A METHODOLOGY FOR DESIGNING, MODIFYING, AND IMPLEMENTING FOURIER TRANSFORM ALGORITHMS ON VARIOUS ARCHITECTURES*

J. R. Johnson,[1] *R. W. Johnson,*[2] *D. Rodriguez,*[3] *and R. Tolimieri*[3]

Abstract. Fourier transform algorithms are described using tensor (Kronecker) products and an associated class of permutations. Algebraic properties of tensor products and the related permutations are used to derive variants of the Cooley–Tukey fast Fourier transform algorithm. These algorithms can be implemented by translating tensor products and permutations to programming constructs. An implementation can be matched to a specific computer architecture by selecting the appropriate variant. This methodology is carried out for the Cray X-MP and the AT&T DSP32.

1. Introduction

Shortly after Cooley and Tukey (C–T) introduced their algorithm for computing the Fourier transform (FT) [6], a large number of variations were created and this work was summarized by Cochran *et al.* in [5]. In this paper a method of obtaining variations of the C–T algorithm was presented, and it was conjectured this produced all of the "C–T-type" algorithms. Within a year Pease [13] introduced an algorithm not based on this method. In the introduction to Pease's paper he suggested strongly that the tensor product formulation was a valuable tool in the study of C–T-type algorithms. This was not taken up at this time due to the great success of the "butterfly" as a teaching and programming device.

With the advent of parallel and vector processors, there began another flurry of C–T algorithmic and programming effort. In much of this effort no use of the tensor product was made. However, in their comparison of several vector algorithms [11], Korn and Lambiotte make a slight use of tensor products as a tool. The most successful programming of the FT on vector computers like the Cray X-MP and Cyber 205 is due to Temperton [18], [20], [21]. In his expository paper [19] he places the tensor product at center stage. This paper was written for numerical physicists and is not familiar to the electrical engineering and computer science communities. In our paper we again place the tensor product at center stage, but we extend Temperton's work by presenting a detailed study of the permutations associated with tensor products—stride permutations—and their relation to the addressing requirements in C–T-type algorithms. We also make explicit the relationship of tensor products and stride permutations to the programming of these algorithms on various architectures.

Tensor products offer a natural language for expressing C–T-type algorithms. In the first section we introduce tensor products from a point of view best suited to our algorithmic and programming needs. We emphasize a decomposition which leads to an efficient evaluation of a tensor product on a vector. We also give several isomorphisms which make explicit the connections between various viewpoints of the C–T algorithms. These isomorphisms are later used to construct the indexing needed to program these algorithms.

Closely associated with tensor products are a class of permutations, containing stride permutations and tensor products of stride permutations, which govern the addressing between the stages of a tensor product decomposition. These permutations arise as a permutation of a tensor product basis. From a programming point of view these permutations interchange the order of the nested loops used to program a tensor product factorization. Some previous discussion of these permutations from a different point of view was given by Swartztrauber in his expository papers [16] and [17]. It is these permutations that are the basis of all variations of the C–T algorithm.

After introducing tensor products and the associated permutations, we give a direct method for programming tensor product factorizations. Once the programming of tensor products is made explicit, we can systematically study various ways of optimizing an implementation and modifying it to a specific architecture. The basic idea is to obtain a natural loop implementation based on tensor product identities and then to unroll some of the loops to match specific instruction sets. Furthermore, various factorizations of the permutations that arise can be used to adapt the algorithm to the specific addressing capabilities of a given architecture. In particular, in Section 4.4 we give a detailed study of the implementation of tensor products on the Cray X-MP [2]. Using tensor product and permutation identities, we obtain a vectorized segmented algorithm that uses addressing that can be efficiently implemented on this machine. Finally, using the programming techniques

discussed earlier we obtain a program that implements this algorithm. In [7] Granata and Rofheart use some of these ideas to obtain an efficient implementation of a 1K FT on the AT&T DSP32 [3]. By performing various compile time optimizations that we discuss in Section 4.2 they were able to save the cost of run time permutations. This savings helped them to achieve a program that was twice as fast as the distributed fast Fourier transform (FFT).

In the remaining sections we show how tensor products can be used in the design and implementation of FT algorithms. The derivation of the standard variations of the C–T algorithm using the tensor product and permutation language discussed in this paper was presented by Rodriguez in his thesis [14].

At the end of the paper we give some ideas on automating the techniques presented. Essentially, the algebraic properties of tensor products and stride permutations need to be incorporated into a special purpose compiler which can automatically generate codes to implement various C–T algorithms. Ideally, heuristics could be added to derive an algorithm suited to a given architecture. With such a compiler, an environment would be created for easily implementing and modifying FT algorithms on various architectures.

2. Tensor products

In this section we introduce some of the basic properties of tensor products which will play a major role in the design and implementation of FT algorithms. The formalism of tensor product notation can be used to keep track of the complex index calculation needed in FT algorithms. This property of the formalism can be used to aid in program design and verification. If the mapping between tensor product notation and machine instructions is automated, mathematical properties of tensor products can be used to guarantee the correctness of the implementation. Furthermore, tensor product identities can be used directly to transform the corresponding programs.

We begin with some definitions and some identifications which allow us to look at tensor products from several perspectives. Let C^n denote the n-dimensional vector space of n-tuples of complex numbers. The collection of vectors with a one in the ith position and zeros elsewhere form the standard basis for this vector space. We use e_i^n to denote such a vector. The superscript n indicates the size of the vector. Furthermore, we let i range from 0 to $n - 1$. We also use x^n to denote an arbitrary n-dimensional vector.

We can form the the tensor product $C^m \otimes C^n$ of the vector spaces C^m and C^n to get an mn-dimensional vector space with basis $\{e_i^m \otimes e_j^n | 0 \le i < m - 1,$ $0 \le j < n - 1\}$. We can associate this vector space with C^{mn} by ordering the basis lexicographically. This gives the following map:

$$e_i^m \otimes e_j^n \leftrightarrow e_{in+j}^{mn}, \tag{1}$$

which relates tensor products to mixed-radix indices. Let

$$x^m = \begin{pmatrix} x_0 \\ \vdots \\ x_{m-1} \end{pmatrix} = \sum_{0 \leq i < m} x_i e_i^m$$

and

$$y^n = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix} = \sum_{0 \leq j < n} y_j e_j^n.$$

Then using the bilinearity of the tensor product and this map we have

$$x^m \otimes y^n = \sum_{0 \leq i < m \, 0 \leq j < n} x_i y_j (e_i^m \otimes e_j^n) = \sum x_i y_j e_{in+j}^{mn} = \begin{pmatrix} x_0 y^n \\ \vdots \\ x_{m-1} y^n \end{pmatrix}.$$

This mn-dimensional vector can be mapped to an $m \times n$ matrix by placing the segments $x_i y^n$ of n elements in consecutive rows. For example,

$$x^m \otimes y^n \rightarrow \begin{pmatrix} x_0 y_0 & \cdots & x_0 y_{n-1} \\ \vdots & \ddots & \vdots \\ x_{m-1} y_0 & \cdots & x_{m-1} y_{n-1} \end{pmatrix}.$$

This operation identifies the vector space $C^m \otimes C^n$ with the vector space $C^{m,n}$ of $m \times n$ matrices. The standard basis for $C^{m,n}$ is $\{E_{i,j}^{m,n} | 0 \leq i < m, 0 \leq j < n\}$, where $E_{i,j}^{m,n}$ is the $m \times n$ matrix with a one in the (i, j)th position and zeros elsewhere. Using these basis elements, the preceding operation can be written as

$$e_i^m \otimes e_j^n \leftrightarrow E_{i,j}^{m,n}. \tag{2}$$

We can extend this map to act on an arbitrary mn-dimensional vector by placing consecutive segments of n elements in m consecutive rows. This identifies the vector space C^{mn} with $C^{m,n}$ by associating the basis vectors with the map

$$e_{in+j}^{mn} \leftrightarrow E_{i,j}^{m,n}. \tag{3}$$

The inverse map which takes an $m \times n$ matrix to an mn-dimensional vector by placing consecutive rows after each other is the same map that is used to store a two-dimensional array in linear memory. Clearly, this entire discussion could have been carried out based on antilexicographic ordering, which would have given the column method of storing arrays that is used in FORTRAN, instead of the row major ordering used in languages like Pascal. In the next section we study an important permutation which allows us to go back and forth between these two representations.

We can extend the definition of the tensor product of vectors to a tensor product of linear transformations by the following definition.

Definition 1. $(A \otimes B)(x \otimes y) = Ax \otimes By$, where A and B are linear transformations on the appropriate dimensional vector spaces.

If A and B are represented by matrices with respect to the standard basis and are of dimensions m and n, respectively, we have the following matrix picture:

$$A \otimes B = \begin{pmatrix} a_{0,0}B & \cdots & a_{0,m-1}B \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B & \cdots & a_{m-1,m-1}B \end{pmatrix}.$$

This block structured matrix, which replaces each element of the first matrix A by that element times the second matrix B is called the tensor product of two matrices (sometimes it is called the Kronecker product).

The action of the matrix $A \otimes B$ on an arbitrary mn-dimensional vector can be performed efficiently with the aid of the following decomposition:

$$A \otimes B = (A \otimes I_n)(I_m \otimes B) = (I_m \otimes B)(A \otimes I_n), \tag{4}$$

where I_n and I_m are n- and m-dimensional identity matrices. This decomposition is a corollary of the multiplication rule for tensor products.

Theorem 1 (Multiplication Rule for Tensor Products). $(A \otimes B)(C \otimes D) = AC \otimes BD$, where A and C are $m \times m$ matrices and B and D are $n \times n$ matrices.

This follows immediately from the definition since

$$(A \otimes B)(C \otimes D)(e_i^m \otimes e_j^n) = (A \otimes B)(Ce_i^m \otimes De_j^n) = (AC)e_i^m \otimes (BD)e_j^n.$$

Applying this identity to $A \otimes B = AI_m \otimes I_nB = I_mA \otimes BI_n$ gives the decompositions in equation (4). This multiplication rule and its implications are the most important tools in the design of efficient algorithms for computing with tensor products.

In order to better understand the computation of $(A \otimes B)x$ we need to examine the factors $I_m \otimes B$ and $A \otimes I_n$ that arise in decomposition (4). $I_m \otimes B$ is the direct sum of m copies of B:

$$I_m \otimes B = \begin{pmatrix} B & & \\ & \ddots & \\ & & B \end{pmatrix},$$

and its action on x is performed by computing the action of B on the m consecutive segments of size n. Clearly, this direct sum can be computed in

parallel on separate segments of the vector x, hence we call it a parallel tensor product term.

An alternative view of this computation can be obtained if we map x to the matrix X using equation (3). In this case B acts on the rows of the matrix:

$$X = \begin{pmatrix} x_0 & \cdots & x_{n-1} \\ x_n & \cdots & x_{2n-1} \\ \vdots & \ddots & \vdots \\ x_{(m-1)n} & \cdots & x_{mn-1} \end{pmatrix}.$$

Since matrices usually act on column vectors, we must first transpose X, let B act on the columns and then transpose the result to return to row representation. Thus the computation is given by the following matrix multiplication $(BX^t)^t = XB^t$, which can be mapped back to a vector using the inverse of the previous map.

The factor $A \otimes I_n$ can be interpreted as a vector operation on vectors of length n:

$$A \otimes I_n = \begin{pmatrix} a_{0,0}I_n & \cdots & a_{0,m-1}I_n \\ \vdots & \ddots & \vdots \\ a_{m-1,0}I_n & \cdots & a_{m-1,m-1}I_n \end{pmatrix}.$$

The action of $A \otimes I_n$ can be interpreted as a vector operation if we segment the input vector x into m consecutive segments of length n. If we let X_i denote the ith such segment, then $(A \otimes I_n)x$ is the following vector operation:

$$\begin{pmatrix} a_{0,0}X_0 + \cdots + a_{0,m-1}X_{m-1} \\ \vdots \\ a_{m-1,0}X_0 + \cdots + a_{m-1,m-1}X_{m-1} \end{pmatrix},$$

where $a_{i,j}X_j$ denotes a scalar-vector multiply and $+$ denotes a vector addition. This computation is just the evaluation of A on vector segments of length n.

An alternative interpretation of this computation results from the matrix point of view. In this case, A naturally acts on the columns of the matrix X giving the computation AX. The two factors $I_m \otimes B$ and $A \otimes I_n$ are related by changing from a row representation of X to a column representation of X.

Thus the computation of $y = (A \otimes B)x$ can be thought of as a parallel operation followed by a vector operation, or more conventionally as the matrix operation $Y = A(XB^t)$, where the result is obtained from the rows of the matrix Y. Clearly, the order of these two operations could be interchanged, as indicated by the two forms of the decomposition in equation (4), or by the associativity of matrix multiplication. Furthermore, the computation of the two types of factors $A \otimes I_n$ and $I_m \otimes B$ are identical up to a change of representation given by the transpose of the matrix X.

In terms of the vector x, this change of representation is obtained by a permutation, called a stride permutation, that skips over the segments representing the rows or columns. This permutation governs the addressing needed in the implementation of the factors $A \otimes I$ and $I \otimes B$. Further discussion of these permutations and their implementation is given in the next section.

Before discussing stride permutations, we need to show how to extend tensor products to arbitrarily many factors, and thus give a natural setting for multidimensional problems. The way to proceed is by induction from the two-dimensional case. For example,

$$(A_1 \otimes A_2 \otimes A_3)(x \otimes y \otimes z) = (A_1 \otimes A_2)(x \otimes y) \otimes A_3 z$$
$$= A_1 x \otimes A_2 y \otimes A_3 z,$$

where $A_1, A_2,$ and A_3 are $m \times m, n \times n,$ and $p \times p$ matrices and $x, y,$ and z are m-, n-, and p-dimensional vectors, respectively. In order for this to make sense, it is essential that the tensor product be associative. From our point of view this reduces to an index computation that is related to the induction used to store multidimensional arrays. Namely, to show that $A_1 \otimes A_2 \otimes A_3$ is associative we use the associativity of $x \otimes y \otimes z$, and to show that $x \otimes y \otimes z$ is associative we compute

$$(e_i^m \otimes e_j^n) \otimes e_k^p = e_{in+j}^{mn} \otimes e_k^p = e_{(in+j)p+k}^{mnp}$$
$$= e_i^m \otimes (e_j^n \otimes e_k^p) = e_i^m \otimes e_{jp+k}^{np} = e_{inp+jp+k}^{mnp}.$$

Using associativity we can uniquely define the tensor product of n terms by induction. This recursive definition makes it inductively easy to derive and program factorizations of tensor products, which can be used to compute multiple tensor products. In Section 4.1 we examine several important factorizations and their programming implications.

3. Commuting tensor products and stride permutations

In this section the permutations that arise from commuting tensor products are studied. The ability to commute tensor products is essential to modifying tensor-product factorization and hence modifying algorithms for computing with tensor products. As pointed out previously, stride permutations can be used to convert the parallel operation $I \otimes A$ to the vector operation $A \otimes I$. Alternatively, these permutations can be thought of as converting a row representation of a matrix to a column representation, or in other words transposing a matrix.

