A Tensor Product Formulation of Strassen's Matrix Multiplication Algorithm with Memory Reduction

B. KUMAR[1], C.-H. HUANG[1], P. SADAYAPPAN[1], AND R.W. JOHNSON[2]

[1]*Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210-1277; e-mail: {kumar-b,chh,saday}@cis.ohio-state.edu*

[2]*Department of Computer Science, St. Cloud State University, St. Cloud, MN 56301; e-mail: rwj@eeyore.stcloud.msus.edu*

ABSTRACT

In this article, we present a program generation strategy of Strassen's matrix multiplication algorithm using a programming methodology based on tensor product formulas. In this methodology, block recursive programs such as the fast Fourier Transforms and Strassen's matrix multiplication algorithm are expressed as algebraic formulas involving tensor products and other matrix operations. Such formulas can be systematically translated to high-performance parallel/vector codes for various architectures. In this article, we present a nonrecursive implementation of Strassen's algorithm for shared memory vector processors such as the Cray Y-MP. A previous implementation of Strassen's algorithm synthesized from tensor product formulas required working storage of size $O(7^n)$ for multiplying $2^n \times 2^n$ matrices. We present a modified formulation in which the working storage requirement is reduced to $O(4^n)$. The modified formulation exhibits sufficient parallelism for efficient implementation on a shared memory multiprocessor. Performance results on a Cray Y-MP8/64 are presented.

1 INTRODUCTION

Tensor products (Kronecker products) have been used to model algorithms with a recursive computational structure that occur in application areas such as digital signal processing [6, 15], image processing [16], linear system design [5], and statistics [7]. In recent years, a programming methodology based on tensor products has been successfully used to design and implement high-performance algorithms to compute fast Fourier Transforms (FFT) [12, 14] and matrix multiplication [10, 13] for shared memory vector multiprocessors. A set of multilinear algebra operations such as tensor product and matrix multiplication are used to express block recursive algorithms. These algebraic operations can be systematically translated into high-level programming language constructs such as sequential composition, iteration, and parallel/vector operations. Tensor product formulas representing an algorithm can be algebraically manipulated to restructure the computation to achieve different performance characteristics. In this way, the algorithm can be tuned to match the underlying architecture.

Matrix multiplication is an important core computation in many scientific applications. Conventional matrix multiplication of $2^n \times 2^n$ matrices requires $O(8^n)$ operations. In 1969, V. Strassen proposed an algorithm for matrix multiplication

Received September 1994
Revised April 1995

Scientific Programming, Vol. 4, pp. 275–289 (1995)
CCC 1058-9244/95/040275-15

[17] that employs a computationally efficient method to compute the product of 2×2 matrices using only seven multiplications. A recursive application of this algorithm for multiplying $2^n \times 2^n$ matrices requires only $O(7^n)$ operations, compared with $O(8^n)$ for conventional matrix multiplication. Efficient parallel implementations of this algorithm have been described in [1, 10]. This algorithm has been used for fast matrix multiplication in implementing Level 3 BLAS [9] and linear algebra routines [2].

In this article, we describe the tensor product formulation of Strassen's matrix multiplication algorithm, and discuss program generation for shared memory vector processors such as the Cray Y-MP. Achieving high performance on these architectures requires operating on large vectors and reducing memory bank conflicts, at the same time exploiting coarse-grained parallelism. We show how the tensor product formula of Strassen's algorithm can be manipulated to operate on full vectors with unit stride. An important feature of the generated code is that it employs no recursion.

The initial formulation presented in [10] required a working array of size $O(7^n)$ for the multiplication of $2^n \times 2^n$ matrices. We present a modified formulation that significantly reduces the size of working array to $O(4^n)$. This reduction is made possible through the reuse of working storage. We describe how this memory reuse can be captured in tensor product formulas with the use of a *selection* operator. We present a strategy for automatic code synthesis from tensor product formulas containing a selection operator. The modified formulation exhibits sufficient parallelism for efficient implementation on a vector-parallel machine such as the Cray Y-MP. In addition, we express Winograd's variation [3] using our notation and describe its translation to a programming code. Winograd's variation uses the same number of multiplications, but a smaller number of additions, than the original Strassen's algorithm.

This article is organized as follows. Section 2 contains an overview of the tensor product notation. A formulation of Strassen's algorithm using this notation is presented in Section 3, along with a discussion on how the formulation can be modified to achieve reduction in working storage. Section 4 presents a strategy for automatic code generation from a tensor product formula. Winograd's variation of the Strassen's algorithm is also presented. Section 5 presents performance results on the Cray Y-MP. Conclusions are presented in Section 6.

2 AN OVERVIEW OF THE TENSOR PRODUCT NOTATION

In this section, we give a brief overview of the tensor product notation and the properties that are used in this article. Let $A \in \mathcal{R}^{m \times n}$ and $B \in \mathcal{R}^{p \times q}$. The tensor product $A \otimes B$ is the block matrix obtained by replacing each element $a_{i,j}$ by the matrix $a_{i,j}B$, i.e.,

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

Whenever all the involved matrix products are valid, the following properties hold:

Property 2.1 (Tensor Product)

1. $A \otimes B \otimes C = A \otimes (B \otimes C) = (A \otimes B) \otimes C$
2. $(A \otimes B)(C \otimes D) = AC \otimes BD$
3. $A \otimes B = (A \otimes I_n)(I_m \otimes B) = (I_m \otimes B)(A \otimes I_n)$
4. $(A \otimes B)^T = A^T \otimes B^T$
5. $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$
6. $(\bigotimes_{i=0}^{n-1} A_i B_i) = (\bigotimes_{i=0}^{n-1} A_i)\,(\bigotimes_{i=0}^{n-1} B_i)$
7. $\prod_{i=0}^{n-1} (A_i \otimes B_i) = \prod_{i=0}^{n-1} A_i \otimes \prod_{i=0}^{n-1} B_i$
8. $I_{mn} = I_m \otimes I_n$

where I_n represents the $n \times n$ identity matrix. $\prod_{i=0}^{n-1} A_i = A_{n-1}A_{n-2} \ldots A_0$, and $\bigotimes_{i=0}^{n-1} A_i = A_{n-1} \otimes A_{n-2} \otimes \ldots \otimes A_0$.

