TENSOR PRODUCT ALGEBRA AS A TOOL FOR VLSI IMPLEMENTATION OF THE DISCRETE FOURIER TRANSFORM

Domingo Rodriguez

Electrical and Computer Engineering Department
University of Puerto Rico, Mayagüez Campus
Mayagüez, PR 00709-5000

ABSTRACT

This work describes a tool to aid in the automated VLSI implementation of the discrete Fourier transform (DFT). This tool is tensor product algebra, a branch of finite-dimensional multilinear algebra. Tensor product formulations of fast Fourier transform (FFT) algorithms to compute the DFT are presented. These mathematical formulations are manipulated, using properties of tensor product algebra, to obtain variants that adapt to performance constraints in a VLSI implementation process. The possibility of automating this procedure by processing these mathematical formulations or expressions in a behavioral synthesis environment of a silicon compilation system is discussed. A transformation technique between a symbolic computation environment and a behavioral synthesis environment for the transferring of functional primitives is also discussed.

1. Introduction

The objective of this work is to present a methodology to assist in the the high level functional description of fast Fourier transform (FFT) algorithms in an automated VLSI synthesis system. This methodology is based on the use of tensor product algebra as a language for mathematical formulations of FFT algorithms. Properties of this algebra allows for the analysis and synthesis of the various stages of a given algorithm, obtaining a mathematical expression for each stage. These mathematical expressions are named functional primitives and are then used in a transformational environment. We concentrate on additive, Cooley-Tukey type FFT algorithms; but, the methodology can be also extended to multiplicative, Winograd type FFT algorithms.

Tensor product algebra can assist a designer at the functional specification level in an automated VLSI synthesis system. How this can be accomplished is illustrated with the following simple example. Consider taking the DFT of an 8-point complex sequence x. If we decide to compute the DFT using an FFT algorithm, this algorithm may be described using a signal flow graph (SFG) like the one in Fig. 1. A tensor product formulation of this algorithm can also be provided as it is shown in the same figure.

This way, a one-one correspondence is established between a tensor product decomposition and the signal flow graph (SFG) of an FFT algorithm. How to arrive at these mathematical expressions is clearly explained in the sections that follow. Pipepilining and parallel implementation possibilities can be exploited by applying properties of tensor product algebra to this decomposition. Examples of other tensor product decompositions obtained are shown on subsequent illustrations. We can use properties of tensor product algebra to go from one of these decompositions to another. We can also obtain other decompositions. At the present time we are trying to develop a methodology by which we can automate this process in a symbolic computation environment. This environment may, in turn, produce high level language programming code of an algorithm which can be translated to register transfer language (RTL) description as demonstrated by T. Tanaka, T. Kobayashi, and O. Karatsu [5]. Thus, this work concentrates on the mathematical formulations of FFT algorithms in tensor product language to be used in an abstract environment. Properties of these language allow manipulation of these formulations with relative ease. For this reason it can be used as a tool to search for tensor product decompositions suitable for VLSI implementations in a silicon compilation system.

2. Concepts of Tensor Product Algebra

We present some basic concepts of tensor product algebra. This algebra is becoming an important tool for presenting mathematical formulations of fast Fourier transform algorithms [3] for two main reasons. First, it serves as a mathematical language or environment in which to analyze in a unified format similarities and differences among canonical forms of commonly known FFT algorithms and their variants. Second, it serves as an analytic tool for the study of algorithm structures or constructs for machine hardware and software implementations as well as the identification of new algorithms. For instance, tensor product expressions can be identified with specific processing operations performed on targeted VLSI architectures. These identifications will assist in determining, in an automated VLSI design process which FFT algorithm constructs will produce efficient VLSI implementations relative to specified design constraints. This will help in reducing multiple design iterations. We start with the following definition of the tensor product of two matrices.