A stride permutation $P(mn, n)$ is defined by

Definition 2 (Stride Permutation).

$$P(mn, n)e_i^m \otimes e_j^n = e_j^n \otimes e_i^m.$$

To get an $mn \times mm$ matrix representation of this permutation, observe that $P(mn, n)$: $e_{in+j}^{mn} \to e_{jm+i}^{mn}$. For example,

$$P(6, 2)(x_0 e_0 + x_1 e_1 + x_2 e_2 + x_3 e_3 + x_4 e_4 + x_5 e_5)$$
$$= x_0 e_0 + x_1 e_3 + x_2 e_1 + x_3 e_4 + x_4 e_2 + x_5 e_5.$$

As a matrix computation, this can be written as:

$$P(6, 2)x = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{pmatrix}.$$

Thus we see that the elements of x are collected at stride 2 into two consecutive segments containing three elements each. The first segment begins with x_0, and the second segment begins with x_1. In general, $P(mn, n)$ reorders the coordinates at stride n into n consecutive segments of m elements; the ith segment beginning with x_{i-1}. This reordering of the coordinates corresponds to the inverse of the permutation of the basis elements. A physical interpretation of such a reordering can be observed when a deck of mn cards are dealt into n piles.

On some machines the action of a stride permutation might be implemented, as elements of the input vector are loaded from main memory into registers. For example, each segment might be loaded at the appropriate stride into a separate register beginning at the appropriate offset. For architectures where this is the case, considerable savings can be obtained by performing these permutations when loading the input vector into the registers. If it is necessary to load the input vector to perform some arithmetic operation, and the permutation is performed during this load, then a separate computation of the permutation can be avoided. In Section 4.4 we will see an architecture where these savings can be obtained. There is also a discussion of the implementation of stride permutations on that machine.

Because of this interpretation, we use L_n^{mn} to denote the stride permutation $P(mn, n)$. This notation indicates that a vector of size mn is reordered by loading into n segments at stride n. Shortly, we will see that the inverse of the stride permutation $P(mm, n)$ is the stride permutation $P(mn, m)$. Along the lines of the load interpretation of $P(mn, n)$, there is an interpretation of $P(mn, n)^{-1}$ as a store operation. In this case, n consecutive segments, residing in n registers, are stored back to memory at stride n with the ith segment beginning i positions from the first segment. It is clear that a load operation followed by the inverse store operation leaves the input vector fixed. Similarly to the L notation used for the load operation, we use S_n^{mn} to denote the inverse

store operation. Even though $(L_n^{mn})^{-1} = L_m^{mn} = S_n^{mn}$ all denote the same permutation, the notational distinction will be important when we are concerned with implementation. Even when we are not concerned with the implementation of these permutations, we use the L and S notation to help keep track of the indices. For example, the definition of $P(mn, n)$ can be conveniently remembered with

$$L_n^{mn}(x^m \otimes x^n) = x^n \otimes x^m. \tag{5}$$

An alternative view of these permutations as a change of representation arises from the matrix representation of $e_i^m \otimes e_j^n$. In this case, $L_n^{mn} E_{i,j}^{m,n} = E_{j,i}^{n,m} = (E_{i,j}^{m,n})^t$. Thus we see that a stride permutation effects a transposition. In other words, it collects terms by striding over the segments storing the rows of a matrix. For example, it we map a vector x containing six elements to the matrix.

$$X = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \\ x_4 & x_5 \end{pmatrix},$$

then

$$X^t = \begin{pmatrix} x_0 & x_2 & x_4 \\ x_1 & x_3 & x_5 \end{pmatrix}$$

gets mapped to the vector $L_2^6 x$.

The most important property of stride permutations is that they commute the factors in the tensor product of matrices. Using this property we will be able to show, as indicated in the last section, that

$$A \otimes B = L_m^{mn}(I_n \otimes A)L_n^{mn}(I_m \otimes B), \tag{6}$$

so that both factors in the decomposition can be performed as a loop, which is indicated by the direct sum interpretation of $I \otimes A$. In order to do this it is necessary to perform a change of basis by a stride permutation, which corresponds to changing from row representation to column representation. This change of basis is given in the following commutation theorem.

Theorem 2 (Commutation Theorem). $L_n^{mn}(A \otimes B) = (B \otimes A)L_n^{mn}$ where A is an $m \times m$ matrix and B is an $n \times n$ matrix. In other words $B \otimes A = L_n^{mn}(A \otimes B)(L_n^{mn})^{-1}$.

The proof is nothing more than a simple computation based on the definition:

$$L_n^{mn}(A \otimes B)(e_i^m \otimes e_j^n) = L_n^{mn}(Ae_i^m \otimes Be_j^n) = Be_j^n \otimes Ae_i^m.$$

Similarly,

$$(B \otimes A)L_n^{mn}(e_i^m \otimes e_j^n) = (B \otimes A)(e_j^n \otimes e_i^m) = Be_j^n \otimes Ae_i^m.$$

As an application of the commutation theorem observe that $A \otimes I_n = (L_n^{mn})^{-1}(I_n \otimes A)L_n^{mn} = S_n^{mn}(I_n \otimes A)L_n^{mn}$. The readdressing denoted by L_n^{mn} on input and S_n^{mn} on output turns the vector expression $A \otimes I_n$ into the parallel expression $I_n \otimes A$. In the same way, $I_m \otimes B = (L_n^{mn})^{-1}(B \otimes I_m)L_n^{mn}$. which turns the parallel expression $I_m \otimes B$ into the vector expression $B \otimes I_m$. As promised, we can now write $A \otimes B$ as

$$(A \otimes I_n)(L_n^{mn})^{-1}(B \otimes I_m)L_n^{mn} \tag{7}$$

or

$$(L_n^{mn})^{-1}(I_n \otimes A)L_n^{mn}(I_m \otimes B). \tag{8}$$

Using the fact that $(L_n^{mn})^{-1} = L_m^{mn}$, we can write these factorizations as

$$(A \otimes I_n)L_m^{mn}(B \otimes I_m)L_n^{mn} \tag{9}$$

and

$$L_m^{mn}(I_n \otimes A)L_n^{mn}(I_m \otimes B). \tag{10}$$

These factorizations decompose $A \otimes B$ into a sequence of vector operations and parallel operations, respectively. The intervening stride permutations provide a mathematical language for describing the readdressing between the stages of the computation. In the next two sections we show how knowledge of these permutations can be used to implement the addressing of a tensor product factorization on a variety of architectures. Furthermore, the direct interpretations of $A \otimes I$ as a vector operation and $I \otimes B$ as a parallel operation along with the commutation theorem automatically allows us to derive parallel and vector algorithms.

Before examining the implementation of stride permutations and the use of the commutation theorem in deriving various tensor product factorizations, we need to obtain a better understanding of these permutations. The algebra of stride permutations is especially rich and serves as an important tool in algorithm design. We begin with a multiplication rule, which can be expressed formally with our notation as:

Theorem 3. *If $N = rst$, then $L_{st}^N = L_s^N L_t^N$.*

Proof. First observe that $L_{st}^N(x^r \otimes x^s \otimes x^t) = x^s \otimes x^t \otimes x^r$. Since also $L_s^N L_t^N(x^r \otimes x^s \otimes x^t) = L_s^N(x^t \otimes x^r \otimes x^s) = x^s \otimes x^t \otimes x^r$, the theorem is proved. $\square$

In particular, since $L_N^N = I_N$, we have $L_n^N L_{N/n}^N = I_N$, so that $(L_n^N)^{-1} = L_{N/n}^N$. Since in general the inverse of a permutation matrix is the transpose we have that $L_{N/n}^N = (L_n^N)^t$. For example,

$$(L_2^6)^{-1} = (L_2^6)^t = L_3^6 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

As a simple application of Theorem 3 we get

Corollary 1. *If $N = p^k$, the set of stride permutations corresponding to the divisors of N,*

$$\{L_{p^j}^N : 0 \le j < k\},$$

form a cyclic group of order k generated by L_p^N.

The second type of theorem of importance in the algebra of stride permutations is a tensor-product decomposition of stride permutations. In general, the permutations that arise from commuting terms in a multidimensional tensor product are built up from products of terms of the form $I \otimes L \otimes I$. A permutation of the form $I \otimes L \otimes I$ will commute a tensor product and fix the remaining terms to the left and right. For example,

$$(A_{n_1} \otimes \cdots \otimes A_{n_i} \otimes A_{n_{i+1}} \otimes \cdots \otimes A_{n_t}) = (I_{n_1 \cdots n_{i-1}} \otimes L_{n_i}^{n_i n_{i+1}} \otimes I_{n_{i+2} \cdots n_t})$$
$$(A_{n_1} \otimes \cdots \otimes A_{n_{i+1}} \otimes A_{n_i} \otimes \cdots \otimes A_{n_t})$$
$$(I_{n_1 \cdots n_{i-1}} \otimes L_{n_{i+1}}^{n_i n_{i+1}} \otimes I_{n_{i+2} \cdots n_t}),$$

where A_{n_i} is an $n_i \times n_i$ matrix. To see this, use the multiplicative rule for tensor products and the commutation theorem. An alternative interpretation of this permutation is obtained by its action on a basis vector of the form $x^{n_1} \otimes \cdots \otimes x^{n_t}$. The collection of basis elements of this form is called a *tensor product basis* and the general permutations associated with tensor products result from permutations of the component positions of these basis elements. In terms of the tensor basis, this permutation exchanges the ith component with the $(i + 1)$st, and can be represented as a permutation of t objects by the transposition $(i, i + 1)$ written in cycle notation. Thus we have the following map which reduces stride permutations and tensor products of stride permutations to permutations of an appropriate tensor basis:

$$I_{n_1 \cdots n_{i-1}} \otimes L_{n_{i+1}}^{n_i n_{i+1}} \otimes I_{n_{i+2} \cdots n_t} \leftrightarrow (i, i + 1). \tag{11}$$

All permutations arising from repeated applications of the commutation theorem can thus be thought of as a permutation of the terms in the tensor basis, and can be written as a product of permutations of the form $I \otimes L \otimes I$. As such we have a convenient notation for dealing with such permutations.

Two special cases of these types of permutations are especially important for some architectures. These are the permutations $I_r \otimes L_t^{st}$ and $L_t^{st} \otimes I_r$. The first permutes the elements within the segments of the input vector and the second permutes the segments themselves. The permutation $I_r \otimes L_t^{st}$ permutes the elements in each of the r segments of size st by L_t^{st}, and $L_t^{st} \otimes I_r$ permutes the st segments of size r by L_t^{st}. $I_r \otimes L_t^{st}$ can be implemented as a loop of r stride permutations L_t^{st}, where the same permutation is performed, but the initial offset is incremented by st each iteration. $L_t^{st} \otimes I_r$ can be implemented by loading blocks of r consecutive elements, beginning at offsets given by the permutation L_t^{st}. A combination of these two types of permutations can by implemented efficiently on an architecture that can load at a given stride and can stride the offsets.

With these types of permutations in mind we shall derive a tensor product decomposition of stride permutations. This decomposition will be of importance on certain architectures, where the size of the registers must be taken into account. A detailed example using the Cray X-MP architecture is given in Section 4.4. The stride permutation L_t^N, where $N = rst$ can be thought of as a rotation of the tensor basis $x^r \otimes x^s \otimes x^t$. As such it can be decomposed into two transpositions. Formally, the permutation (r, s, t) can be written as $(r, s)(s, t)$, where the permutations are composed from right to left. This observation leads to the following decomposition theorem.

Theorem 4. *If* $N = rst$, *then* $L_t^N = (L_t^{rt} \otimes I_s)(I_r \otimes L_t^{st})$.

Proof. Since $L_t^N(x^r \otimes x^s \otimes x^t) = x^t \otimes x^r \otimes x^s$, and

$$(L_t^{rt} \otimes I_s)(I_r \otimes L_t^{st})(x^r \otimes x^s \otimes x^t) = (L_t^{rt} \otimes I_s)(x^r \otimes x^t \otimes x^s) = (x^t \otimes x^r \otimes x^s),$$

the theorem is proved.   □

Other decompositions of the permutations that arise in tensor-product factorizations can be obtained in the same way. The important point is that these permutations are really only permutations of the tensor-product basis rather than arbitrary permutations of the full vector. In many cases special features of the architecture can be used to implement these permutations without resorting to a general purpose implementation of an arbitrary permutation. For many algorithms dealing with tensor products, including the FT, this observation can lead to a substantial efficiency gain.

4. Implementing tensor products

We can now use the commutation theorem and other tensor-product identities to obtain some important factorization theorems for multidimensional tensor products. These factorization theorems have important implications for FT algorithms on various parallel and vector architectures. Using these factorizations and the addressing information given by the permutations, we show how to obtain a direct implementation of the factorization on a serial machine.

Tensor-product identities can be used to modify a factorization so that the addressing permutations are more suitable to a given architecture. First we show how to obtain a general vector or parallel algorithm. These algorithms contain features such as maximum vector length and constant data flow. However, when designing an algorithm for a specific machine, these idealized algorithms may not be appropriate. For example, a vector machine might have a maximum vector length, or a parallel machine might have a fixed number of processors or a specific communication network. In these cases, we need to fine tune an algorithm to conform to or take advantage of the features of the machine. We end this section with an example of how to use properties of tensor products in this tuning process.

4.1. *Factorization of tensor products and implications for various architectures*

Before deriving the factorizations, we need to introduce some notation. This notation is used throughout this section. Let n_i be a positive integer and $N(i) = n_1 \cdots n_i$. We use the convention that $N(0) = 1$. Finally, we represent an $n_i \times n_i$ matrix by A_{n_i}. When we have an arbitrary number of matrices in a tensor product, we use t to denote the number of factors. In this special case, we let $N = N(t)$. With this notation, we begin with the fundamental tensor-product factorization.

Theorem 5 (Fundamental Tensor-Product Factorization).

$$A_{n_1} \otimes \cdots \otimes A_{n_t} = \prod_{i=1}^{t} (I_{N(i-1)} \otimes A_{n_i} \otimes I_{N/N(i)}).$$

Furthermore, the factorization is true for any permutation of the factors $(I_{N(i-1)} \otimes A_{n_i} \otimes I_{N/N(i)})$.