A matrix basis $E_{i,j}^{m,n}$ is an $m \times n$ matirx with a one in the i-th row and the j-th column and zeros elsewhere. A vector basis e_i^m is a column vector of length m with a one in the i-th position and zeros elsewhere. If the basis $E_{i,j}^{m,n}$ of an $m \times n$ matrix is stored by rows, it is isomorphic to the tensor product of two vector bases $e_i^m \otimes e_j^n$. The tensor product of two vector bases $e_i^m \otimes e_j^n$ is equal to the vector basis e_{in+j}^{mn}, i.e., $e_i^m \otimes e_j^n = e_{in+j}^{mn}$. The tensor product of two vector bases $e_i^m \otimes e_j^n$ is called a *tensor basis*. If the basis elements are ordered lexicographically then

$$e_{i_1}^{n_1} \otimes \cdots \otimes e_{i_t}^{n_t} = e_{i_1 n_2 \cdots n_t + \cdots + i_{t-1} n_t + i_t}^{n_1 \cdots n_t}$$

Expressing a vector basis e_i^M as the tensor product of vector bases $e_{i_1}^{m_1} \otimes \cdots \otimes e_{i_t}^{m_t}$, where $M = m_1 \times \cdots \times m_t$ and $i_k = (i \; div \; M_k) \; mod \; m_k$, $M_k = \prod_{i=k+1}^{t} m_i$,

$M_t = 1$ is called the factorization of the vector basis, e.g., the vector basis e_8^{12} can be factorized into the tensor bases $e_1^2 \otimes e_1^3 \otimes e_0^2$ or $e_2^4 \otimes e_2^3$. Expressing a tensor basis $e_{i_1}^{n_1} \otimes \cdots \otimes e_{i_t}^{n_t}$ as a vector basis $e_{i_1 n_2 \cdots n_t + \cdots + i_{t-1} n_t + i_t}^{n_1 \cdots n_t}$ is called *linearization* of the tensor basis. For example, the tensor basis $e_2^4 \otimes e_2^3$ can be linearized to give the vector basis e_8^{12}.

One of the permutations used frequently in the representation of algorithms in tensor product formulas is the *stride permutation*. Stride permutation L_n^{mn} is defined as

$$L_n^{mn}(e_i^m \otimes e_j^n) = e_j^n \otimes e_i^m$$

L_n^{mn} permutes the elements of a vector of size mn with stride distance n. This permutation can be represented as an $mn \times mn$ transformation. For example, L_2^6 can be represented by the matrix

$$L_2^6 x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{bmatrix}$$

The stride permutation has the following properties:

Property 2.2 (Stride Permutation)

1. $(L_n^{mn})^{-1} = L_m^{mn}$
2. $L_{st}^{rst} = L_s^{rst} L_t^{rst}$
3. $L_t^{rst} = (L_t^{rt} \otimes I_s)(I_r \otimes L_t^{st})$

A permutation of the form $I_m \otimes L_q^{pq} \otimes I_n$ is called a *tensor permutation*.

The following theorem illustrates how a tensor product of two matrices can be commuted by applying a stride permutation.

Theorem 2.1 (Commutation Theorem) If A is an $m \times m$ matrix and B is an $n \times n$ matrix, then $L_n^{mn}(A \otimes B) = (B \otimes A)L_n^{mn}$.

Pairwise multiplicaiton between two vectors implies the product between the corresponding elements of those vectors, e.g.,

$$\begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_{n-1} \end{bmatrix} * \begin{bmatrix} y_0 \\ y_1 \\ \cdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 y_0 \\ x_1 y_1 \\ \cdots \\ x_{n-1} y_{n-1} \end{bmatrix}$$

If the elements x_i and y_i are themselves submatrices, then $x_i y_i$ corresponds to matrix multiplication between them.

3 A TENSOR PRODUCT FORMULATION OF STRASSEN'S ALGORITHM

Strassen's matrix multiplication algorithm is based on a computationally efficient way of multiplying 2×2 matrices using only seven multiplications [17]. Consider the matrix multiplication $C = AB$, where

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

Strassen's algorithm can then be written as follows. First, the following intermediate values are calculated.

$$\begin{aligned} t_0 &= (a_{00} + a_{11})(b_{00} + b_{11}) \\ t_1 &= (a_{10} + a_{11})b_{00} \\ t_2 &= a_{00}(b_{01} - b_{11}) \\ t_3 &= a_{11}(-b_{00} + b_{10}) \\ t_4 &= (a_{00} + a_{01})b_{11} \\ t_5 &= (-a_{00} + a_{10})(b_{00} + b_{01}) \\ t_6 &= (a_{01} - a_{11})(b_{10} + b_{11}) \end{aligned}$$

Then the individual elements of C are given by:

$$\begin{aligned} c_{00} &= t_0 + t_3 - t_4 + t^6 \\ c_{10} &= t_1 + t_3 \\ c_{01} &= t_2 + t_4 \\ c_{11} &= t_0 - t_1 + t_2 + t_5 \end{aligned}$$

In matrix notation, this can be represented as:

$$\overline{C} = S_c(S_a \overline{A} * S_b \overline{B})$$

where

$$S_a = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}, S_b = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \text{ and } S_c = \begin{bmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

and $\overline{A}$, $\overline{B}$, and $\overline{C}$ are vectors of length 4, and represent the storage of matrices A, B, and C in column major form. The notation $\overline{A}$ corresponds exactly to the $vec(A)$ notation [8], however, we shall use the former for readability purposes. The matrices S_a, S_b, and S_c are termed *basic operators*, and do not have to be explicitly generated, but specify which operations have to be performed on specific components of the input vectors.