CH2977-7/91/0000-1025 $1.00 © 1991 IEEE

Let A be a $t \times r$ matrix. Let B be a $u \times s$ matrix. The tensor product $A \otimes B$ of the matrices A and B is a matrix C

$$C = \begin{bmatrix} a_{(0,0)}B & a_{(0,1)}B & \cdots & a_{(0,r-1)}B \\ a_{(1,0)}B & a_{(1,1)}B & \cdots & a_{(1,r-1)}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{(t-1,0)}B & a_{(t-1,1)}B & \cdots & a_{(t-1,r-1)}B \end{bmatrix} \quad (1)$$

The matrix C is $m \times n$ where $m = t \cdot u$, and $n = r \cdot s$. Let $Z/t = \{0, 1, \ldots, t-1\}$. Let $Z/t \times Z/u$ be the Cartesian set formed by Z/t and Z/u. We can relate the elements of the set $Z/t \times Z/u$, ordered lexicographically, to the elements of the set Z/m ordered in the natural order, i.e., $Z/m = \{0, 1, \ldots, m-1\}$. A possible mapping follows

$$\begin{aligned} \alpha : Z/t \times Z/u &\longrightarrow Z/m \\ (j_1, j_2) &\longmapsto \alpha(j_1, j_2) = (j) = j_1 + tj_2 \end{aligned} \quad (2)$$

In the same manner, we can establish a mapping between the set $Z/r \times Z/s$, ordered lexicographically, and the set Z/n, ordered in the natural order. We describe this mapping in the following way:

$$\begin{aligned} \beta : Z/r \times Z/s &\longrightarrow Z/n \\ (k_1, k_2) &\longmapsto \beta(k_1, k_2) = (k) = k_1 + rk_2 \end{aligned} \quad (3)$$

Using the above mappings, we write the tensor product of A and B in the following way:

$$C = A \otimes B = [c_{(j,k)}] = [c(j_1 + tj_2; k_1 + rk_2)] \quad (4)$$

where $j_1 \in Z/t, j_2 \in Z/u, k_1 \in Z/r, k_2 \in Z/s$.

The key to analyze tensor product formulations of FFT's is establishing a relationship between one-dimensional and two-dimensional arrays of indexing sets in a given tensor product decomposition. This type of relationship is also useful when studying stride permutation matrices. Below we give a brief description of stride permutation matrices. We then proceed in the next section with a tensor product formulation of a canonical form of the Cooley-Tukey FFT algorithm and discuss VLSI implementation.

Permutation matrices play a crucial role when trying to obtain variants of FFT algorithms to match targeted VLSI architectures. Specifically, we can state that the actual data flow required to carry out Cooley-Tukey type algorithms can be guided by stride permutation matrices. They are termed "stride" permutations because they can essentially be performed by striding or sampling through the data with a constant distance or length.

Take $n = r \cdot s$. Consider a "stride by s" permutation $P_{n,s}$ of order n. Its action on a tensor product $\underline{x} \otimes \underline{y}$ of two vectors $\underline{x}$ and $\underline{y}$, of length r and s, respectively, can be used to define the matrix itself. We can use two-dimensional arrays for this definition. Below, T denotes matrix transposition. The tensor product $\underline{x} \otimes \underline{y}$ is equivalent to the 2-dimensional array

$$[x_0\underline{y}\, x_1\underline{y} \, \ldots \, x_{r-1}\underline{y}] \quad (5)$$

$$[x_0\underline{y}\, x_1\underline{y} \, \ldots \, x_{r-1}\underline{y}]^T = [y_0\underline{x}\, y_1\underline{x} \, \ldots \, y_{s-1}\underline{x}] \quad (6)$$

$$P_{n,s}(\underline{x} \otimes \underline{y}) = \underline{y} \otimes \underline{x} \quad (7)$$

3. Implementation of the DFT

In this section we describe how tensor product algebra can assist in the VLSI implementation of fast algorithms for the computation of the DFT. The main idea is as follows. We associate basic hardware computational kernels with what we call functional primitives. A given FFT algorithm can be written as a composition of functional primitives. This action establishes a one-one correspondence between a mathematical formulation of an algorithm and a given hardware implementation. Variants of this mathematical formulation can be obtained using properties of tensor product algebra. These variants may satisfy certain design criteria such as pipelining, parallelism, data flow control, etc. In turn, each of these new variants will produce a different hardware implementation. The efficiency of the algorithm is evaluated when a cost function is imposed on the design criteria. In this section we present tensor product formulations of a class of fast Fourier transform algorithms to compute the one-dimensional discrete Fourier transform of a complex sequence. These algorithms are known as additive Cooley-Tukey FFT's. They are called additive because they take advantage of the additive structure of the input and output data indexing sets during computation. To this class belong canonical forms such as the Pease algorithm, the autosort or Stockham algorithm, the Agarwal-Cooley algorithm, etc. [4]. Variants of these forms may be obtained by using properties of tensor product algebra. Here we present a canonical form of the Cooley-Tukey FFT algorithm and discuss some of its variants. We first introduce some important definitions.