Proof. The proof is by induction on t. For $t = 2$ the theorem is just the factorization $A_{n_1} \otimes A_{n_2} = (A_{n_1} \otimes I_{n_2})(I_{n_1} \otimes A_{n_2})$ given in equation (4). For the general case we have

$$A_{n_1} \otimes A_{n_2} \otimes \cdots \otimes A_{n_t} = (A_{n_1} \otimes I_{N/n_1})(I_{n_1} \otimes A_{n_2} \otimes \cdots \otimes A_{n_t}),$$

which by induction is equal to

$$(A_{n_1} \otimes I_{N/n_1})\left(I_{n_1} \otimes \prod_{i=2}^{t} (I_{N(i-1)/n_1} \otimes A_{n_i} \otimes I_{N/N(i)}) \right).$$

Using the tensor-product identities

$$(I \otimes BC) = (I \otimes B)(I \otimes C), \tag{12}$$

$$I_m \otimes I_n = I_{mn} \tag{13}$$

obtained from the multiplicative rule, in Theorem 1, and the definition, we get

$$(A_{n_1} \otimes I_{N/n_1})\prod_{i=2}^{t} (I_{N(i-1)} \otimes A_{n_i} \otimes I_{N/N(i)})$$

which gives the desired result. Since any two terms commute, the theorem is true, independent of the order of the factors. $\square$

Each term in this factorization $(I_m \otimes A \otimes I_n)$ involves m copies of the vector operation $A \otimes I_n$. In order to understand the implementation and modification of this factorization, we must study a general tensor-product term of this form. We begin by showing two ways to convert this term into the parallel form $I_{mn} \otimes A$. Besides the benefit for a parallel machine, this form has a natural interpretation as a loop, hence it can be directly implemented on a serial machine using a loop construct. In order to program $I_n \otimes A \otimes I_m$ as the loop $I_{mn} \otimes A$, we must keep track of the indexing given by the necessary stride permutations. Two possible indexing schemes are given by the following equations:

$$I_m \otimes A_{n_i} \otimes I_n = I_m \otimes L_{n_i}^{n_i n}(I_n \otimes A_{n_i})L_n^{nn_i}$$
$$= (I_m \otimes L_{n_i}^{n_i n})(I_{mn} \otimes A_{n_i})(I_m \otimes L_n^{nn_i}), \tag{14}$$

$$I_m \otimes A_{n_i} \otimes I_n = (I_m \otimes A_{n_i}) \otimes I_n = L_{mn_i}^{mn_i n}(I_{mn} \otimes A_{n_i})L_n^{mn_i n}. \tag{15}$$

The first equation can be implemented with a pair of nested loops with the innermost loop indexing given by the stride permutations L^{nn_i} and $L_{n_i}^{nn_i}$ or as a single loop with the input indexing given by $I_m \otimes L_n^{nn_i}$ and the output indexing given by its inverse. The second equation can be implemented with a single loop with its indexing given by $L_n^{mn_i n}$ and its inverse. A detailed translation between these equations and their implementations is given in Section 4.2.

These two modifications can be applied to the fundamental tensor-product factorization to obtain alternative factorizations which give direct implementations using loops.

The first indexing scheme gives the following two alternative factorizations:

Theorem 6.

$$A_{n_1} \otimes \cdots \otimes A_{n_t} = \prod_{i=1}^{t} I_{N(i-1)} \otimes (L_{n_i}^{N/N(i-1)}(I_{N/N(i)} \otimes A_{n_i})L_{N/N(i)}^{N/N(i-1)})$$

$$= \prod_{i=1}^{t} (I_{N(i-1)} \otimes L_{n_i}^{N/N(i-1)})(I_{N/n_i} \otimes A_{n_i})(I_{N(i-1)} \otimes L_{N/N(i)}^{N/N(i-1)}).$$

The last factorization can be simplified if we combine adjacent permutations, and thereby eliminate some permutations. We carry out this simplification to familiarize the reader with the permutation manipulations that arise in modifying tensor-product factorizations. The form of the factorization that would be used on a particular machine depends on the types of permutations that can be efficiently implemented. This simplification can easily be obtained from the permutation identity $(i - 1, i, \ldots, t)^{-1}(i, \ldots, t) = (i - 1, t)$. As a permutation of the tensor basis,

$$(I_{N(i-2)} \otimes L_{N/N(i-1)}^{N/N(i-2)})(I_{N(i-1)} \otimes L_{n_i}^{N/N(i-1)}) = P_{(i-1,t)},$$

where $P_{(i-1,t)}$ exchanges the basis elements in the $(i - 1)$st and tth positions. This leads to the following factorization

$$A_{n_1} \otimes \cdots \otimes A_{n_t} = \prod_{i=1}^{t} P_{(i-1,t)}(I_{N/n_i} \otimes A_{n_i}). \tag{16}$$

The second indexing scheme gives rise to

Theorem 7 (Parallel Tensor-Product Factorization).

$$A_{n_1} \otimes \cdots \otimes A_{n_t} = \prod_{i=1}^{t} L_{N(i)}^{N}(I_{N/n_i} \otimes A_{n_i})L_{N/N(i)}^{N} = \prod_{i=1}^{t} L_{n_i}^{N}(I_{N/n_i} \otimes A_{n_i}).$$

The second equation is obtained by using the multiplication rule for stride permutations, given in Theorem 3, to simplify adjacent stride permutations.

All of the modified factorizations that we have presented so far are completely parallelized in the sense that all tensor-product terms are of the form $I \otimes A$. We can easily use the commutation theorem to convert the parallel terms to vector terms to get factorizations that are completely vectorized. By complete vectorization, we mean that each tensor product $A \otimes I$ acts on vectors of the maximum length. For example, we commute the last factorization to get

Theorem 8 (Vectorized Tensor-Product Factorization).

$$\prod_{i=1}^{t} (A_{n_i} \otimes I_{N/n_i})L_{n_i}^{N}.$$

In many practical cases, complete vectorization is not desired because of machine limitations such as the size of the vector registers. In these cases we would like a factorization that is only partially vectorized. For example, if the maximum size of vector operations was 64, then we would like tensor-product terms of the form $I_m \otimes A \otimes I_{64}$. This would correspond to a loop of m vector operations on vectors of the maximum size possible on the machine in question. We can interpret this as segmenting a large vector operation into vector operations that fit on a given machine. Using the mathematical tools presented so far, it is easy to obtain a variety of factorizations that meet the addressing and architectural features of a given machine. In Section 4.4 we give an example of how to modify tensor-product operations to the CRAY X-MP.

4.2. Programming tensor-product factorizations

In this section we show how to program tensor-product factorizations. In general, we start with a set of base macros and combine them by performing operations corresponding to tensor products, stride permutations, and compositions. Thus macros are algebraically combined to form new macros. Furthermore, a macro can be optimized by applying algebraic transformations to it. Several examples of these operations are presented in this section.

We begin by showing how $I_m \otimes A$ corresponds to a loop. For this example and the rest of the examples throughout this section, we let

$$A = F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

To produce the code for $y = (I_m \otimes F_2)x$, we must start with the code for F_2.

$$F_2 \equiv F2(y, x)$$
$$\equiv y(0) = x(0) + x(1)$$
$$y(1) = x(0) - x(1)$$

$F2$ is a macro with two parameters corresponding to the base addresses of the input x and the output y. The tensor product of this code, $(I_m \otimes F_2)$, is constructed by looping over F_2.

$$I_m \otimes F_2 \equiv ITF2(m, y, x)$$
$$\equiv \text{for } i = 0, \ldots, m - 1$$
$$F2(y(2i), x(2i))$$
$$\equiv \text{for } i = 0, \ldots, m - 1$$
$$y(2i) = x(2i) + x(2i + 1)$$
$$y(2i + 1) = x(2i) - x(2i + 1)$$

Using this construction, any tensor-product term of the form $I \otimes A$ naturally gets mapped to a code that computes its action on a vector.

Two such code sequences can be concatenated to create a program that computes the product of matrices of that form. This is done by creating a temporary vector which serves as the output of the first code sequence and the input to the next. For example, $y = (I_m \otimes F_2)(I_m \otimes F_2)x$ is computed with

$$ITF2(m, t, x),$$
$$ITF2(m, y, t),$$

where t is a temporary vector.

In general tensor-product factorizations, not all tensor-product terms are of the form we desire to construct loop implementations. For example, in the fundamental factorization, the generic tensor-product term is of the form $I_m \otimes F_2 \otimes I_n$. This has a natural interpretation as a loop of m vector operations on vectors of length n.

$$\text{for } i = 0, \ldots, m - 1$$

$$\begin{pmatrix} Y_{2ni} \\ Y_{2n(i+1)} \end{pmatrix} = \begin{pmatrix} I_n & I_n \\ I_n & -I_n \end{pmatrix} \begin{pmatrix} X_{2ni} \\ X_{2n(i+1)} \end{pmatrix}$$

On a machine that does not offer vector instructions, $F_2 \otimes I_n$ must be computed as a loop. In order to obtain a direct loop interpretation, we must apply the commutation theorem. If we do this, we introduce stride permutations, $F_2 \otimes I_n = L_2^{2n}(I_n \otimes F_2)L_n^{2n}$, that must either be performed as actual permutations or incorporated into the code for computing $I_n \otimes F_2$ as readdressing.

First we show how to use the mathematical definition of a stride permutation to write a code to implement it. Recall that the stride permutation L_n^{mn} permutes the basis elements $e_{in+j}^{mn} \to e_{jm+j}^{mn}$. If we write an arbitrary vector in terms of this basis, we have

$$L_n^{mn}\left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} x_{in+j}e_{in+j}^{mn}\right) = \sum_{j=0}^{n-1}\sum_{i=0}^{m-1} x_{in+j}e_{jm+i}^{mn}.$$

Thus if $y = L_n^{mn}x$, we have that $y(jm + i) = x(in + j)$, where the index into the arrays is given by the basis elements. To get a program that computes this permutation, we need to loop over all possible values of i and j.

$$L_n^{mn} \equiv L(m, n, y, x)$$
$$\text{for } i = 0, \ldots, m - 1$$
$$\text{for } j = 0, \ldots, n - 1$$
$$y(jm + i) = x(in + j)$$

The order of these loops does not matter; however, it is essential that the dimensions are associated with the proper indices.

Using these stride macros, we can program $w = (F_2 \otimes I_n)x$ as $w = L_2^{2n}(I_n \otimes F_2)L_n^{2n}x$ with the following code sequences.

$$L(2, n, y, x)$$
$$ITF2(n, z, y)$$
$$L(n, 2, w, z)$$

We can optimize this code sequence by incorporating the permutations L_n^{2n} and L_2^{2n} into the code for $I_n \otimes F_2$ as readdressing.

To see how this readdressing is carried out, we expand these code sequences.

$$\text{for } j = 0, \ldots, 1$$
$$\quad \text{for } i = 0, \ldots, n - 1$$
$$\quad\quad y(i2 + j) = x(jn + i)$$
$$\text{for } i = 0, \ldots, n - 1$$
$$\quad z(2i) = y(2i) + y(2i + 1)$$
$$\quad z(2i + 1) = y(2i) - y(2i + 1)$$
$$\text{for } i = 0, \ldots, n - 1$$
$$\quad \text{for } j = 0, \ldots, 1$$
$$\quad\quad w(jn + i) = z(i2 + j)$$

In this expansion we have taken the liberty consistently to associate i with n and j with 2. This makes it easier to see the substitutions that are needed to combine the code sequences to obtain the correct readdressing. The first stride permutation L_n^{2n} combines with the code for $I_n \otimes F_2$ by substituting the input expression of $ITF2(n, z, y)$ with the output expression of $L(2, n, y, x)$. To get a direct match, we must further expand $L(2, n, y, x)$ by setting $j = 0$ and $j = 1$: $(e_{2i}^{2n} \rightarrow e_i^{2n}$ and $e_{2i+1}^{2n} \rightarrow e_{i+n}^{2n})$.

$$\text{for } i = 0, \ldots, m - 1$$
$$y(2i) = x(i)$$
$$y(2i + 1) = x(i + n)$$

After this expansion, we can combine the two code sequences.

$$\text{for } i = 0, \ldots, n - 1$$
$$z(2i) = x(i) + x(i + n)$$
$$z(2i + 1) = x(i) - x(i + n)$$

The composition with the output permutation $L_2^{2n} = S_n^{2n}$ is carried out in the same way. However, in this case the substitution is carried out with the

output variable. We use the notation S_n^{2n} instead of L_2^{2n} to make this distinction.

After both of these compositions are carried out, we arrive at the transformed (conjugated) code sequence which computes $w = (F_2 \otimes I_n)x$.

$$\text{for } i = 0, \ldots, n - 1$$
$$w(i) = x(i) + x(i + n)$$
$$w(i + n) = x(i) - x(i + n)$$

This code transformation eliminates the runtime permutations $L(2, n, y, x)$ and $L(n, 2, w, z)$ by changing the indexing of $ITF2(n, z, y)$ at compile time, thereby saving runtime memory accesses.

An alternative approach to this problem of transforming code sequences would be to augment the parameters of $ITF2$ to include stride information. For example, we could redefine $ITF2$ with the stride parameters a, s, b, and t. In the definition of $ITF2$ we have included the macro for $F2$ with the additional stride parameters s and t.

$$ITF2(m, y, b, t, x, a, s) \equiv \text{for } i = 0, \ldots, m - 1$$
$$F2(y(bi), t, x(ai), s)$$
$$\equiv \text{for } i = 0, \ldots, m - 1$$
$$y(bi) = x(ai) + x(ai + s)$$
$$y(bi + t) = x(ai) - x(ai + s)$$

This being the case, we have $S_m^{2m} ITF2(m, z, 2, 1, y, 2, 1)L_m^{2m} = ITF2(m, z, 1, m, y, 1, m)$.

We can now use these techniques to program $I_m \otimes F_2 \otimes I_n$. If we rewrite this as $I_m \otimes S_n^{2n}(I_n \otimes F_2)L_n^{2n}$, it can be programmed by looping over $ITF2(n, y, x)$.

$$\text{for } i = 0, \ldots, m - 1$$
$$ITF2(n, y(2ni), x(2ni))$$

We then expand this code sequence.

$$\text{for } i = 0, \ldots, m - 1$$
$$\text{for } k = 0, \ldots, n - 1$$
$$y(2ni + k) = x(2ni + k) + x(2ni + k + n)$$
$$y(2ni + k + n) = x(2ni + k) - x(2ni + k + n)$$

This code sequence could also have been constructed from the tensor-product expression $(I_m \otimes S_n^{2n})(I_m \otimes I_n \otimes F_2)(I_m \otimes L_n^{2n})$. The simplest way to implement this would be to construct $I_m \otimes S_n^{2n}$, $I_m \otimes (I_n \otimes F_2)$, and $I_m \otimes L_n^{2n}$ separately using the looping techniques discussed previously, and then

compose the code sequences together with the appropriate introduction of temporaries. If we then optimize by combining the permutations with $I_m \otimes I_n \otimes F_2$, we arrive at the same code sequence produced from $I_m \otimes (S_n^{2n}(I_n \otimes F_2)L_n^{2n})$. To see this observe that $(I_m \otimes L_n^{2n})e_i^m \otimes e_j^2 \otimes e_k^n = e_i^m \otimes e_k^n \otimes e_j^2$ which implies that $e_{(2i+j)n+k}^{m2n} \rightarrow e_{(in+k)2+j}^{m2n}$. Before applying this substitution, we list the code for $I_m \otimes I_n \otimes F_2$.

$$I_m \otimes I_n \otimes F_2 \equiv IITF2(m, n, x, y)$$
$$\equiv \text{for } i = 0, \ldots, m - 1$$
$$\text{for } k = 0, \ldots, n - 1$$
$$y(2in + 2k) = x(2in + 2k) + x(2in + 2k + 1)$$
$$y(2in + 2k + 1) = x(2in + 2k) - x(2in + 2k + 1)$$

If we make the substitution corresponding to the permutation $I_m \otimes L_n^{2n}$ and $I_m \otimes S_n^{2n}$, we obtain the same code for $I_m \otimes F_2 \otimes I_n$ that we derived previously.