The above formulation can be easily extended to matrices of size $2^n \times 2^n$ by considering a_{ij}, b_{ij}, and c_{ij} to be blocks of size $2^{n-1} \times 2^{n-1}$. First, we describe the block recursive storage of matrices in memory. Let X be any $2^n \times 2^n$ matrix. At the top level, X can be viewed as:

$$X = \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix}$$

A vector $\overline{X}$ representing an r-level block recursive representation of X is recursively defined as:

$$\overline{X} = \begin{bmatrix} \overline{X}_{00} \\ \overline{X}_{10} \\ \overline{X}_{01} \\ \overline{X}_{11} \end{bmatrix}$$

with the boundary condition that $\overline{Y}$ is the column major representation of any $2^{n-r} \times 2^{n-r}$ block Y. An example of block recursive storage is given in Figure 1.

Let $\overline{A}$, $\overline{B}$, and $\overline{C}$ be the one-level block recursive representation of $2^n \times 2^n$ matrices A, B, and C. Strassen's algorithm for computing $C = AB$ can be written as:

$$\overline{C} = (S_c \otimes I_{4^{n-1}})[(S_a \otimes I_{4^{n-1}})\overline{A} *_{n-1} (S_b \otimes I_{4^{n-1}})\overline{B}]$$

where $*_{n-1}$ denotes pairwise matrix multiplication between matrices of size $2^{n-1} \times 2^{n-1}$. We refer to the above as one-level block recursive Strassen's algorithm. In this case, the intermediate values t_i are $2^{n-1} \times 2^{n-1}$ block matrices, and block matrix multiplications are performed using conventional matrix multiplication. This algorithm can be conveniently viewed in terms of a recursion tree (Fig. 2), where the root node corresponds to the update of C, and the leaf nodes correspond to the evaluation of the intermediate values. The steps marked by □ refer to computations that require working memory. Note that all the intermediate values can be computed in parallel, because there are no data dependences between them. Each intermediate value requires a working memory of $O(4^{n-1})$. Hence, a one-level block recursive Strassen's algorithm requires a total working storage of size $O(7 \cdot 4^{n-1})$.

Even though the above formulation has been

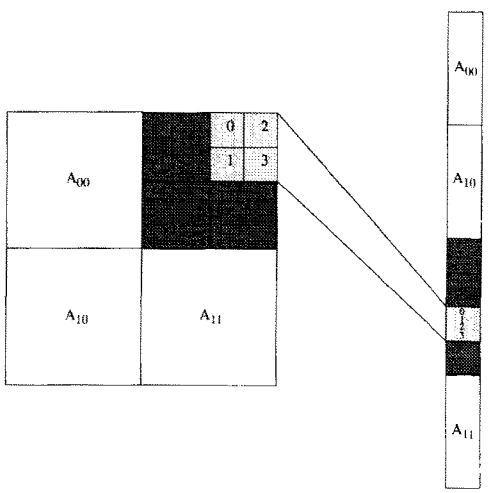

FIGURE 1 Three-level block recursive storage.

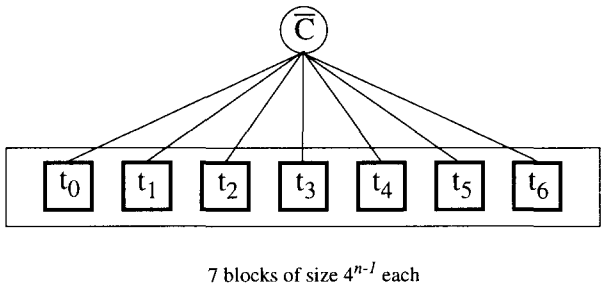

FIGURE 2 Recursion tree of depth 1 for Strassen's algorithm.

given for matrix sizes of the form $2^n \times 2^n$, it is straightforward to generalize the implementation to handle arbitrary dimensions of matrices A and B. A common technique used is to pad the matrices with rows and columns of zeros to increase the matrix sizes to the next higher powers of two, compute the extended matrix product, and then extract the desired result [17]. Another approach [4] is to drop the last rows and columns from the computation to achieve even dimensions and then compute the partial matrix product. The complete matrix product is then obtained with a rank-k update ($k = 1, 2, 3$).

3.1 Block Recursive Strassen's Algorithm: Breadth-First Evaluation

In one-level application of Strassen's algorithm, $2^{n-1} \times 2^{n-1}$ block multiplications were computed using conventional matrix multiplication. To get an additional savings in the total number of arithmetic operations required to compute the matrix product, Strassen's algorithm can be recursively applied to compute the block multiplications also. Let $\overline{A}$, $\overline{B}$, and $\overline{C}$ be the l-level block recursive representations of $2^n \times 2^n$ matrices A, B, and C. The computation of $\overline{C}$ is described by the following formulation [10]:

$$\overline{C} = S_c^{n,l}[S_a^{n,l}(\overline{A}) *_{n-1} S_b^{n,l}(\overline{B})]$$

where

$$S_a^{n,l} = \bigotimes_{i=0}^{l-1} S_a \otimes I_{4^{n-l}} = \prod_{i=0}^{l-1}(I_{7^i} \otimes S_a \otimes I_{4^{n-(i+1)}})$$
$$S_b^{n,l} = \bigotimes_{i=0}^{l-1} S_a \otimes I_{4^{n-l}} = \prod_{i=0}^{l-1}(I_{7^i} \otimes S_b \otimes I_{4^{n-(i+1)}})$$
$$S_c^{n,l} = \bigotimes_{i=0}^{l-1} S_c \otimes I_{4^{n-l}} = \prod_{i=n-1}^{0}(I_{7^i} \otimes S_c \otimes I_{4^{n-(i+1)}})$$

and $*_{n-l}$ denotes pairwise multiplication between blocks of size $2^{n-l} \times 2^{n-l}$. This computation can be interpreted as a breadth-first evaluation of the recursion tree shown in Figure 3. Each intermediate block matrix t_i is itself computed using Strassen's algorithm yielding intermediate subblocks $t_{i0}, \ldots, t_{i6}$. This process is recursively applied until blocks of size $2^{n-l} \times 2^{n-l}$, which are then computed using conventional matrix multiplication. Following our convention, $\square$ denotes computation that requires working storage. The working array requirement in this case is $O(7^l 4^{n-l})$. In the extreme case, Strassen's algorithm can be applied recursively down to blocks of size 2×2, and such an $(n-1)$-level (or n-level) Strassen's algo-