The diagonal matrix $D_{n,r}$, of order r is defined by

$$D_{n,r} = \text{diag}\,[1, w_n, \ldots, w_n^{r-1}], \; w_n = e^{-2\pi i/n}, \; i = \sqrt{-1} \quad (8)$$

The twiddle factor (phase factor) matrix $T_{n,s(r)}$, of order n, is defined as the direct sum of s diagonal matrices $D_{r,n/s}$ of order n/s:

$$T_{n,s(r)} = \sum_{\ell=0}^{s-1} \oplus D_{r,n/s}^{\ell} \quad (9)$$

The discrete Fourier transform (DFT) of an n-point sequence x is defined as

$$y(k) = \sum_{0 \le j < n} x(j) w_n^{jk}, \quad 0 \le k < n; \quad w_n = e^{-2\pi i/n} \quad (10)$$

where $y(k)$ represents the kth-term of the DFT sequence y; and $y = F_n \cdot x$, where F_n denotes the DFT matrix of order n. Here n is a composite of the form $n = r \cdot s$.

The sequence x is represented as a 2-dimensional array which we denote by X. The transformed output sequence $y = F_n \cdot x$ is also expressed as a 2-dimensional array which we denote by Y. Through these identifications, the original DFT sum is turned into a set of two recursive equations, each equation describing a modified Fourier sum. We first state the identifications explicitly and then proceed to formulate the Cooley-Tukey algorithm, expressing it in tensor product form. Using input/output index correspondences of the form:

$$\begin{aligned} \alpha : Z/s \times Z/r &\longrightarrow Z/n \\ (j_2, j_1) &\mapsto \alpha(j_2, j_1) = (j) = j_2 + sj_1 \end{aligned} \quad (11)$$

$$\begin{aligned}\beta : Z/r \times Z/s &\longrightarrow Z/n \\ (k_1, k_2) &\mapsto \beta(k_1, k_2) = (k) = k_1 + rk_2\,,\end{aligned} \tag{12}$$

we obtain 2-dimensional array representations of $Eq.$(10)

$$Y(k_1, k_2) = \sum_{j_2=0}^{s-1} \sum_{j_1=0}^{r-1} w_n^{(j_2+sj_1)(k_1+rk_2)} X_1(j_1, j_2) \tag{13}$$

which we rewrite as

$$Y(k_1, k_2) \;=\; \sum_{j_2=0}^{s-1} \Big[\sum_{j_1=0}^{r-1} X_1(j_1, j_2) w_r^{j_1 k_1} \Big] w_n^{j_2 k_1} w_s^{j_2 k_2} \tag{14}$$

Let $X_1 = X^T$ and let $\underline{x}_1$ be the vector formed by reading, in order, down the columns of X_1. The arrays X_1 and X are then related through the one-dimensional equivalence $\underline{x}_1 = P_{n,s}\underline{x}$. If we look closely at the inner Fourier sum

$$Y_1(k_1, j_2) \;=\; \sum_{j_1=0}^{r-1} X_1(j_1, j_2) w_r^{j_1 k_1}, \tag{15}$$

We notice that it can be written, using tensor product notation, in the following manner.

$$\underline{y}_1 = (I_s \otimes F_r)\underline{x}_1 = (I_s \otimes F_r) P_{n,s}\underline{x} \tag{16}$$

The next stage of the computation:

$$Y_2(k_1, j_2) \;=\; Y_1(k_1, j_2) w_n^{j_2 k_1} \tag{17}$$

can be given by the diagonal matrix multiplication

$$\underline{y}_2 \;=\; T_{n,s}\underline{y}_1 \tag{18}$$

where $T_{n,s}$ was previously defined by $Eq.$(9) above.