So far we have only been able to program $I_m \otimes F_2 \otimes I_n$ using two nested loops. Yet we would like to be able to rewrite this as $I_{mn} \otimes F_2$ and only use one loop of mn iterations. The reason that we have not been able to do this is that the addressing given by the permutations $I_m \otimes L_n^{2n}$ and $I_m \otimes S_n^{2n}$ requires two indices. Instead of using the transformation that led to these permutations, we might try the second transformation $I_m \otimes F_2 \otimes I_n = S_n^{m2n}(I_{mn} \otimes F_2)L_n^{m2n}$. However, a careful inspection shows that if the permutations are composed with $I_{mn} \otimes F_2$, the same problem occurs. There is no way to combine the permutation $e_{in+j}^{m2n} \rightarrow e_{jm+i}^{m2n}$ with the loop for $I_{mn} \otimes F_2$ listed below.

$$\text{for } i = 0, \ldots, mn - 1$$
$$y(2i) = x(2i) + x(2i + 1)$$
$$y(2i + 1) = x(2i) - x(2i + 1)$$

The only way to combine the permutation with the code would be to rewrite $I_{mn} \otimes F_2$ as $I_m \otimes I_n \otimes F_2$ as before and look at L_n^{m2n} as a permutation on $e_i^m \otimes e_j^2 \otimes e_k^n$. In this way we see that $L_n^{m2n}(e_i^m \otimes e_j^2 \otimes e_k^n) = e_k^n \otimes e_i^m \otimes e_j^2$ and $e_{(2i+j)n+k}^{m2n} \rightarrow e_{(km+i)2+j}^{m2n}$, and this substitution leads to the same code as before.

The only way to get a single loop is to look at the complete tensor-product factorization. Using Theorem 7, we have

$$F_2 \otimes \cdots \otimes F_2 = \prod_{k=1}^{t} L_2^{2^t}(I_{2^{t-1}} \otimes F_2).$$

In this factorization we have terms of the form $L_2^{2mn}(I_{mn} \otimes F_2) = S_{mn}^{2mn}(I_{mn} \otimes F_2)$, which can be programmed with a single loop.

$$ITF2(mn, y, 1, mn, x, 2, 1) \equiv$$

$$\text{for } i = 0, \ldots, mn - 1$$
$$F2(y(i), mn, x(2i), 1)$$
$$\equiv \text{ for } i = 0, \ldots, mn - 1$$
$$y(i) = x(2i) + x(2i + 1)$$
$$y(i + mn) = x(2i) - x(2i + 1)$$

In this factorization, adjacent stride permutations combine to produce a stride permutation with appropriate stride so that it can be combined with F_2 in a single loop. However, the input and output permutations are no longer the same. This makes programming the algorithm in place impossible. Nonetheless, with the introduction of temporaries, t copies of this code can be used to compute $F_2 \otimes \cdots \otimes F_2$.

4.3. Modifying the implementation to parallel architectures

In this section we show how tensor-product factorizations lead to parallel and vector implementations. Tensor-product factors of the form $I_m \otimes A$ can be implemented as m copies of A, which can be done in parallel. Instead of implementing $y = (I_m \otimes F_2)x$ as a loop, each F_2 can be computed in parallel on m separate processors. If each processor has access to a shared memory containing the input vector x and the output vector y, then the code sequence $F2(y(2i), x(2i))$ can be computed by the ith processor. In the same fashion, $F_2 \otimes I_m$ can be computed as $S_m^{2m}(I_m \otimes F_2)L_m^{2m}$, where each processor computes $F2(y(i), m, x(i), m)$, an F_2 at stride m. Here we are using the $F2$ macro which includes stride parameters.

In general, $I_m \otimes F_2 \otimes I_n$ can be computed by mn processors labeled by the pair of integers (i, j), $0 \le i < m$, $0 \le j < n$, each computing $F2(y(2ni + j), n, x(2ni + j), n)$. Alternatively, $I_m \otimes (I_n \otimes F_2)$ can be thought of as m parallel computations of $I_n \otimes F_2$. In this case, each processor would compute the following code sequence.

$$\text{for } j = 0, \ldots, n - 1$$
$$y(2ni + 2j) = x(2ni + 2j) + x(2ni + 2j + 1)$$
$$y(2ni + 2j + 1) = x(2ni + 2j) - x(2ni + 2j + 1)$$

Using $I_m \otimes (I_n \otimes F_2)$ instead of $I_{mn} \otimes F_2$, is a natural way of controlling the granularity of the parallel computation. This is especially useful if there is a fixed number of processors. Returning to the general term, $I_m \otimes F_2 \otimes I_n = I_m \otimes (S_m^{2m}(I_n \otimes F_2)L_m^{2m})$ can be computed with m processors each computing the following loop.

$$\text{for } j = 0, \ldots, n - 1$$
$$y(2ni + j) = x(2ni + j) + x(2ni + j + n)$$
$$y(2ni + j + n) = x(2ni + j) - x(2ni + j + n)$$

We can now use the fundamental factorization and these ideas to compute $F_2 \otimes \cdots \otimes F_2$ in parallel. However, there is some added difficulty. First, between each stage $I_{2^{i-1}} \otimes F_2 \otimes I_{2^{t-i}}$, a barrier synchronization is needed to guarantee that the input to the next stage is correct. Furthermore, the natural interpretation of each stage leads to a different degree of parallelism at each stage. Different addressing and hence different programming at each stage is required to get a consistent degree of parallelism. This problem can be fixed if we use the parallel factorization instead. In this case, the addressing is the same at each stage and the natural interpretation has the maximum possible degree of parallelism at each stage. The only problem with this factorization is that we may desire a larger granularity and not the maximum degree of parallelism.

If we wish to adapt to a parallel processor with a fixed number of processors, we can modify this factorization accordingly. For example, if we have eight processors and wish to compute $F_2 \otimes \cdots \otimes F_2 = F_{2,10}$, the tensor product of ten factors of F_2, we could rewrite the factorization as

$$\prod_{i=1}^{10} S_{512}^{1024}(I_{512} \otimes F_2) = \prod_{i=1}^{10} S_{512}^{1024}(I_8 \otimes I_{64} \otimes F_2).$$

In this case each processor would compute the following code sequence.

$$\text{for } j = 0, \ldots, 63$$
$$\quad F2(y(2^7 i + j), 2^9, x(2^7 i + 2j), 1)$$
$$\equiv$$
$$\quad \text{for } j = 0, \ldots, 63$$
$$\quad y(2^7 i + j) = x(2^7 i + j) + x(2^7 i + 2j + 1)$$
$$\quad y(2^7 i + j + 2^9) = x(2^7 i + j) - x(2^7 i + 2j + 1)$$

There are two potential problems with this parallel implementation. These difficulties stem from locality and granularity considerations. After each computation of F_2, the results are stored back to main (shared) memory. It might be advantageous to do more local computation before doing the memory operation. Even for serial computers this might be the case. In fact any implementation on a machine with an hierarchical memory structure should be concerned with doing the proper amount of local computation. Furthermore, depending on the memory organization, it might be beneficial to load and store more than one operand or result at a time. These objectives can be satisfied by further modifying the factorization.

More local computation can be obtained if the decomposition is based on a larger unit of computation than F_2. For example, we may have enough local memory to compute $F_2 \otimes F_2$ instead of just F_2. In this case we can use the modified parallel factorization

$$\prod_{i=1}^{5} (S_{256}^{1024}(I_8 \otimes I_{32} \otimes (F_2 \otimes F_2))).$$

In this factorization there are only five factors instead of ten. Therefore, the number of required synchronizations has been decreased and the granularity has been increased.

In this example, we have chosen the number of F_2's in each stage (two) to divide the total number of factors (ten). If we had chosen a decomposition that was not compatible in this sense, the only difficulty that would arise would be an odd-sized factor somewhere in the computation. Such a decision might be necessitated by the limitations of the given architecture. Returning to our example, suppose we could compute $F_2 \otimes F_2 \otimes F_2 = F_{2,3}$ instead of just $F_2 \otimes F_2$. If we chose to have the odd-sized factor in front, there are two natural factorizations that we could use:

$$F_2 \otimes F_{2,9} = F_2 \otimes \prod_{i=1}^{3} (S_{64}^{512}(I_{64} \otimes F_{2,3}))$$

$$= (F_2 \otimes I_{512}) \prod ((I_2 \otimes S_{64}^{512})(I_2 \otimes I_{64} \otimes F_{2,3}))$$

$$= S_{512}^{1024}(I_{512} \otimes F_2)L_{512}^{1024} \prod ((I_2 \otimes S_{64}^{512}(I_2 \otimes I_{64} \otimes F_{2,3})), \quad (17)$$

$$F_{2,10} = S_{512}^{1024}(I_{512} \otimes F_2) \prod_{i=1}^{3} S_{128}^{1024}(I_{128} \otimes F_{2,3})). \quad (18)$$

The first factorization is obtained by a tensor-product construction of the decomposition of $F_{2,9}$ which is compatible with $F_{2,3}$. The difficulty with this construction is that the addressing given by the permutations is not compatible with $I_{128} \otimes F_{2,3}$, but instead is compatible with $I_2 \otimes I_{64} \otimes F_{2,3}$. This combined with the extra permutation given by the commutation of $F_2 \otimes I_{512}$ leads to a complicated program that cannot easily be programmed on our eight processor machine. However, the second factorization, which follows immediately from the mixed radix parallel factorization (Theorem 7) starting from $F_2 \otimes F_{2,3} \otimes F_{2,3} \otimes F_{2,3}$, can easily be programmed in the same manner as the compatible example based on $F_{2,2}$.

The second possible problem with the computation given by the parallel factorization is that each memory operation must be done separately. We would like to be able to do permutations within local memories of the separate processors, and then store large blocks of results back to shared memory. This type of data flow can be obtained by decomposing the intervening permutations into a collection of local permutations followed by a global block permutation. This is given by the tensor-product decomposition of stride permutations (Theorem 4). For our example, where we need to compute $S_{512}^{1024}(I_8 \otimes I_{64} \otimes F_2)$ this allows us to write

$$S_{512}^{1024} = (S_8^{16} \otimes I_{64})(I_8 \otimes S_{64}^{128}).$$

$I_8 \otimes S_{64}^{128}$ is carried out by permuting elements in local memory for each of the eight processors by S_{64}^{128}. After that local permutation, the results are sent

back to main memory in segments of length 64. These segments are permuted by S_8^{16}.

The regularity of the factorization $\prod S_{N/2}^N(I_{N/2} \otimes F_2)$ implies that the computation could be carried out on an array processor or in VLSI. For each stage of the computation, the same instructions (F_2) are computed on each segment of the input vector. Furthermore, the same permutation of the output is performed after each stage, so that the same routing instructions could be used to transmit the data after each stage. Thus it is quite conceivable to program tensor-product factorizations on an array processor, a single instruction multiple data architecture, with various interconnection networks for data transmission. Factorizations of stride permutations can be used to adapt algorithms to a given interconnection network.

As a simple example, we look at an array of four processors with a perfect shuffle interconnection network called the omega network [12]. This network has been designed for problems involving stride permutations [15]. The permutation $L_{2^n}^{2^n-1}$ has been called the perfect shuffle since its action on a deck of cards is obtained by shuffling two equal piles of cards so that the cards are interleaved one from each pile. Since the permutation L_4^8 is hardwired into this processor, a single pass through the network performs this permutation. If each processor can add and subtract, a single pass through this network can be used to compute $(I_4 \otimes F_2)L_4^8$ (see Figure 1). In the factorizations we have considered so far, terms of this form have not arisen. We have only seen $S_4^8(I_4 \otimes F_2)$. However, if we rederive the parallel
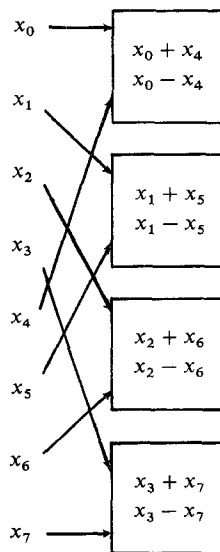

Figure 1. Omega network computing $(I_4 \otimes F_2)L_4^8$.

factorization for $F_{2,3}$ in the opposite order, the appropriate terms are obtained:

$$F_{2,3} = (I_4 \otimes F_2)(I_2 \otimes F_2 \otimes I_2)(F_2 \otimes I_4)$$
$$= (I_4 \otimes F_2)L_4^8(I_4 \otimes F_2)L_4^8(I_4 \otimes F_2)L_4^8. \tag{19}$$

Using this factorization, $F_{2,3}$ can be computed in three passes through the network.

In general, if we had an $N/2$ ($N = 2^n$) processor omega network, we could compute $F_{2,n}$ in $n = \log N$ passes. However, if we have a fixed number of processors, we can use the omega network as a module to help in larger computations. Such programs are constructed from tensor products:

$$F_{2,n} = \prod (S_{N/8}^N(I_{N/8} \otimes I_4 \otimes F_{2,3})), \tag{20}$$
$$F_{2,n} = \prod (S_{N/2}^N(I_{N/8} \otimes I_4 \otimes F_2))$$
$$= \prod ((S_{N/8}^{N/4} \otimes I_4)(I_{N/8} \otimes S_4^8)(I_{N/8} \otimes I_4 \otimes F_2)). \tag{21}$$

In this section we have presented a variety of techniques for implementing tensor products on a variety of parallel architectures. We have shown how to use tensor-product identities to modify algorithms to situations with a fixed number of processors, fixed granularity, shared memory, and special interconnection networks. While this should give a general overview of using tensor-product formulations to modify algorithms and obtain parallel implementations, none of the techniques can be fully appreciated without a specific example. In the next section we deal with questions of vectorization using the Cray X-MP as a specific example. While we have not specifically discussed vectorization in this section, the techniques needed are similar to the ones presented for the parallel architectures discussed.

4.4. The Cray X-MP: a design example

In this section we use the mathematical techniques developed in the previous sections to design algorithms for a specific architecture. In particular, we study how to implement tensor-product operations and the corresponding stride permutations efficiently on the Cray X-MP. The X-MP is a sample architecture from a class of machines called vector processors. In order to obtain an efficient implementation on this machine, it is essential that the algorithm be programmed to take advantage of its architectural features. For a vector processor like the X-MP, the two key programming concerns are vectorization and segmentation. These issues will become clearer as we present some examples.

The reason we have singled out a particular machine is not due to the limitations of our techniques, but rather that our techniques can be used to tune an algorithm to a specific architecture, and the X-MP serves as a nice example. Before studying the implementation of tensor products on

the X-MP, we briefly review the X-MP's architecture and highlight some of the key parameters, which are needed for tuning our algorithms to the machine. More information on the X-MP can be found in the hardware reference and CAL assembly manuals [1], [2]. A discussion of algorithm design and modification for the X-MP can be found in [8] and [9].

The X-MP is a pipelined vector processor. Built into the X-MP's instruction set are instructions for performing vector operations. For example, the X-MP has an instruction for adding all of the components of two vectors of floating-point numbers. Vector instructions are implemented, in hardware, by pipelining the elements of the vectors through a functional unit that performs the corresponding operation. In our example of floating-point addition, the functional unit that performs the addition is a six-stage pipeline. Therefore, the result obtained from adding the first two elements is produced in 6 CPs (clock periods) and the remaining results are produced one every clock period after that. So two vectors containing 64 elements each can be added in $6 + 63 = 69$ CPs. If the vector addition were performed with a loop of scalar additions, it would take $64 \cdot 6 = 384$ CPs. It is this speed-up that gives vector processing its power.