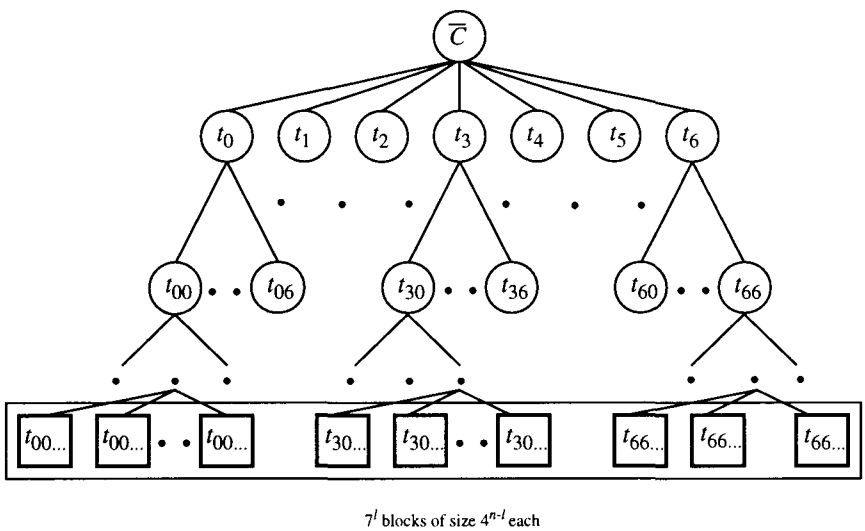

FIGURE 3 l-Level block recursive Strassen's algorithm.

Table 1. Comparison of Operation Counts for Strassen's Algorithm and Conventional Matrix Multiplication

Algorithm	Operation Count		
	Additions	Multiplications	Total
MM	8^n	8^n	$2 \cdot 8^n$
STR	$6(7^n - 4^n)$	7^n	$7^{n+1} - 6 \cdot 4^n$
BLOCK_STR	$6 \cdot 4^k(7^{n-k} - 4^{n-k})$	$7^{n-k}(2 \cdot 8^k - 4^k)$	$7^{n-k}(2 \cdot 8^k + 5 \cdot 4^k) - 6 \cdot 4^n$

rithm would require a working storage of size $O(7^n)$.

Table 1 presents the total number of operations required for multiplying two matrices of size $2^n \times 2^n$. MM denotes conventional matrix multiplication, STR refers to an n-level block recursive Strassen's algorithm, and BLOCK_STR denotes a $(n - k)$-level block recursive Strassen's algorithm. STR has a lower operation count than MM only for $n \geq 10$. The expression for the operation count for BLOCK_STR has a minima at $k = 3$ for all integer values of n and k. For $k = 3$, BLOCK_STR has a lower operation count than MM for $n \geq 4$. Hence, block Strassen's algorithm is better than conventional matrix multiplication in terms of total operation count even for small values of n. However, for implementation on a shared memory vector machine such as the Cray Y-MP, a lower operation count does not imply smaller execution time, because the effect of vector length and stride also comes into play.

3.2 Block Recursive Strassen's Algorithm: Depth-First Evaluation

An l-level Strassen's algorithm requires fewer operations than conventional matrix multiplication when the number of levels l is increased. An optimal value is attained at $n - l = 3$. However, the working storage requirement for an l-level algorithm is $O(7^l 4^{n-l})$, and hence increases exponentially with an increase in l. This high storage requirement comes due to the breadth-first expansion of the recursion tree in which all the intermediate values have to be stored.

To achieve reduction in working storage, we can perform the computation of Strassen's algorithm using a depth-first expansion of the recursion tree. Instead of expanding all the leaves in the recursion tree, we only compute a subtree, and use the results obtained from that subtree to update C. This process is repeatedly applied until all the subtrees are evaluated. It is necessary to ensure that no redundant computation is performed. The memory requirement for the algorithm in this case will be the memory requirement for a single subtree, because the same space can be reused for the evaluation of different subtrees.

For the 2×2 case, the algorithm is modified as follows. t is a temporary variable that is used to store intermediate values.

Step 1: $t = (a_{00} + a_{11})(b_{00} + b_{11})$;
$c_{00} = t$; $c_{11} = t$

Step 2: $t = (a_{10} + a_{11})b_{00}$;
$c_{10} = t$; $c_{11} = c_{11} - t$;

Step 3: $t = a_{00}(b_{01} - b_{11})$
$c_{01} = t$; $c_{11} = c_{11} + t$;

Step 4: $t = a_{11}(-b_{00} + b_{10})$
$c_{00} = c_{00} + t$; $c_{10} = c_{10} + t$;

Step 5: $t = (a_{00} + a_{01})b_{11}$;
$c_{00} = c_{00} - t$; $c_{01} = c_{01} + t$;

Step 6: $t = (-a_{00} + a_{10})(b_{00} + b_{01})$;
$c_{11} = c_{11} + t$;

Step 7: $t = (a_{01} - a_{11})(b_{10} + b_{11})$;
$c_{00} = c_{00} + t$;

Now the extra memory requirement is of only one element, because the same memory location can be reused to evaluate different t_i's. In the original formulation, seven memory locations are required because all the intermediate values are calculated before the update of C is performed. The total number of arithmetic operations is unchanged.

We now formulate the concept of memory reduction using the tensor product framework. Define D_j^7 to be a 7×7 matrix with $d_{jj} = 1$ and zeros elsewhere. Note that $\sum_{j=0}^{6} D_j^7 = I_7$. Memory reduction for a 2×2 case can be formulated in matrix notation as:

$$\overline{C} = \sum_{j=0}^{6} (S_c D_j^7)[(D_j^7 S_a)\overline{A} * (D_j^7 S_b)\overline{B}]$$

D_j^7 is termed as a *selection operator* and selects subsets of the input vector on which the computation is to be performed.