We complete the computation by

$$Y(k_1, k_2) \;=\; \sum_{j_2=0}^{s-1} Y_2(k_1, j_2) w_s^{j_2 k_2}. \tag{19}$$

If we denote by $\underline{y}_2$ the vector obtained by writing down, in order, the columns of Y_2, we can write

$$\underline{y} \;=\; (F_s \otimes I_r)\underline{y}_2 \tag{20}$$

Combining the above results we arrive at the tensor product formulation of the Cooley-Tukey FFT algorithm for computing a given n-point sequence x, where $n = r \cdot s$:

$$\underline{y} \;=\; (F_s \otimes I_r) T_{n,s} (I_s \otimes F_r) P_{n,s}\underline{x} \tag{21}$$

which produces the following decomposition for the Fourier matrix F_n:

$$F_n \;=\; (F_s \otimes I_r) T_{n,s} (I_s \otimes F_r) P_{n,s} \tag{22}$$

This tensor decomposition is commonly known as the decimation in time (DIT) Cooley-Tukey 2-factor FFT algorithm.

Performing a matrix transposition on both sides of the equation results in decimation in frequency (DIF):

$$F_n \;=\; P_{n,s}^{-1} (I_s \otimes F_r) T_{n,s} (F_s \otimes I_r) \tag{23}$$

Properties of tensor product algebra may be used in these two formulations to obtain other FFT variants. Also if n is a composite of the form $n = n_0 n_1 n_2 n_3 \ldots n_{k-1}$, iterative procedures can be used to obtain general tensor product decompositions. Many FFT variants can be obtained in this fashion, with certain classes of variants being identified with VLSI hardware structures.

6. Conclusion

As a conclusion it is important to emphasize that the increasing demand on VLSI circuits for digital signal processing applications is forcing some electronic companies to develop their own silicon compilation systems in order to improve the VLSI circuit design effort. Some of these companies claim to have reduced from months to days the time it takes to design custom integrated circuits for some of their products. Unfortunately, as expressed by Jonathan Allen on a recent article [1], current CAD synthesis resources still require the active participation of a skillful designer; and, as the design efforts grow large, "a human designer alone cannot handle the complexity implied by the large set of design possibilities." We see tensor product algebra as an important tool to aid at the high end level of functional specification so that we can exploit many alternative representations at the layout level and select the ones that best conform to given performance constraints.

Acknowlegement:

We acknowledge the assistance provided by the undergraduate student Marinés Puig Medina in the preparation of the illustrations.

REFERENCES:

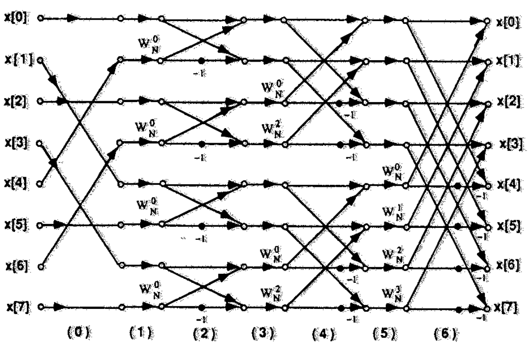
[1] J. Allen, "Performance-Directed Synthesis of VLSI Systems," Proceedings of IEEE, Vol. 78, No. 2, pp. 336-355: February 1990.

[2] W. G. Bliss, A. W. Julien, "Efficient and Reliable VLSI Algorithms and Architectures for the Discrete Fourier Transform," Proc. ICASSP '90, Vol. 2, pp. 901-904: April 1990.

[3] H. V. Sorensen, C. A. Katz, C. S. Burrus, "Efficient FFT Algorithms for DSP Processors Using Tensor Product Decompositions," Proc. ICASSP '90, Vol. 3, pp. 1507-1510: April 1990.

[4] D. Rodríguez, "Tensor Products Formulations of Additive Fast Fourier Transform Algorithms And Their Implementations," Ph.D. Thesis, City University of New York, Feb. 1988.

