT can be computed in two computational stages. By partitioning the 2-D DCT algorithm, we can replace a PE shown in figure 1 with two PEs, each of which performs one stage of the computations. The result is shown in figure 2 and the data path of the PE is shown in figure 3. In figure 2, PEs in the first row process the first stage of the computations and the PEs in the second row process the second stage of computations.

Since the processing of one row of input is independent from the processing of any other row of input for the first stage of computations, **M** rows can be computed in parallel by different PEs. In figure 3, PE 1 through PE M can operate at the same time for different rows of input. A PE does not need to transfer the r state variables to other PEs because these values are computed, stored, and used internally. This will greatly reduce the data transfers. Each PE processes an input in **M** cycles. Therefore, **M** PEs can process one pixel in a single cycle.

To obtain $\mathbf{G}_1$, we have to perform another matrix multiplication $\mathbf{G}_1 = \mathbf{AY}$. One column of $\mathbf{Y}$ is required to compute each element of $\mathbf{G}_1$. However, the elements of each column of $\mathbf{Y}$ are distributed among the **M** PEs. Therefore, $y(k,0)$ and $y(k,1)$, which belong to one column, are in different PEs. Therefore, each PE has to transmit intermediate results to the next PE which has the next element of the column of $\mathbf{Y}$. PE **M+1** through PE **2M** have to compute the second stage of computations.

For the second stage of computations, each equation involves **M** multiplications (see equation (9)). The first multiplication of the each equation requires one element in the first row of $\mathbf{Y}$ which is the first element in a column. The second multiplication of the each equation requires one element in the second row of $\mathbf{Y}$ which is the second element in a column. PE **M+1** has the first row of $\mathbf{Y}$ and PE **M+2** has the second row of $\mathbf{Y}$. Therefore, PE **M+1** computes the first multiplication and passes the result to P EM+2. Then, PE **M+2** computes the second multiplication, addition and passes the result to PE **M+3**, and so on. And PE **2M** computes the output with one multiplication and one addition. The order of transfers and the order of fetches for the q state variables are the same. Therefore, the transfer can be implemented with FIFOs. When we combine these two stages of computations, we can achieve a real-time 2-D DCT.

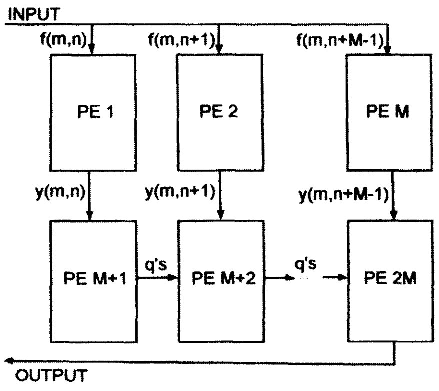

Figure 2: An implementation of the 2-D DCT with the data and algorithm partitioning

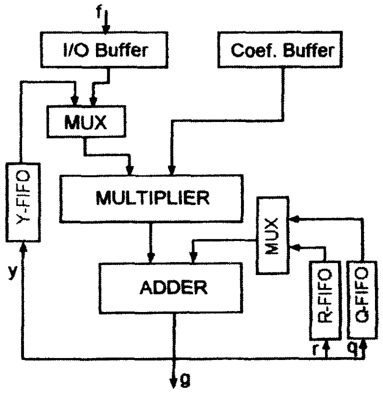

Figure 3: A data path of the PE for the system with the data and algorithm partitioning

Ⅳ. VLSI IMPLEMENTATION

We can implement a real-time 2-D DCT with a single VLSI chip with the implementation scheme introduced in this paper. The implementation presented has many good properties for VLSI implementation such as limited and local data transfers and a modular structure which reduces design time and cost. We do not need matrix transposition between the two matrix by matrix multiplications.

Figure 4 shows an example of VLSI implementation of a 4×4 2-D DCT with the algorithm partitioning. This VLSI implementation can be extended to implement a higher order 2-D DCT. For instance, we can use four 4×4 2-D DCT chips to implement a 16×16 2-D DCT which has the same throughput as the 4×4 2-D DCT. This configuration is shown in figure 5.

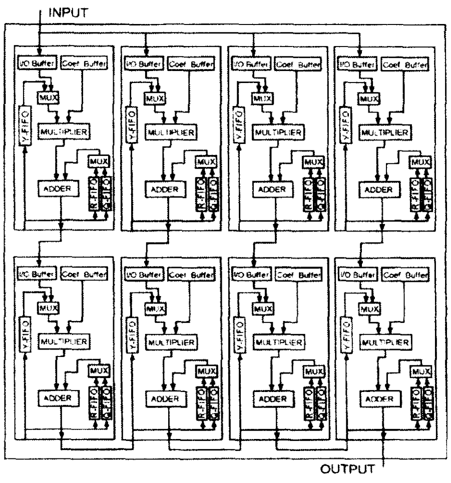

Figure 4: A VLSI implementation of the 4×4 2-D DCT with the algorithm and data partitioning