This framework can be extended to multiplying matrices of size $2^n \times 2^n$. We begin with the one-level Strassen's algorithm and assume that the data matrices are stored in a one-level block recursive form. The tensor product formula to compute $C = AB$ can then be written as:

$$\overline{C} = \sum_{j=0}^{6} (S_c D_j^7 \otimes I_{4^{n-1}}) \left[(D_j^7 S_a \otimes I_{4^{n-1}})\overline{A} *_{n-1} (D_j^7 S_b \otimes I_{4^{n-1}})\overline{B} \right]$$

We can apply memory reduction at multiple levels by performing the same operation recursively on the smaller blocks. Assuming that the matrices are stored in an l-level block recursive form, an l-level Strassen's algorithm with memory reduction can be formulated as:

$$\overline{C} = \sum_{j_0, j_1, \cdots, j_{l-1}=0}^{6} \left[\left(\left(\bigotimes_{r=0}^{l-1} S_c D_{j_r}^7 \right) \otimes I_{4^{n-l}} \right) \left(\left(\left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 S_a \right) \otimes I_{4^{n-l}} \right) \overline{A} *_{n-l} \left(\left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 S_b \right) \otimes I_{4^{n-l}} \right) \overline{B} \right) \right]$$

We refer to the above formulation as the *partial evaluation* form of Strassen's algorithm. The computation specified in the above formulation can be described using the recursion tree shown in Figure 4. The current intermediate blocks being computed are represented by □. Working storage is required for the intermediate blocks from the leaf node being computed, to the root of the recursion tree. Hence, the working storage required is $O(\sum_{i=1}^{l} 4^{n-i}) = O(4^n)$.

3.3 Combining Breadth-First and Depth-First Evaluations

The * operator in the tensor product formula for partial computation refers to pairwise matrix multiplication. Each block matrix multiplication in the pairwise matrix multiplication can itself be performed using complete evaluation. Hence, we have a three-level hierarchy. At the highest level, partial evaluation is performed till blocks of size $2^{n-l} \times 2^{n-l}$. Then complete evaluation is performed till blocks of size $2^k \times 2^k$ are reached, after which conventional matrix multiplication is applied. This can be expressed in the tensor product notation as:

$$\overline{C}_{PS} = \sum_{j_0, \cdots, j_{l-1}=0}^{6} \left[\left(\left(\bigotimes_{r=0}^{l-1} S_c D_{j_r}^7 \right) \otimes I_{4^{n-l}} \right) \left(\left(\left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 S_a \right) \otimes I_{4^{n-l}} \right) \overline{A} *_{CS_{n-l}} \left(\left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 S_b \right) \otimes I_{4^{n-l}} \right) \overline{B} \right) \right]$$

$$\overline{C'}_{CS} = \tilde{S}_c^{n-l,k} (\tilde{S}_a^{n-l,k} \overline{A}' *_{MM_k} \tilde{S}_b^{n-l,k} \overline{B}')$$

where $*_{CS_{n-l}}$ denotes pairwise matrix multiplication between blocks of size $2^{n-l} \times 2^{n-l}$ using complete evaluation, $\overline{C'}_{CS}$ corresponds to each block pairwise multiplication during the partial evalua-

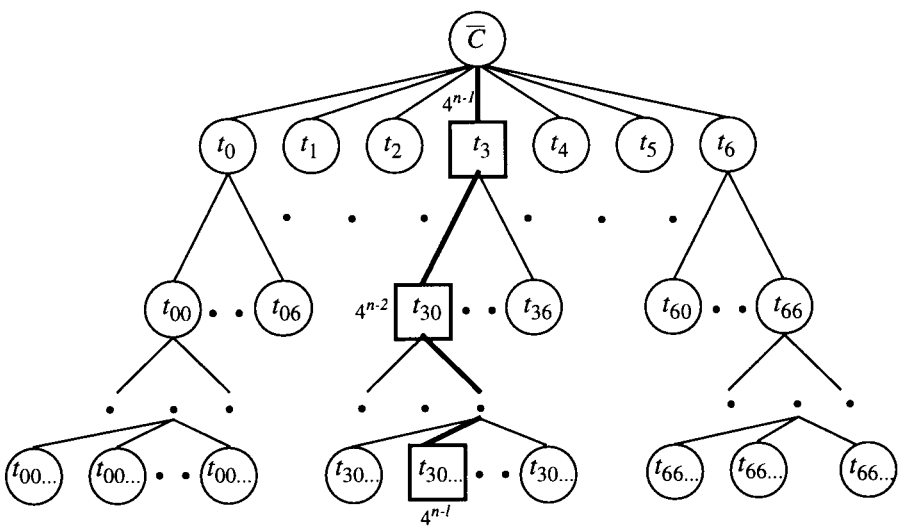

FIGURE 4 l-Level block recursive Strassen's algorithm with memory reduction.

tion, which itself is evaluated using an $(l-k)$-level Strassen's algorithm, and $*_{MM_k}$ denotes pairwise matrix multiplication between blocks of size $2^k \times 2^k$ using conventional matrix multiplication.

The root of the recursion tree is defined to be at level 0. At level i, a working array of size $O(4^{n-i})$ is required to store the intermediate results of partial evaluation. The breadth-first expansion of the last $(l-k)$ levels requires a working array of $O(7^{n-l-k} \cdot 4^k)$. Hence, the total memory requirement is $O(\sum_{i=1}^{l-1} 4^{n-i} + 7^{n-l-k} \cdot 4^k) = o(4^n + 7^{n-l-k} \cdot 4^k)$. Even for moderate values of n and small values of l, this represents a significant savings compared with $O(7^{n-k} \cdot 4^k)$ for complete evaluation. If the matrices are of size $N \times N$ where N is not a power of 2, the technique of padding can be used, and the memory requirement with reduction will be $o(4^{\lceil \lg N \rceil} + 7^{\lceil \lg N \rceil - l - k} \cdot 4^k)$ compared with $O(7^{\lceil \lg N \rceil - k} \cdot 4^k)$ for complete evaluation.