Thus the first concern in obtaining an efficient algorithm for the X-MP, is maximizing the use of vector instructions. Some important vector instructions for our purposes are vector addition and subtraction and scalar-vector multiplication.

It is important to realize that the vector instructions available on the X-MP are carried out on vectors located in vector registers. For example, the instruction

- $V0 \quad V1 + FV2 \qquad$ Floating point vector add

adds the vectors contained in registers $V1$ and $V2$ and produces the result in register $V0$. Since the vector registers can contain a maximum of only 64 elements, this limits the size of vectors that can be used in vector instructions. However, several vector instructions can be combined to perform operations on larger vectors, which we call supervectors. Splitting a supervector into appropriate segments on which vector instructions can be used, is the second major concern in designing an algorithm for the X-MP. In this case, the size of the vector registers is a key design parameter.

Before giving an example of a supervector instruction, we need to examine how vectors are loaded into and stored from the vector registers. Also, studying these memory operations is essential to implement the loadstride permutations that arise from tensor-product operations efficiently. A vector of elements in memory begining at X and separated at stride s can be loaded into a vector register with the following instruction:

- $Vi \quad X, s \qquad$ Load a vector beginning at X into Vi at stride s.

The number of elements that are loaded is determined by the contents of a

special register called the vector length register VL. Similarly, a vector register can be stored to memory at any given stride:

- $, Y, s$ Vk Store Vk to Y at stride s.

Both of these instructions are performed in the same pipelined fashion that other vector instructions are. Ignoring potential memory conflicts, a vector of 64 elements can be loaded in $17 + 63 = 80$ CPs, the time to load the first element plus one CP each for the remaining elements. If memory operations were not done with vector instructions, then performance degradation would be disasterous. By properly segmenting an algorithm its performance can be improved dramatically.

Now that we know how to load segments of vectors into the vector registers, we can see how to perform a supervector instruction like super-vector addition. To do this we need a loop that loads a segment of each vector, adds them, and stores the resulting segment. Since the functional units on the X-MP are independent and there are three independent memory ports, these operations can be performed concurrently. The overlap obtained from this concurrency can be thought of as another level of pipelining. In the example of supervector addition, we have a three-stage pipeline so that while two segments are being loaded, another two can be added, and the previous result can be stored. Here we see another important benefit of proper segmentation, namely the overlap of the operations on the segments.

We now begin our study of the implementation of tensor-product opera-tions on the X-MP. Tensor-product terms of the form $A \otimes I_n$, for $n \leq 64$, can be implemented directly with vector instructions. For example.

$$y = (F_2 \otimes I_3)x = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_0 + x_3 \\ x_1 + x_4 \\ x_2 + x_5 \\ x_0 - x_3 \\ x_1 - x_4 \\ x_2 - x_5 \end{pmatrix}.$$

If we let $V0$ contain the vector (x_0, x_1, x_2) and $V1$ contain (x_3, x_4, x_5), the tensor-product operation can be performed with the following vector instruc-tions.

- $V2$ $V0 + FV1$
- $V3$ $V0 - FV1$

The result is obtained by storing $V2$ followed by $V3$ back to memory. If Y is the location of the output vector, this is done with the following instructions:

- $, Y, 1$ $V2$ Store first segment at stride 1.
- $, Y + 3, 1$ $V3$ Store second segment at stride 1.

In this case the stride is 1 for the individual stores, but the offset must be incremented by the size of the vector segments.

Next we see how to implement stride permutations using load and store operations. As in the previous example, we let x be a vector with six elements. First we would like to perform the load operation $y = L_2^6 x$, which corresponds to the permutation

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{pmatrix}.$$

The following load instructions can be used to perform this operation:

- $V0 \quad , X, 2 \quad$ Load first segment at stride 2.
- $V1 \quad , X + 1, 2 \quad$ Load second segment at stride 2.

After these loads are performed,

$$V0 = \begin{pmatrix} x_0 \\ x_2 \\ x_4 \end{pmatrix}, \qquad V1 = \begin{pmatrix} x_1 \\ x_3 \\ x_5 \end{pmatrix}.$$

To obtain the permuted vector Y, we must store these registers back to memory:

- $, Y, 1 \quad V0 \quad$ Store first segment at stride 1.
- $, Y + 3, 1 \quad V1 \quad$ Store second segment at stride 1.

The same permutation could be carried out with the storestride operation S_3^6. In this case we load the registers $V0$, $V1$, and $V2$ with consecutive elements, and store the registers back at stride 3:

- $V0 \quad , X, 1 \quad$ Load first segment at stride 1.
- $V1 \quad , X + 2, 1 \quad$ Load second segment at stride 1.
- $V2 \quad , X + 4, 1 \quad$ Load third segment at stride 1.

After these loads, we have

$$V0 = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}, \qquad V1 = \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}, \qquad V2 = \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}.$$

These registers are then stored back to main memory.

- $, Y, 3 \quad V0 \quad$ Store first segment at stride 3.
- $, Y + 1, 3 \quad V1 \quad$ Store second segment at stride 3.
- $, Y + 2, 3 \quad V2 \quad$ Store third segment at stride 3.

After the first store operation $y = (x_0, \; , \; , x_1, \; , \;)$, after the second store $y = (x_0, x_2, \; , x_1, x_3,)$, and after the third store y contains the appropriately permuted vector.

To see how loadstride and storestride permutations can be implemented in conjunction with tensor product operations, we show how terms like $(A \otimes I)L$ and $S(A \otimes I)$ are implemented. We begin by looking at $(F_2 \otimes I_3)L_2^6 x$. As in the previous example, we load x into two vector registers at stride 2. However, before storing the vectors back to memory, we perform the vector operation $F_2 \otimes I_3$ as in the first example, obtaining

$$V2 = \begin{pmatrix} x_0 + x_1 \\ x_2 + x_3 \\ x_4 + x_5 \end{pmatrix}, \qquad V3 = \begin{pmatrix} x_0 - x_1 \\ x_2 - x_3 \\ x_4 - x_5 \end{pmatrix}.$$

Finally, these registers are stored back giving the desired output vector. It is imperative that the loadstride operation be compatible with the tensor-product operation. In this case, we must have two registers with three elements each in order to be able to perform $F_2 \otimes I_3$.

The operation $S_2^6(F_2 \otimes I_3)$ can be implemented in a similar fashion. In this case, after performing the vector operation $F_2 \otimes I_3$, we have two registers each containing three elements, which can be stored at stride 2. An important feature of both of these examples, is that in order to perform $F_2 \otimes I_3$ we must load the input vectors and store the result even if there were no permutation. By performing the permutation during the loading or storing phases we are essentially obtaining the permutation for free. In effect we are saving the extra memory operations that would be needed if the permutation was carried out separately.

To see how tensor-product factors with preceding loadstride permutations can arise, let A be a 2×2 matrix and let B be a 3×3 matrix and consider the factorization given by the commutation theorem. In this case we need to implement the factorization

$$z = (A \otimes B)x = (A \otimes I_3)(I_2 \otimes B)x \qquad (22)$$
$$= (A \otimes I_3)L_2^6(B \otimes I_2)L_3^6 x. \qquad (23)$$

We would perform $y = (B \otimes I_2)L_2^6 x$ followed by $z = (A \otimes I_3)L_2^6 y$. This factorization allows $A \otimes B$ to be performed using only vector instructions. Also the factorization forces the tensor-product operations to be compatible with the loadstride operations.

Up until now, by assuming vectors fit inside the vector registers, we have ignored the problem of segmentation. Since the size of the vector registers is 64, we would like to perform vector instructions on vectors with 64 elements. In terms of tensor-product operations, we would like factors of the form $I_m \otimes A \otimes I_{64}$. Such a factor corresponds to performing a loop of m tensor-product operations on vectors of length 64. Factors of this form are not

always present in tensor-product factorizations; however, it is possible to use tensor-product identities to manipulate the factorization so that appropriate factors can be obtained.

For example, suppose we need to evaluate $F_2 \otimes I_{128}$. Here the size of the vector operation is 128, so that the vector appears not to fit in the vector registers. We would like to rewrite this as $I_2 \otimes F_2 \otimes I_{64}$, in order to get the correct vector length. It is clear that the commutation theorem can be used to do this; however, if we do this in the obvious way, we run into some difficulty. To see this observe

$$F_2 \otimes I_{128} = (F_2 \otimes I_{64}) \otimes I_2 \tag{24}$$

$$= L_{128}^{256}(I_2 \otimes (F_2 \otimes I_{64}))L_2^{256}. \tag{25}$$

In order to perform the vector operation $F_2 \otimes I_{64}$, we need vectors of length 64, but L_2^{256} gives two vectors of length 128:

$$\begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ \vdots \\ x_{254} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_{255} \end{pmatrix}.$$

In order to solve this difficulty, we need to use the loadstride factorization

$$L_2^{256} = (L_2^4 \otimes I_{64})(I_2 \otimes L_2^{128})$$

given by Theorem 4. This factorization can easily be remembered if the terms in $F_2 \otimes I_{64} \otimes I_2$ are commuted in stages. The first factor $I_2 \otimes L_2^{128}$ corresponds to a permutation within segments, and the second factor $L_2^4 \otimes I_{64}$ can be thought of as a permutation of segments. $I_2 \otimes L_2^{128}$ creates four segments of size 64:

$$V0 = \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{126} \end{pmatrix}, \quad V1 = \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_{127} \end{pmatrix}, \quad V2 = \begin{pmatrix} x_{128} \\ x_{130} \\ \vdots \\ x_{254} \end{pmatrix}, \quad V3 = \begin{pmatrix} x_{129} \\ x_{131} \\ \vdots \\ x_{255} \end{pmatrix}.$$

$L_2^4 \otimes I_{64}$ permutes these segments giving $(V0, V2, V1, V3)$. Now we can apply $I_2 \otimes F_2 \otimes I_{64}$ to these segments. First apply $F_2 \otimes I_{64}$ to $(V0, V2)$ and then to $(V1, V3)$. It is clear that the addressing given by $(L_2^4 \otimes I_{64})(I_2 \otimes L_2^{128})$ can be carried out by doing loadstrides as before on the segments and by changing the initial offsets of the loadstrides to perform the permutation of the segments. In this example, we start with an offset to x_0 and another offset to x_{128}. We then go through a loop which loads segments of 64 elements with L_2^{128}.

To complete the operations, we need to store the elements produced by $I_2 \otimes F_2 \otimes I_{64}$. Furthermore, this must be done in the same order as they are computed. This can be done in the same way that L_2^{256} was performed. Since $L_{128}^{256} = S_2^{256} = (L_2^{256})^{-1}$,

$$
\begin{aligned}
S_2^{256} &= ((L_2^4 \otimes I_{64})(I_2 \otimes L_2^{128}))^{-1} \\
&= (I_2 \otimes L_2^{128})^{-1}(L_2^4 \otimes I_{64})^{-1} \\
&= (I_2 \otimes (L_2^{128})^{-1})((L_2^4)^{-1} \otimes I_{64}) \\
&= (I_2 \otimes S_2^{128})(S_2^4 \otimes I_{64}).
\end{aligned}
$$

Thus the storestride operations can be carried out in a manner analogous to the loadstride operations. In our example, after performing $F_2 \otimes I_{64}$ on $(V0, V2)$, namely

- $V4 \quad V0 + FV2$
- $V5 \quad V0 - FV2$

we store $V4$ at stride 2 starting at y_0 and $V5$ at stride 2 starting at y_{128}. After the next $F_2 \otimes I_{64}$. $V6 = V1 + FV3$ is stored at stride 2 beginning at y_1 and $V7 = V1 - FV3$ is stored at stride 2 beginning at y_{129}. Thus the complete operation $F_2 \otimes I_{128}$ is performed as a vector loop (or supervector instruction) corresponding to the factorization

$$
(I_2 \otimes S_2^{128})(S_2^4 \otimes I_{64})(I_2 \otimes F_2 \otimes I_{64})(L_2^4 \otimes I_{64})(I_2 \otimes L_2^{128}).
$$

In this supervector instruction, segments of 64 elements are loaded into vector registers, a vector F_2 is performed, and the resulting segments are stored. By keeping the appropriate offsets, given by the loadstride and storestride factorizations, this can be programmed as a simple loop.

Several key points should be noted about the segmentation obtained for this loop. First, when the segments of 64 elements are loaded, both the addition and subtraction from F_2 are performed before getting rid of the input segments. This is important since it eliminates the unnecessary memory operations that would be performed if the vector had to be reloaded to do the subtraction. This is an example of the key design goal of minimizing memory operations and keeping vectors in the vector registers as long as possible. It should be pointed out that the X-MP has only eight vector registers, so that the care that we have gone through in segmenting the computation (i.e., matching input and output segments) is essential.

A second benefit of the segmentation is the overlap it implies. In this example, while we are doing the additions or subtractions, we can simultaneously be doing the loads and stores on other segments. The way to think of this overlap, as pointed out before, is as a vector instruction pipeline. The overlap in this example is depicted in the timing diagram in Figure 2. The operations indicated on the left correspond to functional units and memory operations that can be performed concurrently on the X-MP. The numbers
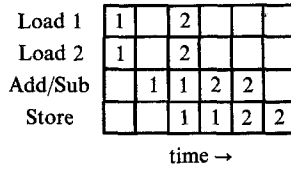
Load 1	1		2		
Load 2	1		2		
Add/Sub		1	1	2	2
Store			1	1	2	2

time →

Figure 2. Timing diagram.

indicate the segment the operation is being performed on, and more than one number in a single column indicates overlap. Furthermore, the add unit is fully utilized and a vector is produced every time slot.

We would like to point out that there is another way to segment the operation $F_2 \otimes I_{128}$. In general, there are many different ways of performing tensor-product operations, and these different methods of computation correspond to different tensor-product factorizations. The different factorizations can conveniently be obtained by applying different tensor-product identities in various orders. An alternative method of segmenting $F_2 \otimes I_{128}$ can be obtained as follows:

$$F_2 \otimes I_{128} = (F_2 \otimes I_2) \otimes I_{64} \tag{26}$$

$$= S_2^4 (I_2 \otimes F_2) L_2^4 \otimes I_{64} \tag{27}$$

$$= (S_2^4 \otimes I_{64})(I_2 \otimes F_2 \otimes I_{64})(L_2^4 \otimes I_{64}). \tag{28}$$

In this case we let

$$V0 = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{63} \end{pmatrix}, \quad V1 = \begin{pmatrix} x_{64} \\ x_{65} \\ \vdots \\ x_{127} \end{pmatrix}, \quad V2 = \begin{pmatrix} x_{128} \\ x_{129} \\ \vdots \\ x_{191} \end{pmatrix}, \quad V3 = \begin{pmatrix} x_{192} \\ x_{193} \\ \vdots \\ x_{255} \end{pmatrix},$$

and compute

- $V4 \quad V0 + FV2,$
- $V5 \quad V0 - FV3,$
- $V6 \quad V1 + FV3,$
- $V7 \quad V1 - FV3,$

storing the results in the order ($V4$, $V6$, $V5$, $V7$). This computation can be implemented by setting offsets to x_0 and x_{128}. As the computation proceeds, the offsets are incremented by 64 and the corresponding $F_2 \otimes I_{64}$ is performed.