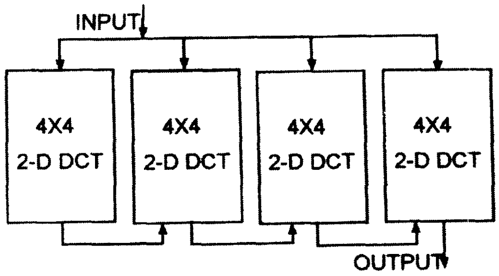

Figure 5: An implementation of the 16×16 2-D DCT with 4×4 DCTs

First, I analyses the data transfer. For the second stage of computations, each PE has to transfer state variables to the next PE because q state variables computed for one row are required for the processing of the next row. One data transfer is needed for every second stage of computations. For the first stage of computations, no data transfers are needed. For the system without the algorithm partitioning, each PE computes both stages of the algorithm. Therefore, the average data transfer rate is one data transfer for every two cycles. For the system with the algorithm partitioning, the data transfer rate between the PEs in the second row is one data transfer per cycle. It is twice as high as the rate for the system without algorithm partitioning. In addition to this, we need additional data transfers between the PE in the first row and the PE in the second row. This data rate is one data transfer for every **M** cycles because each y is computed in **M** cycles. Thus, this data rate is relatively slow compared to the input data rate.

Each PE, in a system with algorithm partitioning, has to compute **M** multiplications and **M** additions to process one pixel of input data because each PE has to compute one of the two stages of the algorithm. This means that we need **M** PEs for each stage of the algorithm for real-time processing. Therefore, the total number of PEs for real-time processing is **2M**. This is the same as the real-time system without algorithm partitioning. However, the use of algorithm partitioning reduces the number of PEs which need to be connected to the input data bus from **M** to **2M**. This allows us to reduce the input and output control hardware, especially for the high order real-time system which requires a large number of PEs. This is important because even though there are no bus contention problems, we still may have fan-in and fan-out limitations.

V. CONCLUSION

The implementation of the 2-D DCT introduced in this paper are mainly based upon the data partitioning. In the presented 2-D DCT implementation based on the state space model, one row of data is defined as one data partition; the first row of data is assigned to the first PE, the second row of data is assigned to the second PE, and so on. With this implementation scheme, we can achieve the followings:

- Speedup is virtually linear as we add more PEs. Thus, real-time throughput is possible with relatively slow PEs.
- Throughput is adjustable because it depends upon the number of PEs.
- Asynchronous operations for all PEs are possible. Thus, no global synchronization is required.
- Only limited operation overhead for asynchronous operation is achieved through the use of block data transfers and FIFOs.
- Only predetermined local data transfer is used.
- Simple interconnection networks are used, compared with the vector-radix type architecture
- Simple hardware is used, compared with the system based on the row-column decomposition technique. There is no requirement for a matrix transposition between stages and the input data does not have to be reformatted.
- It can be easily expanded for a higher order 2-D DCT.
- Very flexible implementation is possible for a VLSI chip. The amount of memory required is small.

REFERENCES

1. N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE Trans. Comput.*, vol. C-23, pp. 90-94, January 1974.
2. A. K. Jain, "Image data compression, A review," *Proceedings of IEEE*, vol. 69, pp. 349-389, March 1981.
3. A. Rosenfeld and A. C. Kak, *Digital Picture Processing*, 2nd edition. New York, NY: Academic Press, 1982.
4. M. Vetterli and A. Ligtenberg, "A discrete Fourier-cosine transform chip," *IEEE J. Select. Areas Commun.*, vol. SAC-4, pp. 49-61, January 1986.
5. S. C. Chan and K. L. Ho, "A new two-dimensional fast cosine transform algorithm," *IEEE Trans. Signal Process.*, vol. SP-39, no. 2, pp. 481-485, February 1991.
6. M. T. Sun, T. C. Chen, and A. M. Gottlieb, "VLSI implementation of a 16 16 discrete cosine transform," *IEEE Trans. Circuits Syst.*, vol. CAS-36, no. 4, pp. 610-617, April 1989.
7. D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.
8. K. Aono, M. Toyokura, and T. Araki, "A 30ns (600Mops) image processor with a reconfigurable pipeline architecture," in *Proceedings of IEEE 1989 Custom Int.* Circ. Conf., pp. 24.4.1-24.4.4, 1989.
9. U. Totzek and F. Matthiesen, "Two-dimensional discrete cosine transform with linear systolic array," in *Systolic array processors*, pp. 388-397, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989.
10. Anil K. Jain, *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1989.
11. H. C. Andrews and W. K. Pratt, "Transform image coding," in *Proc. Comp. Proc. Comm.*, pp. 63-84, New York, NY, Polytechnic Press, 1969.