3.4 Matrix Storage in Main Memory

The formulation presented in the previous sections assumes for simplicity of presentation that the data matrices are stored in a block recursive form. However, when implementing a block recursive algorithm on a shared memory machine, matrices are usually stored in a row major or column major form. We have implemented Strassen's algorithm using Fortran on the Cray Y-MP, hence the data matrices are stored in memory in column major form. The tensor product formula to convert a $2^n \times 2^n$ matrix from a column major form to a k-level block recursive form is given by [11]:

$$R^{n,k} = \left[\prod_{i=0}^{n-k-1} (I_{2^{2i+1}} \otimes L_2^{2^{n-k-i}} \otimes I_{2^{n+k-i-1}}) \right] (I_{2^{n-k}} \otimes L_{2^{n-k}}^{2^n} \otimes I_{2^k})$$

$R^{n,k}$ is termed as a *conversion operator*. There are two ways in which storage conversion can be implemented. One way is to perform explicit conversion from row/column major form to a block recursive form through data movement. However, a more efficient way is to merge the conversion operator into the computation in Strassen's algorithm, which results in a modification of the data array indexing functions. The modified tensor formulation for Block Strassen's algorithm is:

$$\begin{aligned} \overline{C} &= (R^{n,k})^{-1} S_c^{n,k} [R^{n,k} S_a^{n,k} \overline{A} *_k R^{n,k} S_b^{n,k} \overline{B}] \\ &= \tilde{S}_c^{n,k} [\tilde{S}_a^{n,k} \overline{A} *_k \tilde{S}_b^{n,k} \overline{B}] \end{aligned}$$

where

$$\tilde{S}_a^{n,k} = \prod_{i=0}^{n-k-1} [(I_{7^i} \otimes S_a \otimes I_{4^{n-i-1}})(I_{7^i} \otimes I_2 \otimes L_2^{2^{n-k-i}} \otimes I_{2^{n+k-i-1}})](I_{2^{n-k}} \otimes L_{2^{n-k}}^{2^n} \otimes I_{2^k})$$

$$\tilde{S}_b^{n,k} = \prod_{i=0}^{n-k-1} [(I_{7^i} \otimes S_b \otimes I_{4^{n-i-1}})(I_{7^i} \otimes I_2 \otimes L_2^{2^{n-k-i}} \otimes I_{2^{n+k-i-1}})](I_{2^{n-k}} \otimes L_{2^{n-k}}^{2^n} \otimes I_{2^k})$$

$$\tilde{S}_c^{n,k} = (I_{2^{n-k}} \otimes L_{2^k}^{2^n} \otimes I_{2^k}) \prod_{i=n-k-1}^{0} [(I_{7^i} \otimes I_2 \otimes L_{2^{n-k-i-1}}^{2^{n-k-i}} \otimes I_{2^{n+k-i-1}})(I_{7^i} \otimes S_c \otimes I_{4^{n-i-1}})]$$

With l-level memory reduction, the above formulation is modified into:

$$\overline{C} = \sum_{j_0, j_1, \cdots, j_{l-1}=0}^{6} \left[\left(\tilde{S}_c^{n,k} \otimes \left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 \right) \right) \left(\left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 \otimes \tilde{S}_a^{n,k} \overline{A} \right) *_k \left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 \otimes \tilde{S}_b^{n,k} \overline{B} \right) \right) \right]$$

4 CODE GENERATION FOR VECTOR PROCESSORS

4.1 Block Strassen's Algorithm

Matrix factorizations form the basis of translating tensor product formulas by mapping the operations implied by the formula to program constructs in a high-level programming language. The translation process starts with the top-level abstraction and generates more refined code as it proceeds to lower-level abstractions. At each level, semantically equivalent program constructs are chosen to replace mathematical operations. Efficient programs can be synthesized from tensor product formulas by exploiting the regular computational structure expressed by such formulas. The tensor product formulation of block recursive algorithms usually involves certain basic computations, such as S_a, S_b, and S_c in the case of Strassen's algorithm. It is sometimes necessary to use manually optimized codes for these basic computations to achieve high performance.

We now illustrate the code generation strategy with an example. Let B be an $m \times n$ matrix, and X be a vector of size np. Consider the application of $(I_p \otimes B)$ to X, i.e.,

$$\begin{bmatrix} B & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B \end{bmatrix} \begin{bmatrix} X[0 : n-1] \\ X[n : 2n-1] \\ \vdots \\ X[(p-1)n : pn-1] \end{bmatrix}$$

$$= \begin{bmatrix} BX[0 : n-1] \\ BX[n : 2n-1] \\ \vdots \\ BX[(p-1)n : pn-1] \end{bmatrix}$$

This can be interpreted as p copies of B acting in parallel on p disjoint segments of X, resulting in a vector of size mp. Hence, $Y = (I_p \otimes B)X$ can be implemented as:

$$\begin{aligned} Code[Y = (I_p \otimes B)X] \equiv\ & \textbf{doall } i = 0, p-1 \\ & Code[Y[in : (i+1)n - 1] \\ & = BX[in : (i+1)n - 1]] \\ & \textbf{enddoall} \end{aligned}$$

Once an algorithm is expressed using the tensor product framework, efficient implementation can be obtained by algebrically manipulating the tensor product formula. For example, consider the implementation of

$$Y = (B \otimes I_p)X$$

where Y, B, and X are vectors as described before. Using the commutation rule, it can be determined that

$$(B \otimes I_p) = L_m^{mp}(I_p \otimes B)L_p^{np}$$

Hence, one implementation to compute Y might be to permute X according to L_p^{np}, perform $(I_p \otimes B)$, and permute the result according to L_m^{mp}. A more efficient implementation would be to incorporate the stride permutations into the indexing of the input and output data arrays. The above can be written as:

$$(L_p^{mp}Y) = (I_p \otimes B)(L_p^{np}X)$$

i.e.,

$$\begin{bmatrix} Y[0 : mp-1 : p] \\ Y[1 : mp-1 : p] \\ \vdots \\ Y[p-1 : mp-1 : p] \end{bmatrix}$$

$$= \begin{bmatrix} B & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B \end{bmatrix} \begin{bmatrix} X[0 : np-1 : p] \\ X[1 : np-1 : p] \\ \vdots \\ X[p-1 : np-1 : p] \end{bmatrix}$$

Hence, the code can be written as

$$\begin{aligned} Code[Y = (B \otimes I_p)X] \equiv\ & \textbf{doall } i = 0, p-1 \\ & Code[Y[in : (i+1)n - 1 : p] \\ & = BX[in : (i+1)n - 1 : p]] \\ & \textbf{enddoall} \end{aligned}$$