Both of the methods of computing $F_2 \otimes I_{128}$ easily generalize as indicated by the two factorizations

$$F_2 \otimes I_{64r} = (I_2 \otimes S_r^{64r})(S_r^{2r} \otimes I_{64})(I_r \otimes F_2 \otimes I_{64})(L_r^{2r} \otimes I_{64})(I_2 \otimes L_r^{64r}) \tag{29}$$

$$= (S_r^{2r} \otimes I_{64})(I_r \otimes F_2 \otimes I_{64})(L_r^{2r} \otimes I_{64}). \tag{30}$$

Moreover, both factorizations can easily be implemented with a supervector loop that performs $F_2 \otimes I_{64}$. In both cases the addressing is given by tensor products of loadstrides and storestrides, which are implemented by striding the individual elements $I \otimes L$ or striding the offsets $L \otimes I$. Programs computing these factorizations can be obtained in the same way that we derived programs for scalar machines using loops. The only difference is that some loops are unrolled and replaced by vector instructions.

In order to implement the second factorization we start with the loop implementation of $L_r^{2r} \otimes I_{64}$. Since $(L_r^{2r} \otimes I_{64}) e_i^2 \otimes e_j^r \otimes e_k^{64} = e_j^r \otimes e_i^2 \otimes e_k^{64}$, the index permutation is $(ir + j)64 + k \to (2j + i)64 + k$ and we get the following loop.

$$\text{for } i = 0, \ldots, 1$$
$$\text{for } j = 0, \ldots, r - 1$$
$$\text{for } k = 0, \ldots, 63$$
$$y(2j64 + i64 + k) = x(ir64 + j64 + k)$$

In order to compose this code sequence with a supervector loop implementing $I_r \otimes F_2 \otimes I_{64}$, the outer loop must be unrolled and the inner loop must be replaced with a vector instruction. The resulting supervector loop is

$$\text{for } j = 0, \ldots, r - 1$$
$$Y(2j64) = X(j64)$$
$$Y(2j64 + 64) = X(r64 + j64)$$

where X and Y are vectors of length 64 beginning with the indicated offset. This can be composed with the supervector loop computing $I_r \otimes F_2 \otimes I_{64}$.

$$\text{for } j = 0, \ldots, r - 1$$
$$Y(2j64) = X(2j64) + X((2j + 1)64)$$
$$Y((2j + 1)64) = X(2j64) - X((2j + 1)64)$$

The resulting code can be composed with the output permutation $S_r^{2r} \otimes I_{64}$ to obtain

$$\text{for } j = 0, \ldots, r - 1$$
$$Y(j64) = X(j64) + X((r + j)64)$$
$$Y((r + j)64) = X(j64) - X((r + j)64)$$

which can be implemented on the X-MP with a supervector loop.

$$\text{for } j = 0, \ldots, r - 1$$

```
V0   , X_{j64}, 1
V1   , X_{(j+r)64}, 1
V2   V0 + FV1
V3   V0 - FV1
,  Y_{j64}, 1    V2
,  Y_{(r+j)64}, 1   V3
```

The other factorization can be implemented in the same way. The only new phenomenon is the composition of the two permutations $(L_r^{2r} \otimes I_{64})(I_2 \otimes L_r^{64r})$ and their implementation on the X-MP. The permutation $I_2 \otimes L_r^{64r}$ can be programmed with the following loop.

$$
\begin{aligned}
&\text{for } j = 0, \ldots, r - 1 \\
&\quad \text{for } k = 0, \ldots, 63 \\
&\qquad y(j64 + k) = x(kr + j) \\
&\qquad y((r + j)64 + k) = x((k + 64)r + j)
\end{aligned}
$$

If the inner loop is unrolled it can be implemented with vector loads at stride r.

$$
\begin{aligned}
&\text{for } j = 0, \ldots, r - 1 \\
&\quad Y_{64j} \leftarrow X_j, r \\
&\quad Y_{(r+j)64} \leftarrow X_{j+64r}, r
\end{aligned}
$$

Combining this with the other code sequences gives the following Cray implementation.

$$
\begin{aligned}
&\text{for } j = 0, \ldots, r - 1 \\
&\quad V0 \quad , X_j, r \\
&\quad V1 \quad , X_{j+64r}, r \\
&\quad V2 \quad V0 + FV1 \\
&\quad V3 \quad V0 - FV1 \\
&\quad , Y_j, r \quad V2 \\
&\quad , Y_{j+64r}, r \quad V3
\end{aligned}
$$

We end this section by producing a vectorized segmented factorization of $F_{2,n+6}$ which can be programmed on the X-MP using some of the techniques that we have discussed. We begin with the vector factorization from Theorem 8:

$$
F_{2,n+6} = \prod_{i=1}^{n+6} ((F_2 \otimes I_{N/2})L_2^N).
$$

This factorization can be conjugated to obtain

$$
F_{2,n+6} = \prod_{i=1}^{n+6} (S_{2^n}^N(I_{2^n} \otimes F_2 \otimes I_{64})L_2^N L_{2^n}^N) \tag{31}
$$

$$
= S_{2^n}^N \left\{ \prod_{i=1}^{n+6} ((I_{2^n} \otimes F_2 \otimes I_{64})L_2^N) \right\} L_{2^n}^N. \tag{32}
$$

The permutation L_2^N in each stage of the product can be factored as in our example; therefore, up to relabeling of the input and output, we get a segmented vectorized algorithm for the X-MP. Alternatively L_2^N and $L_{2^n}^N$ can be combined to get $L_{2^{n+1}}^N$ which produces appropriately sized vectors. However, in this case, $S_{2^n}^N$ must be factored to operate on vectors of length 64.

5. What is the finite FT?

Let f be a complex function. If we sample f at n points and assume that f is periodic outside the sample, we get a function $f: Z/(n) \to C$. We can represent f as an n-tuple of complex numbers

$$
\begin{pmatrix} f(0) \\ f(1) \\ \vdots \\ f(n-1) \end{pmatrix},
$$

corresponding to the values of f. The collection of functions on $Z/(n)$ form a vector space denoted by $L(n)$ which, under the above representation, is isomorphic to C^n. If we define $\langle f, g \rangle = \sum f(i)\overline{g(i)}$, where $\overline{g(i)}$ is the complex conjugate of $g(i)$, $L(n)$ becomes an inner product space denoted by $L^2(n)$.

The finite FT is the linear operator $F_n: L^2(n) \to L^2(n)$ defined by

Definition 3 (FT). $F_n e_i^n = f_i^n$, where $f_i^n = \sum_{j=0}^{n-1} \omega^{ij} e_j^n$, and $\omega = e^{2\pi i/n}$.

The matrix representation is given by $F_n = (\omega^{ij})$, $0 \le j, k < n$. For example,

$$
F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix},
$$

$$
F_n = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)^2} \end{pmatrix}.
$$

The FT of a function $f \in L^2(n)$ is given by the matrix vector multiplication $F_n f$. Note that $1/\sqrt{n} F_n$ is a unitary operator, i.e., $\{1/\sqrt{n} f_i^n\}$ form an

orthonormal basis. To see this,

$$\langle f_i^n, f_j^n \rangle = \sum_{k=0}^{n-1} \omega^{ik}\overline{\omega^{kj}} = \sum \omega^{(i-j)k}$$

$$= \begin{cases} n & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

Therefore, we have the following properties:

1. $F_n F_n^* = n I_n$ and $F_n^{-1} = \frac{1}{n} F_n^*$, where * denote the conjugate transpose,
2. $1/n F_n^2 e_i^n = 1/n \sum_{j=0}^{n-1} \langle f_i^n, \overline{f_j^n} \rangle e_j^n = e_{n-i}^n$,
3. $F_n^4 = n^2 I_n$.

These properties can be used as tools for debugging FT algorithms. The main idea is to try the program on various bases.

5.1. *An algorithm for computing the FT*

The Fourier matrix has many redundancies. If we take advantage of this redundancy we naturally arrive at a matrix factorization of F_n that allows us to compute its action on a vector efficiently. For example, the computation $y = F_4 x$ can be performed more efficiently if we notice that $t_0 = x_0 + x_2$, $t_1 = x_0 - x_2$, $t_2 = x_1 + x_3$, and $t_3 = x_1 - x_3$ only need to be computed once. In terms of the temporary values, $y_0 = t_0 + t_2$, $y_1 = t_1 + it_3$, $y_2 = t_0 - t_2$, and $y_3 = t_1 - it_3$. This observation implies the matrix factorization

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & i \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & i \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -i \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix}.$$

If we collect the multiplication by i we introduce a diagonal matrix in the factorization:

$$F_4 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix}$$

$$\times \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We can rewrite this factorization using tensor-product notation as $F_4 = (F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)L_2^4$, where T_2^4 is the diagonal matrix $\mathrm{diag}(1, 1, 1, i)$.

We now carry out this factorization for another example. Namely

$$
F_8 = \begin{vmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\
1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\
1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\
1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\
1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\
1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\
1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega
\end{vmatrix},
$$

where ω is a primitive eighth root of unity. This can be factored as

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & \omega & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & \omega^2 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & \omega^3 \\
1 & 0 & 0 & 0 & \omega^4 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & \omega^5 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & \omega^6 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & \omega^7
\end{pmatrix}
\begin{pmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & \omega^2 & \omega^4 & \omega^6 & 0 & 0 & 0 & 0 \\
1 & \omega^4 & 1 & \omega^4 & 0 & 0 & 0 & 0 \\
1 & \omega^6 & \omega^4 & \omega^2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & \omega^2 & \omega^4 & \omega^6 \\
0 & 0 & 0 & 0 & 1 & \omega^4 & 1 & \omega^4 \\
0 & 0 & 0 & 0 & 1 & \omega^6 & \omega^4 & \omega^2
\end{pmatrix}
L_2^8.
$$

If we look at the eighth roots of unity, we see that $(1, \omega^2, \omega^4, \omega^6)$ are the fourth roots of unity. In particular we have $\omega^4 = -1$. Using this information we get the factorization

$$
F_8 = (F_2 \otimes I_4)T_4^8(I_2 \otimes F_4)L_2^8.
$$

where T_4^8 is the diagonal matrix $\mathrm{diag}(1, 1, 1, 1, 1, \omega, \omega^2, \omega^3)$.

Definition 4 (Twiddle Factors). $T_s^{rs} e_i^r \otimes e_j^s = \omega^{ij} e_i^r \otimes e_j^s$, where ω is a primitive nth root of unity, where $n = rs$.

The definition implies that $T_s^{rs} = \bigoplus_{i=0}^{r-1} (D_s^{rs})^i$, the direct sum of powers of the diagonal matrix $D_s^{rs} = \mathrm{diag}(1, \omega, \ldots, \omega^{s-1})$. Also, immediately from the definition, we get the following change of basis theorem.

Theorem 9. $T_r^{sr} = L_s^{rs} T_s^{rs}(L_s^{rs})^{-1} = L_s^{rs} T_s^{rs} L_r^{sr}$.

The following theorem is the basis of all Cooley-Tukey (C-T)-type FT algorithms. The basic divide-and-conquer idea behind this theorem and

the means of computation it implies was first presented in the fundamental paper [6].

Theorem 10 (Cooley–Tukey).

$$F_{rs} = (F_r \otimes I_s)T_s^{sr}(I_r \otimes F_s)L_r^{sr}.$$

Proof. The proof involves the defining properties of tensor products and the associated matrices that have been presented. To see this, compute both sides on the basis elements $e_i^s \otimes e_j^r$:

$$(F_r \otimes I_s)T_s^{rs}(I_r \otimes F_s)L_r^{rs}(e_i^s \otimes e_j^r) = (F_r \otimes I_s)T_s^{rs}(I_r \otimes F_s)(e_j^r \otimes e_i^s)$$

$$= (F_r \otimes I_s)T_s^{rs}(e_j^r \otimes f_i^s)$$

$$= (F_r \otimes I_s)T_s^{rs}\left(\sum_{k=0}^{s-1} \omega^{rik}(e_j^r \otimes e_k^s)\right)$$

$$= (F_r \otimes I_s)\sum_{k=0}^{s-1} \omega^{rik+jk}(e_j^r \otimes e_k^s)$$

$$= \sum_{k=0}^{s-1} \omega^{rik+jk}(f_j^r \otimes e_k^s)$$

$$= \sum_{k=0}^{s-1}\sum_{l=0}^{r-1} \omega^{rik+sjl+jk}(e_l^r \otimes e_k^s).$$

since ω is an rsth root of unity, $\omega^{rik+sjl+jk} = \omega^{rik+sjl+jk+rils} = \omega^{(ri+j)(sl+k)}$. Using this observation along with the map $e_l^r \otimes e_k^s \to e_{ls+k}^{rs}$, shows that the last sum in the derivation is f_{ri+j}^n and the theorem is proved. □

The factorization in the preceding theorem decomposes the n-point FT into four stages, the stride permutation L_r^n, the parallel operation $I_r \otimes F_s$, the diagonal matrix multiplication T_r^n, and the vector operation $F_r \otimes I_s$. The only new part in this computation is the diagonal matrix of twiddle factors. From our previous discussions on tensor products and stride permutations, we know how to implement the other components. Implementation of the twiddle factors can be handled in much the same way. We carry out an example that gives the main ideas involved.

In the factorization of F_8 the two factors $(F_2 \otimes I_4)$ and T_4^8 can be combined and performed as a vector butterfly. If W is a vector containing the first four roots of unity $(1, \omega, \omega^2, \omega^3)$, $X_0 = (x_0, x_1, x_2, x_3)$, $X_1 = (x_4, x_5, x_6, x_7)$, and similarly for Y_0 and Y_1, then the computation $Y = (F_2 \otimes I_4)T_2^8 X$ can be performed with the vector operations $Y_0 = X_0 + WX_1$ and $Y_1 = X_0 - WX_1$. The important observation is that the diagonal multiplication can be combined with the vector operation $F_2 \otimes I_4$. However, WX_1 should be stored in a temporary vector, so that the multiplication is only performed

once. If vector instructions are not available to perform this operation, then it can be converted to a loop with the commutation theorem. After applying the commutation theorem, we need to perform $Y = S_4^8(I_4 \otimes F_2)L_4^8 T_4^8 X$. As in the construction of programs for computing with tensor products, the code can be optimized by composing the four operations.

The diagonal multiplication of the input by T_4^8 can be carried out with the following loop obtained from the definition.

$$\text{for } i = 0, \ldots, 1$$
$$\quad \text{for } j = 0, \ldots, 3$$
$$\quad\quad y(4i + j) = W(ij)x(4i + j)$$

In order to combine this with the following permutation, we need to unroll the loop.

$$\text{for } j = 0, \ldots, 3$$
$$\quad y(j) = x(j)$$
$$\quad y(4 + j) = W(j)x(4 + j)$$

This can be combined with the code for L_4^8.

$$\text{for } j = 0, \ldots, 3$$
$$\quad z(2j) = x(j)$$
$$\quad z(2j + 1) = W(j)x(4 + j)$$

The resulting code can then be combined with the loop implementing $I_4 \otimes F_2$,

$$\text{for } j = 0, \ldots, 3$$
$$\quad w(2j) = x(j) + W(j)x(4 + j)$$
$$\quad w(2j + 1) = x(j) - W(j)x(4 + j)$$

which, when combined with the output permutation S_4^8, give the final code segment.

$$\text{for } j = 0, \ldots, 3$$
$$\quad u(j) = x(j) + W(j)x(4 + j)$$
$$\quad u(j + 4) = x(j) - W(j)x(4 + j)$$

This sequence can be further optimized if the common data $W(j)x(4 + j)$ is collected together in a temporary variable. This optimization is suggested by the separate diagonal factor, which makes explicit the temporary allocation.