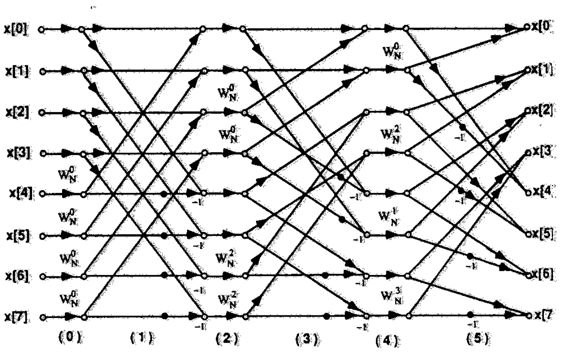
[5] T. Tanaka, T. Kobayashi, O. Karatsu, "HARP: Fortran to Silicon," IEEE Trans. on Computer-Aided Design, Vol. 8, No. 6, pp. 649-660: June 1989.



(0) $(I_2 \otimes P_{4,2})\, P_{8,2}$ (1) $(I_4 \otimes I_2)$ (2) $(I_4 \otimes F_2)$

(3) $(I_2 \otimes T_{4,2})$ (4) $((I_2 \otimes (F_2 \otimes I_2))$ (5) $T_{8,2}$ (6) $(F_2 \otimes I_4)$

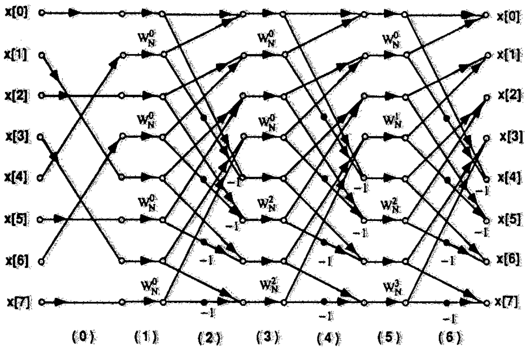
Figure 1



(0) $(\, I_2 \otimes I_4)$ (1) $(F_2 \otimes I_4)$ (2) $(P_{8,4}(I_2 \otimes T_{4,2})\, P_{8,2}$

(3) $(P_{4,2} \otimes I_2)\, (I_2 \otimes (F_2 \otimes I_2))$ (4) $T_{8,4}$ (5) $((\, F_2 \otimes I_4)\, (P_{4,2} \otimes I_2)$

Figure 2

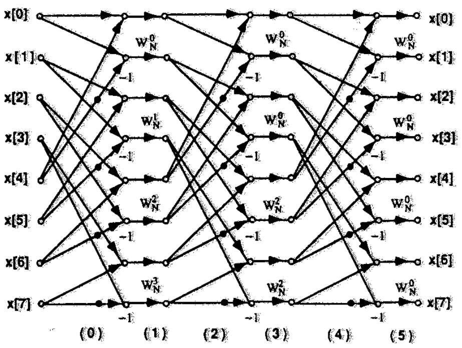


(0) $(I_2 \otimes P_{4,2})\, P_{8,2}$ (1) $(I_2 \otimes I_4)$ (2) $((F_2 \otimes I_4)\, (P_{4,2} \otimes I_2))$

(3) $((I_2 \otimes P_{4,2})\, (\, P_{8,4}\, (I_2 \otimes T_{4,2})\, P_{8,2})\, (I_2 \otimes P_{4,2}))$

(4) $((F_2 \otimes I_4)\, (P_{4,2} \otimes I_2))$ (5) $T_{8,4}$ (6) $((F_2 \otimes I_4)\, (P_{4,2} \otimes I_2))$

Figure 3



(0) $((I_4 \otimes F_2)\, P_{8,4})$ (1) $T_{8,4}$ (2) $((I_4 \otimes F_2)\, P_{8,4})$

(3) $((I_2 \otimes P_{4,2})\, (\, P_{8,4}(I_2 \otimes T_{4,2})\, P_{8,2})\, (I_2 \otimes P_{4,2}))$

(4) $((I_4 \otimes F_2)\, P_{8,4})$ (5) $(I_4 \otimes I_2)$

Figure 4