Let us consider the code generation for $(n-k)$-level block Strassen's algorithm for multiplying $2^n \times 2^n$ matrices. Assume that the matrices are stored in a $(n-k)$-level block recursive format, and that at the lowest level, pairwise multiplication between blocks of size $2^k \times 2^k$ is performed. For simplification, we shall assume that no memory reduction is performed. The tensor product formulation of this algorithm is given by (see Section 3.4):

$$\overline{C} = \tilde{S}_c^{n,k}[\tilde{S}_a^{n,k}\overline{A} *_k \tilde{S}_b^{n,k}\overline{B}]$$

The formula for block Strassen's algorithm contains the operations $\tilde{S}_a^{n,k}$, $\tilde{S}_b^{n,k}$, $*_k$, and $\tilde{S}_c^{n,k}$. All the operations except $*_k$ are linear operations and hence require an array operand. Operation $*_k$ is a bilinear operation and requires two array operands. Each operation corresponds to an assignment statement that stores its result in an array that may be used as input data for the subsequent assignment or represents the final output. Temporary arrays representing working arrays are denoted by T_i. The above formula then translates to the following high-level code:

$$T_0 = \tilde{S}_a^{n,k}\overline{A}$$
$$T_1 = \tilde{S}_b^{n,k}\overline{B}$$
$$T_0 = T_0 *_k T_1$$
$$\overline{C} = \tilde{S}_c^{n,k}T_0$$

The assignment statements are composed sequentially to preserve the semantics of computation. However, the above sequential composition is not unique. For example, the assignment statements for $\tilde{S}_a^{n,k}\overline{A}$ and $\tilde{S}_b^{n,k}\overline{B}$ can be in any order because there are no data dependences between them.

$\tilde{S}_a^{n,k}$, $\tilde{S}_b^{n,k}$, and $\tilde{S}_c^{n,k}$ have the form

$$\left[\prod_{i=1}^{n} F_i\right] \text{ where } F_i = \begin{cases} (I_{r_i} \otimes OP \otimes I_{s_i}) \\ (I_{r_i} \otimes L^{m_i n_i} \otimes I_{s_i}) \end{cases} \text{ and OP is a basic operator}$$

The generic tensor product formula $Y = (I_{r_i} \otimes OP \otimes I_{s_i})X$ can be implemented as a fully parallel doubly nested loop:

doall $i_1 = 0, r_i - 1$
  doall $i_2 = 0, s_i - 1$
    $Code[OP,Y,X,i_1,i_2]$
  enddoall
enddoall

Any tensor permutation that may be present results in a modification of the array indexing functions. Different implementations of the above formula are possible by changing the order and/or blocking the inner loops, as they are fully permutable. However, different orderings of the inner loops result in different data access patterns. These in turn will have different performance charateristics on systems with hierarchical/interleaved memories.

Consider the application of the tensor product formula $\prod_{i=1}^{n} F_i$ to a vector X. The product term corresponds to a sequential outer loop in which the output of the i^{th} stage is fed as input to the $(i + 1)^{th}$ stage, $i = 1, n - 1$. Only two arrays are required for this operation. The input array for the i^{th} step can be reused as the output array for the $(i + 1)^{st}$ step. At the end of each iteration, the arrays are swapped (which can be implemented trivially simply by swapping the pointers to the two arrays) and the resulting pseudocode is:

$T_0 \leftarrow X$
do $i = 1, n$
  $Code[T_1 = F_i T_0]$
  $Swap(T_1, T_0)$
enddo

At the end of the last iteration, T_1 contains the result of $[\prod_{i=1}^{n} F_i]X$. $\tilde{S}_a^{n,k}$, $\tilde{S}_b^{n,k}$, and $\tilde{S}_c^{n,k}$ have the above form, and code can be easily generated for them.

The pairwise multiplication $*_k$ performs a sequence of 7^{n-k} matrix multiplications of $2^k \times 2^k$ blocks. Let the input vectors be T_0 and T_1, corresponding to the evaluation of $\tilde{S}_a^{n,k}\overline{A}$ and $\tilde{S}_a^{n,k}\overline{B}$, respectively. All elements of a given block are stored consecutively in the input arrays. Pseudocode for the operation $T_2 = T_0 *_k T_1$ is presented below:

doall $i = 0, 7^{n-k} - 1$
  $T_2[i4^k : (i + 1)4^k - 1] = MatrixMultiply$
  $(T_0[i4^k : (i + 1)4^k - 1], T_1[i4^k : (i + 1)4^k - 1])$
enddoall

where *MatrixMultiply* refers to conventional matrix multiplication between blocks of size $2^k \times 2^k$ stored in column major form.

4.2 Memory Management for Depth-First Evaluation

Consider the tensor product formula:

$$\overline{C} = \sum_{j_0,j_1,\ldots,j_{l-1}=0}^{6} \left[\left(\left(\bigotimes_{r=0}^{l-1} S_c D_{j_r}^7\right) \otimes I_{4^{n-l}}\right) \left(\left(\left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 S_a\right) \otimes I_{4^{n-l}}\right)\overline{A} *_{n-l} \left(\left(\bigotimes_{r=0}^{l-1} D_{j_r}^7 S_b\right) \otimes I_{4^{n-l}}\right)\overline{B}\right)\right]$$