As in the case of tensor-product factorizations, variations can be obtained by applying the commutation theorem. Two such factorizations are

$$F_{rs} = L_r^{rs}(I_s \otimes F_r)L_s^{rs}T_s^{rs}(I_r \otimes F_s)L_r^{rs}, \tag{33}$$

$$F_{rs} = (F_r \otimes I_s)T_s^{rs}L_r^{rs}(F_s \otimes I_r). \tag{34}$$

The first factorization is a parallel factorization and the second factorization is a vector factorization. Furthermore, the order of the data permutation and the twiddle factor can be interchanged by applying Theorem 9. Finally, since F_n is symmetric, taking the transpose of any factorization leads to a new factorization. For example, taking the transpose of the initial factorization in the C–T theorem gives

$$F_{rs} = L_s^{rs}(I_r \otimes F_s)T_s^{rs}(F_r \otimes I_s). \tag{35}$$

Since this factorization has the permutation on output it is commonly called a decimation-in-frequency algorithm, while the factorization with the permutation on input is called a decimation-in-time algorithm. All of these variations offer programming options that may be used to advantage on some architectures.

When n has more than two factors, Theorem 10 can be repeatedly used to obtain an algorithm for computing F_n. For example,

$$\begin{aligned}
F_8 &= (F_2 \otimes I_4)T_4^8(I_2 \otimes F_4)L_2^8 \\
&= (F_2 \otimes I_4)T_4^8(I_2 \otimes ((F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)L_2^4))L_2^8 \\
&= (F_2 \otimes I_4)T_4^8(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_4 \otimes F_2)(I_2 \otimes L_2^4)L_2^8.
\end{aligned}$$

The permutation $R_8 = (I_2 \otimes L_2^4)L_2^8$ is uniquely defined by $R_8(e_i^2 \otimes e_j^2 \otimes e_k^2) = e_k^2 \otimes e_j^2 \otimes e_i^2$. R_8 is called a bit reversal permutation, since it permutes the indexing set by mapping an index to a number whose binary representation is the reverse of the binary representation of the index. To see this, the defining condition of R_8 implies that $e_{4i+2j+k}^8 \to e_{4k+2j+i}^8$. In general, the bit reversal permutation is defined as

Definition 5 (Bit Reversal). $R_{2^n}(e_{i_1}^2 \otimes e_{i_2}^2 \otimes \cdots \otimes e_{i_n}^2) = e_{i_n}^2 \otimes \cdots \otimes e_{i_2}^2 \otimes e_{i_1}^2$. (It reverses the order of the factors in the tensor-product basis.)

The bit reversal permutation satisfies the following recursion.

Theorem 11.

$$R_{2^n} = (I_2 \otimes R_{2^{n-1}})L_2^{2^n}.$$

Proof. Compute the action of both sides on a basis element in the tensor basis. $\qquad \square$

If we repeatedly apply C–T factorization to F_N where $N = 2^n$, and use the recursion property of the bit reversal permutation, we get the following factorization known as the fast Fourier transform (FFT).

Theorem 12 (FFT).

$$F_{2^n} = \left\{ \prod_{i=1}^{n} (I_{2^{i-1}} \otimes F_2 \otimes I_{2^{n-i}})(I_{2^{i-1}} \otimes T_{2^{n-i}}^{2^{n-i+1}}) \right\} R_{2^n}.$$

This factorization can be implemented using the techniques for implementing tensor products and twiddle factors. The recursive property of the bit reversal permutation can be used to obtain code implementing it. Moreover, tensor-product identities can be used to modify this factorization, in the same way that tensor-product factorization were modified, to obtain programs more suitable to various architectures. In the next section we briefly list some of these variations.

5.2. Variations on C–T: parallelized and vectorized FT algorithms

In this section variants of the C–T FFT are derived. In particular, we obtain algorithms that display complete vectorization and parallelism. Finally, we give one example of a segmented algorithm that is similar to the segmented tensor-product algorithm developed for the Cray. Despite the added complexity of some diagonal matrices and the bit reversal permutation, the same techniques that we used for modifying tensor-product factorizations carry through here.

We begin by re-examining F_4 and F_8. Applying the commutation theorem to the C–T factorization of F_4 we get

$$F_4 = (F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4.$$

We use this form of F_4 as the starting point for recursively deriving a vector FT algorithm. We see how this is done by looking at F_8:

$$F_8 = (F_2 \otimes I_4) T_4^8 (I_2 \otimes F_4) L_2^8$$
$$= (F_2 \otimes I_4) T_4^8 L_2^8 (F_4 \otimes I_2).$$

Substituting the previous form of F_4 into this equation and using the multiplicative rule for tensor products gives the vector algorithm

$$F_8 = (F_2 \otimes I_4) T_4^8 L_2^8 (F_2 \otimes I_4)(T_2^4 \otimes I_2)(L_2^4 \otimes I_2)(F_2 \otimes I_4).$$

Generalizing this computation we get the following theorem due to T. G. Stockham which is described in [5].

Theorem 13 (Stockham). *If $N = 2^n$ and $N(i) = 2^i$, then*

$$F_N = \prod_{i=1}^{n} (F_2 \otimes I_{N/2})(T_{N/2N(i-1)}^{N/N(i-1)} \otimes I_{N(i-1)})(L_2^{N/N(i-1)} \otimes I_{N(i-1)}).$$

This factorization obtains full vectorization. Also the initial bit reversal permutation has been spread through the computation. In fact this algorithm can be derived by starting with the FFT and bringing the bit reversal permutation into the computation. It is this property that makes this algorithm preferable on some vector machines where the bit reversal permutation is hard to implement. Temperton discusses some of the implementation issues of this algorithm and mixed radix generalization in [19]. The effectiveness of this algorithm depends on the ability to implement the permutations $L_2^{N/N(i-1)} \otimes I_{N(i-1)}$. The price of the removal of the bit reversal permutation is the irregular data flow between each of the stages.

A second variation can be obtained that has full vectorization and regular data flow, but keeps the bit reversal permutation. This algorithm is analagous to the vectorized tensor-product factorization in Theorem 8.

To obtain this algorithm, we start with the FFT and apply the commutation theorem to obtain factors of the form $F_2 \otimes I_{N/2}$ as was done in the tensor-product case. The only difference is the twiddle factors, which get commuted by the stride permutations. Before stating this theorem, we look at an example:

$$
\begin{aligned}
F_8 &= (F_2 \otimes I_4)T_4^8(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_4 \otimes F_2)R_8 \\
&= (F_2 \otimes I_4)T_4^8 L_2^8(F_2 \otimes I_4)L_4^8(I_2 \otimes T_2^4)L_4^8(F_2 \otimes I_4)L_2^8 R_8 \\
&= (F_2 \otimes I_4)T_4^8 L_2^8(F_2 \otimes I_4)(T_2^4 \otimes I_2)L_2^8(F_2 \otimes I_4)L_2^8 R_8.
\end{aligned}
$$

We now state the vectorized FFT, which was first described by Korn–Lambiotte in [11], where they discuss implementation of vector FFTs on the STAR 100 computer.

Theorem 14 (Korn–Lambiotte). *If $N = 2^n$, then*

$$
F_N = \left\{ \prod_{i=1}^{n} (F_2 \otimes I_{N/2})(T_{N/2N(i-1)}^{N/N(i-1)} \otimes I_{N(i-1)})L_2^N \right\} R_N.
$$

Important points about this factorization are the constant data flow at each stage given by the simple permutation L_2^N. This algorithm should be well suited to vector processors. The only possible difficulties are the permuted diagonal and the initial bit reversal. Both issues warrant further study. The algorithm can be segmented in the same way that the tensor-product factorization was segmented; however, the notation for the twiddle factors must be refined if we are to write it down.

The vector algorithm just presented was a modification of a parallel algorithm originally developed by Pease in [13]. To get a similar algorithm to that of Pease, we can apply the commutation theorem.

Theorem 15 (Pease). *If $N = 2^n$, then*

$$F_N = R_N \prod_{i=1}^{n} L_2^N (I_{N/2} \otimes F_2) T_i'.$$

where $T_i' = L_{N/2}^N (T_{N/2N(i-1)}^{N/N(i-1)} \otimes I_{N(i-1)}) L_2^N$.

The algorithms that we have presented so far are the major types of vector and parallel algorithms for computing the FT. As in the case of tensor-product algorithms many minor variations can be derived. These variations allow the algorithm to be fine tuned to a specific architecture. Along these lines, we have only indicated how to derive a segmented vectorized algorithm for the Cray X-MP. However, the tools presented earlier in this paper can be used to obtain any desired variation based on the C-T theorem. An important class of variations can be derived from the mixed radix generalizations of the algorithms presented in this section. These generalizations are discussed in [4] and [19]. The importance of the mixed radix cases come from deriving algorithms that increase local computation. As was pointed out in Section 4.3, in the discussion on modifying tensor-product implementations, the mixed radix cases are useful when the size of the local computation does not divide the number of points in the FT.

The major point of this section is the ease with which variations of the FFT algorithm can be derived using the tensor-product formulation. The tools developed in the sections on tensor products give a mechanism for carrying out these modifications and implementing them. It is clear that we would like to have these tools automated so that different variations could be derived, implemented, and tested on various architectures.

6. Some notes on code generation and special purpose compiler

We believe that tensor-product formulation of FT algorithms allows easy manipulation of algorithms to get variations that might be better suited to different architectures. We also indicated that the formulation contains information that is relevant to architectural features of various machines. Furthermore, we showed that mathematical formulation can aid in the implementation of the algorithms. However, if the user is going to be able to use these techniques for selecting, modifying, and implementing algorithms easily, then the process needs to be automated. This is especially true for the code generation phase. The user might be able to guide the mathematical formulation and the heuristics for selecting a specific variation. Nonetheless, there are thousands of choices and the user should be able to try variations quickly without specifying the details.

The first step would be to create a parser, which determines if an algorithm described in tensor-product notation actually computes the desired FT. The

parser should also be interfaced with a code generator. The code generator should produce intermediate code, which can then be sent to a machine specific code generator, which produces the actual code. In this way the same basic code generator could be used for a variety of architectures. Finally, we would like to put heuristics into this special purpose compiler, so that the compiler could automatically select an appropriate algorithm for a given architecture. The architecture should be described in terms of some key parameters such as the size of vector registers, the available special purpose instructions, and the types of permutations that can be efficiently implemented.

In this section we sketch a methodology [10] for automatic algorithm derivation and code generation. The basic idea is to incorporate a set of production rules based on algebraic decompositions and tensor-product algebra into an attribute grammar used to generate the code. The attribute grammar produces a parse tree corresponding to an algorithm for computing the FT F_n. The leaf nodes of the tree correspond to macros that produce common code sequences such as F_2 and the internal nodes correspond to various algebraic operations, such as direct sums, matrix multiplications, and tensor products, used in the derivation of the algorithm. The code for F_n is produced by performing the algebraic operations on the code sequences given by the macros. The code is synthesized up the tree until the root node is reached and the code for F_n is produced.

It should be pointed out that various algebraic operations (commuting tensor products) introduce permutations that need to be incorporated into the addressing of the data. This information is stored as an attribute which is passed down the tree. Permutations get pushed down the tree by applying the algebraic operations at the internal nodes to the permutations themselves. For example, we can take the tensor product of two permutations or we can multiply the permutations together. If the permutations are of a special form, such as stride permutations, we have special rules for applying these operations.

In order to make these ideas a little clearer, we give some examples. First we give some typical production rules. Some rules corresponding to tensor algebra are

$$A_m \otimes B_n \rightarrow L_m^{mn}(B_n \otimes A_m)L_n^{mn}. \tag{36}$$
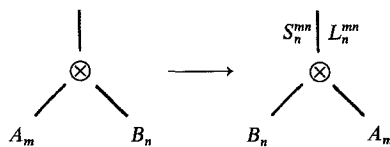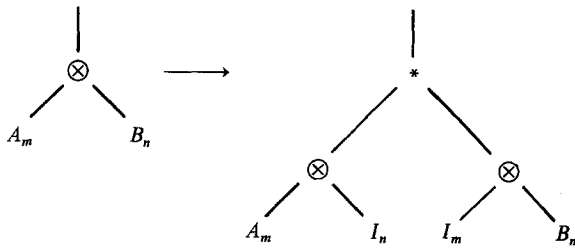

Figure 3. Commutation theorem.

Figure 4. Tensor-product decomposition.

This production rule can be depicted by its action on the parse tree shown in Figure 3. A second important example is given by the tensor-product decomposition

$$A_m \otimes B_n \to (A_m \otimes I_n)(I_m \otimes B_n). \tag{37}$$

This is depicted in the parse tree in Figure 4.

An example production based on algebraic theorems concerning the FT is the following rule corresponding to the C–T theorem,

$$F_{rs} \to (F_r \otimes I_s)T_s^{rs}(I_r \otimes F_s)L_r^{rs} \tag{38}$$

which is diagrammed in Figure 5.

Some possible macros for productions of these types would be $F_2 \otimes I_n$, $I_n \otimes F_2$, and multiplication by roots of unity. It is clear that different macros would have to be given for different machines. Of special interest are vector macros, based on ideas given earlier in the section on the Cray.

Once we have appropriate macros, in order to produce the final code, we need to be able to combine code sequences based on the algebraic operations given in the parse tree. Some useful ideas along these lines were discussed in Section 4.2 on implementing tensor products using loops. In fact if our macros are based on the loop implementation discussed in that section, then
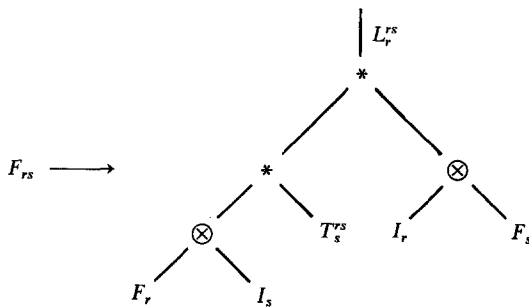

Figure 5. C–T theorem.

we already have an automated way of performing the algebraic operations of interest on those macros. Furthermore, by unrolling loops to match special instructions, as was done to implement stride permutations on the Cray with vector loads, we have a means of converting loop macros to machine-specific macros.

We now give a general framework for how algebraic operations are performed on code sequences. If we let $[A]$ be the code to evaluate $y = Ax$ and $[B]$ be the code to evaluate $z = Bw$, then the code to evaluate the direct sum $A \oplus B$ is just the concatenation of $[A]$ and $[B]$ with the appropriate partitioning of the input and output variables. Likewise, the code to evaluate $y = ABw$ is the concatenation of the codes for A and B, so that temporaries are introduced that pass the output of B to the input of A. Finally, the code for the tensor product can be created from these two constructions using the identity $A \otimes B = (A \otimes I_n)(I_m \otimes B)$ and $I_m \otimes B = \bigoplus_{i=1}^m B$. Therefore, $A \otimes B = L_m^{mn}(I_n \otimes A)L_n^{mn}(I_m \otimes B)$ which gets expanded to $L_m^{mn}(\bigoplus_{i=1}^n A)L_n^{mn}(\bigoplus_{j=1}^m B)$. This is just n copies of $[A]$ followed by m copies of $[B]$ with appropriate permutations in the addressing. If we had procedures to implement $A \otimes I$ or $I \otimes B$ we could implement the tensor product more directly.

Finally, we need some mechanism to guide the application of production rules to give a normal form or unique representation of the parse tree. These rules could be changed for different architectures, or for different passes on the same architecture. This would be useful for trying different algorithms. One possible idea is to use heuristics that choose certain rules over others. This has been examined for the code generation of certain FT algorithms on the VAX [10]. Another alternative would be to force the resulting tree to have certain properties such as tensor-product operations of the form $A \otimes I$ with the vector size matching the vector registers. In this case the compiler would search for the appropriate derivation of an appropriate goal tree.

7. Summary

The tools that have been presented in this paper give a methodology for implementing FTs. The following outlines the methodology and shows how the techniques in this paper can be used. The procedure has been carried out in Section 4.4, using the Cray X-MP as an example.

Given an architecture on which to implement an FT, proceed as follows:

1. Consider how to implement tensor-product construction and the associated permutations (see Section 4 for general techniques and Section 4.4 for a specific example).
2. Decide which constructions have an efficient implementation (see Sections 4.3 and 4.4).

3. Formulate an appropriate FT algorithm using tensor-product notation (see Sections 5.1, 5.2, and [14]). Using tensor algebra (see Sections 2 and 3) manipulate the algorithm to achieve operations that have an effective implementation.
4. Implement the expression obtained in (3) using the techniques developed in (1). The Cray example given in Section 4.4 carries this out to a pseudoassembly implementation.

The last step should be automated so that a code for an expression involving tensor products is produced from the implementation of the basic tensor-product operations given in the first step. Once this step is automated, programming can be carried out using a mathematical description of the FT based on tensor products. This paper gives the basic ideas behind the automation.

The benefits of this design procedure are that a wide class of FT algorithms can easily be programmed on any architecture once the basic tensor-product operations have been implemented. Moreover, the same basic procedure can be followed independent of the architecture. Finally, optimizations can be obtained at a high level based on properties of the FT. For example, vectorization and segmentation were obtained for the Cray X-MP using tensor-product properties. With the inclusion of these optimizations, and the ability to design specific FTs for specific architectures, we believe that the methodology presented in this paper can be used to obtain programs that are as efficient as optimized hand-coded implementations. Moreover, the development time should be significantly reduced.

Even when the procedure is not automated, the tools can help produce efficient programs in a systematic manner. In the Appendix we review the application of this methodology to AT&T's DSP32 signal processing chip.

In steps one and two of the outline we study the special features of the chip's instruction set that can be used to implement tensor products. In this case there are no special vector or parallel operations; however, the chip has special features for loop instructions. Therefore we choose a basic loop implementation of tensor products along the lines of Section 4. Furthermore, the looping constructs incorporate various memory accesses at strides, which can be used to implement stride permutations.

In step three we formulate a tensor-product version of the C–T algorithm. The only modifications to the basic algorithm are due to the number of registers. A radix eight FFT was chosen so that the building blocks of the tensor-product decomposition could fit inside the register set and be performed efficiently. Further optimizations were obtained by incorporating some of the stride permutations into the loops implementing the tensor products. This formulation could then be implemented by translating the tensor-product operations into loop constructs. Even though this translation process has not been automated yet, it is straightforward to carry it out by

hand. Thus the programming process is governed by the tensor-product notation used to describe the algorithm. In [7] Granata and Rofheart carried out this process and obtained an implementation that was roughly twice as fast as the standard library routines.

Since this methodology was carried out by hand, only several algorithms were tried. If steps three and four could be done by a compiler, as suggested in Section 6, many other variations could have been tried, possibly obtaining an even faster program. Furthermore, the compiler would eliminate the potential for translation errors that can easily occur when this is done by hand. Finally, given a set of heuristics, the compiler could generate a class of algorithms suitable for the DSP32. These algorithms could then be timed and the fastest chosen.

Appendix. A loop implementation of the FT on the AT&T DSP32

Using techniques similar to those described in this paper, Granata and Rofheart [7] have implemented efficient FTs for 8, 16, ..., 1024 points on the AT&T DSP32 signal processing chip. Their implementation was roughly twice as fast as the "standard" library implementation and is now being distributed as the new "standard" library routines.

The DSP32 is a single chip, programmable digital signal processor, developed by AT&T Bell Laboratories [3]. It has three architectural features that are significant in choosing an implementation of the FT.

1. A 32 bit floating-point multiply–accumulate instruction.
2. A four-stage pipeline that effectively increases the number of floating-point registers.
3. Two separate execution units: one for floating-point arithmetic (DAU) and one for address generation and control (CAU) which operate in parallel.

All three features are used in the Granata–Rofheart implementation. We discuss in detail only the implication of the third feature.

The general DAU instruction has the form

```
(Z=) aN={-}aM{+,-}Y*X
```

or

```
aN={-}aM{+,-}(Z=Y)*X
```

where X, Y, Z are general operands and aN and aM are one of the four accumulators in the DAU. In the instruction, the parentheses indicate that Z is optional in the position indicated, and the braces indicate that the enclosed operator is optional or a choice must be made. X, Y, Z can be operands in

memory whose addresses are developed in the CAU. The addresses can be one of the following forms where the 16 bit CAU registers rP and rI are defined for $P = 1, \ldots, 14; I = 15, \ldots, 19$.

1. *rP where rP contains the address of the operand.
2. *rP++(*rP--) where rP is to be post incremented (decremented).
3. *rP++rI where rP is post incremented by the contents of rI.

Now every DAU instruction is executed in four states during which three memory reads (fetch, X, Y) and one memory write (Z) are possible. These are "free" in the sense that the instruction will take the same amount of time to execute regardless of the number of memory operations used. In the same sense, the address calculations implied by the various addressing modes are also free. They are executed in the CAU in parallel with the DAU.

These addressing features can be used to implement the following loop constructs at the same cost. Each loop is given in pseudo-Pascal along with its implementation on the DSP32:

$$\text{for } i = 0, \ldots, n - 1$$
$$y(i) = x(i)$$

```
CNT=n-1
r1=X
r2=Y

LOOP:   *r2++=a0=*r1++
        if (CNT-- > =0) goto LOOP
```

Without the free post increment two extra increment instructions would be required. The next loop incorporates a constant stride different than 1. Such a loop can be written as

$$\text{for } i = 0, 1, \ldots, (n - 1)$$
$$y(di) = x(di)$$

or alternatively

$$\text{for } i = 0, d, \ldots, d(n - 1)$$
$$y(i) = x(i)$$

These loops would be implemented on the DSP with

```
CNT=n-1
r1=X
r2=Y
r15=d
LOOP:   *r2++r15=a0=*r1++r15
        if (CNT-- > =0) goto LOOP
```

at no extra cost than the preceding loop. The last loop can be further generalized so that the input and output strides are different. In this case we have

$$\text{for } i = 0, 1, \ldots, (n - 1)$$
$$y(bi) = x(ai)$$

which again gets implemented with no extra cost:

```
CNT=n-1
r1=X
r2=Y
r15=a
r16=b
LOOP:  *r2++r15=a0=*r1++r16
       if (CNT-- > =0) goto LOOP
```

The most general loop construct described allows us to implement efficiently the parametrized macro $F2(X, a, Y, b)$ which has different input and output strides. This macro can then be incorporated in a loop obtained from $I_n \otimes F_2$. In the implementation of the FT on the DSP, Granata and Rofheart chose F_8 as the base macro instead of F_2. The reason for this choice is that a multiplicative Winograd F_8 can be used which takes advantage of the multiply–accumulate instruction.

We begin with a radix 8 factorization of F_{1024}:

$$F_{1024} = (F_2 \otimes I_{512})T_{512}^{1024}$$
$$(I_2 \otimes F_8 \otimes I_{64})(I_2 \otimes T_{64}^{512})$$
$$(I_2 \otimes I_8 \otimes F_8 \otimes I_8)(I_2 \otimes I_8 \otimes T_8^{64})$$
$$(I_2 \otimes I_8 \otimes F_8)(I_2 \otimes I_8 \otimes L_8^{64})(I_2 \otimes L_8^{512})L_2^{1024}.$$

This factorization can be rewritten to obtain a direct loop implementation:

$$F_{1024} = S_{512}^{1024}(I_{512} \otimes F_2)L_{512}^{1024}T_{512}^{1024}$$
$$(I_2 \otimes S_{64}^{512}(I_{64} \otimes F_8)L_{64}^{512})(I_2 \otimes T_{64}^{512})$$
$$(I_{16} \otimes S_8^{64}(I_8 \otimes F_8)L_8^{64}(I_{16} \otimes T_8^{64})$$
$$(I_2 \otimes I_8 \otimes I_8 \otimes F_8)(I_2 \otimes I_8 \otimes L_8^{64})(I_2 \otimes L_8^{512})L_2^{1024}.$$

The various stages in this factorization can be composed at compile time to obtain an efficient implementation. Most importantly, the initial generalized bit reversal permutation should be incorporated into the addressing of $I_2 \otimes I_8 \otimes I_8 \otimes F_8$. Since the bit reversal permutation takes $e_{j_1}^8 \otimes e_{j_2}^8 \otimes e_{j_3}^8 \otimes e_{j_4}^2$ to $e_{j_4}^2 \otimes e_{j_3}^8 \otimes e_{j_2}^8 \otimes e_{j_1}^8$, we get the following loop implementation of the

permutation.

$\quad$ for $j_4 = 0, \ldots, 1$

$\qquad$ for $j_3 = 0, \ldots, 7$

$\qquad\quad$ for $j_2 = 0, \ldots, 7$

$\qquad\qquad$ for $j_1 = 0, \ldots, 7$

$\qquad\qquad\quad y(512j_4 + 64j_3 + 8j_2 + j_1) = x(128j_1 + 16j_2 + 2j_3 + j_4)$

This permutation can be composed with the preceding stage to obtain the following loop.

$\quad$ for $j_4 = 0, \ldots, 1$

$\qquad$ for $j_3 = 0, \ldots, 7$

$\qquad\quad$ for $j_2 = 0, \ldots, 7$

$\qquad\qquad F8(y(512j_4 + 64j_3 + 8j_2), 1, x(16j_2 + 2j_3 + j_4), 128)$

It is only the inner loop that can take special advantage of the DSP instruction set. However, in this case, the most general loop is required since the input and output strides are different.

$\quad$ Rather than doing the multiplications required in the remaining part of the address calculation, we can obtain the same effect by adding a constant stride for each of the nested loops. This implementation corresponds to composing each factor in the bit reversal permutation separately into the FT stage. For example, the j_2 loop can be combined with $I_2 \otimes L_8^{512}$ with the use of the macro $IF8(n, y, b, t, x, a, s)$. This macro has parameters for two input and output strides, one stride for the individual elments of F_8 and another stride for the base address of the input and output to F_8. Using this macro we have the following double loop equivalent to the preceding triple loop. The only difference is how the addresses are computed.

$\qquad$ for $j_4 = 0, \ldots, 1$

$\qquad\quad$ for $j_3 = 0, \ldots, 7$

$\qquad\qquad F8(y(512j_4 + 64j_3), 8, 1, x(2j_3 + j_4), 16, 128)$

This process can be continued to eliminate the remaining multiplications in the address computation.

$\quad$ A macro of the preceding form can be used in the implementation of the remaining FT stages, since each stage is of the form $I_n \otimes F$. The implementation of the remaining stages has already been discussed in the section on programming tensor products and the section on the C–T algorithm, where the implementation of twiddle factors was discussed. Since the twiddle factors can be composed with FT stages, the original factorization, which needs eight run-time stages, can now be computed with four stages after applying the compile-time optimizations. Finally, we should mention that the

implementation of the bit reversal permutation takes advantage of the DSP architecture in one of the three nested loops needed to implement it. This suggests that an alternative factorization might be tried, where the bit reversal is brought into the computation and only one permutation of the form $I \otimes L$ is done at each stage.

References

[1] *CAL Assembler Version 1 Reference Manual.* Cray Research, Inc., Mendota Heights, Minnesota, March 1985. SR-0000.

[2] *CRAY X-MP Series Mainframe Reference Manual.* Cray Research, Inc., Mendota Heights, Minnesota, November 1982. HR-0032.

[3] *WE DSP32 Digital Signal Processor —Information Manual.* AT&T Technologies, Inc., 1986.

[4] R. C. Agarwal and J. W. Cooley. Vectorized mixed radix discrete Fourier transform algorithms. January 1987. Preprint.

[5] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenal, W. W. Lang, G. C. Mailing Jr., D. E. Nelson, C. M. Rader, and P. D. Welch. What is the fast Fourier transform? *IEEE Trans. Audio Electroacoust.,* 15(2): 45–55, June 1967.

[6] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Maths. Comp.,* 19(90): 297–301, April 1965.

[7] J. Granata and M. Rofheart. The tensor product as a tool for designing efficient Fourier transform algorithms for digital signal processing. 1988. Unpublished notes.

[8] J. R. Johnson. Some Issues in Designing Algebraic Algorithms for the CRAY X-MP. Technical Report 88-02, Center for Mathematical Computation, University of Delaware, January 1988. Master's Thesis.

[9] J. R. Johnson. Some issues in designing algebraic algorithms for the CRAY X-MP. In *Computer Algebra and Parallelism* (J. Della Dora and J. Fitch, eds.), pp. 179–195, Academic Press, New York, 1989.

[10] R. W. Johnson, C. Lu, and R. Tolimieri. Fast Fourier transform algorithms for the size N, a product of distinct primes and their implementation on VAX architecture. 1989. Unpublished notes.

[11] D. G. Korn and J. J. Lambiotte, Jr. Computing the fast Fourier transform on a vector computer. *Math. comp.,* 33(147): 977–992, July 1979.

[12] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. Comput.,* 24(2); 1145–1155, December 1975.

[13] M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *J. Assoc. Compt. Mach.* 15(2): 252–264, April 1968.

[14] D. Rodriguez. On Tensor Product Formulations of Additive Fast Fourier Transform Algorithms and Their Implementations. Ph.D. thesis, E.E. Department, City College of New York, City University of New York, 1987.

[15] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Comput,* 20(2): 153–161, February 1971.

[16] P. N. Swartztrauber. FFT algorithms for vector computers. *Parallel Comput.,* 1: 45–63, 1984.

[17] P. N. Swartztrauber. Multiprocessor FFTs. *Parallel Comput.,* 5(1 & 2): 197–210, July 1987.

[18] C. Temperton. A note on prime factor FFT algorithms. *J. Comput. Phys.,* 52(1): 198–204, October 1983.

[19] C. Temperton. Self-sorting mixed-radix fast Fourier transforms. *J. Comput. Phys.,* 52(1): 1–23, October 1983.

[20] C. Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *J. Comput. Phys.,* 58(3): 283–299, May 1985.

[21] C. Temperton. Implementation of a prime factor FFT algorithm on the CRAY-1. *Parallel Comput.,* 99–108, 1988.