The summation operator in the formulation of partial evaluation corresponds to a sequential loop nest, with the i^{th} loop performing a depth-first evaluation of the i^{th} level in the recursion tree. At each level, there are seven subtrees that need to be evaluated. Evaluation of each subtree is followed by an update of its parent. After the update, working storage used by that subtree can be reused for the computation of the next subtree at that level in the recursion tree. The loop structure hence looks like the following:

```
do j0 = 0, 6
    Code[T_a^0 = D_{j0}^7 S_a Ā]        /* partial evaluation */
    Code[T_b^0 = D_{j0}^7 S_b B̄]        /* partial evaluation */
    do j1 = 0, 6
        Code[T_a^1 = D_{j1}^7 S_a Ā]        /* partial evaluation */
        Code[T_b^1 = D_{j1}^7 S_b B̄]        /* partial evaluation */
        ...
        do j_{l-1} = 0, 6
            Code[T_a^{l-1} = D_{j_{l-1}}^7 S_a Ā]                  /* partial evaluation */
            Code[T_b^{l-1} = D_{j_{l-1}}^7 S_a B̄]                  /* partial evaluation */
            Code[T_c^{l-1} = T_a^{l-1} *_{CS_{n-l}} T_b^{l-1}]     /* complete evaluation */
            Code[T_c^{l-2} = D_{j_{l-1}}^7 S_c T_c^{l-1}]          /* update of parent */
        enddo
        ...
        Code[T_c^0 = D_{j1}^7 S_c T_c^1]     /* update of parent */
    enddo
    Code[C̄ = D_{j0}^7 S_c T_c^0]        /* update of parent */
enddo
```

4.3 Implementation of Winograd's Variation

Strassen's algorithm uses 18 scalar additions and 7 scalar multiplications to multiply 2×2 matrices. Winograd presented a more efficient algorithm, which uses 15 scalar additions and 7 scalar multiplications [3]. The Winograd's variation is based on the following three matrix operations:

$$W_a = \begin{bmatrix} -1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$W_b = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & -1 & -1 & 1 \end{bmatrix},$$

and $$W_c = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}.$$

The Winograd's variation for multiplying 2×2 can be written as the matrix formula

$$\begin{bmatrix} c_{0,0} \\ c_{1,0} \\ c_{0,1} \\ c_{1,1} \end{bmatrix} = W_c \left(W_a \begin{bmatrix} a_{0,0} \\ a_{1,0} \\ a_{0,1} \\ a_{1,1} \end{bmatrix} * W_b \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{0,1} \\ b_{1,1} \end{bmatrix} \right).$$

The generated code of operations W_a, W_b, and W_c contains some common terms. For example, $a_{0,0} - a_{1,0}$ is evaluated twice in a direct implementation of W_a. The key to reducing the number of additions in Winograd's variation is to evaluate a common term only once. We factorize W_a, W_b, and W_c to eliminate the common terms:

$$W_a = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$W_b = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \text{ and}$$

$$W_c = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

There are 15 rows containing two nonzero elements in the matrix factorizations of W_a, W_b, and W_c, which correspond to the 15 additions in Winograd's variation of Strassen's algorithm. The rows containing a single one are implemented as data movement, and those containing all zeros are equivalent to null operations. The indices of input and output array elements of W_a are specified by the permutation operations in a tensor product formula and are computed similar to those of S_a. Let p_i, $0 \leq i < 4$, be the indices of the input array elements, and q_i, $0 \leq i < 7$, be the indices of the output array elements. The computation $T = W_a A$ on a vector A of length 7 is translated to the following sequence of assignments:

$$\begin{aligned} Code[T = W_a A] \equiv\ & T[q_1] = A[p_0] \\ & T[q_2] = A[p_2] \\ & T[q_3] = A[p_0] - A[p_1] \\ & T[q_4] = A[p_1] + A[p_3] \\ & T[q_6] = A[p_3] \\ & T[q_0] = -T[q_1] + T[q_4] \\ & T[q_5] = -T[q_0] + T[q_2] \end{aligned}$$

The implementation of Winograd's variation is simply a replacement of the translated code of W_a, W_b, and W_c for the code of S_a, S_b, and S_c in the corresponding implementation of Strassen's algorithm.

Table 2. Execution Times for Block Strassen's Algorithm with Memory Reduction

	$n = 8$					$n = 9$					$n = 10$				
	SGEMM: .109 s (310 MFlop) *SGEMMS*: .093 s (291 MFlop)					*SGEMM*: .868 s (310 MFlop) *SGEMMS*: .665 s (285 MFlop)					*SGEMM*: 6.95 s (309 MFlop) *SGEMMS*: 4.69 s (284 MFlop)				
l	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
0	.468	.179	.103	.093	.098				.666	.697					
	(55)	(136)	(246)	(291)	(305)				(287)	(303)					
1	.474	.182	.106	.095	.103	3.292	1.268	.736	.672	.712					
	(55)	(133)	(239)	(285)	(293)	(56)	(135)	(135)	(284)	(297)					
2	.476	.186	.108	.096		3.308	1.284	.746	.674	.710		8.92	5.19	4.72	4.98
	(54)	(130)	(236)	(282)		(55)	(133)	(238)	(283)	(297)		(135)	(241)	(284)	(298)
3	.494	.200	.114			3.348	1.315	.767	.686		23.3	9.13	5.29	4.78	5.03
	(52)	(121)	(221)			(54)	(130)	(232)	(278)		(55)	(132)	(236)	(281)	(295)
4	.548	.228				3.475	1.412	.815			23.5	9.38	5.42	4.86	
	(47)	(106)				(52)	(121)	(218)			(54)	(128)	(231)	(276)	
5	.671					3.857	1.619				24.4	10.02	5.76		
	(38)					(47)	(105)				(52)	(120)	(217)		
6						4.665					27.1	11.42			
						(39)					(47)	(105)			
7											32.8				
											(39)				

5 PERFORMANCE RESULTS ON THE CRAY Y-MP

Performance statistics were gathered for different matrix sizes, different block sizes, and different levels of partial evaluation. Table 2 shows performance for execution on a single processor. All execution times are in seconds. The numbers in parentheses display performance in megaflops. Empty fields indicate that the program could not be run due to lack of sufficient memory. The matrix size is $2^n \times 2^n$, the level of partial evaluation is l, and the block size at which conventional matrix multiplication is applied is $2^k \times 2^k$. The execution times for the Block Strassen's algorithm is compared with the Cray Scientific Library routines *SGEMM*, which implements conventional matrix multiplication, and *SGEMMS*, which implements Strassen's matrix multiplication. Because *SGEMM* and *SGEMMS* are independent of l and k, the times for those are given only once for each value of n. *SGEMM* is used for block matrix multiplication in the Block Strassen's algorithm.

For any value of l, the lowest execution time occurs for $k = 6$ because the vector length on the Cray Y-MP is 64. The megaflops for $k = 7$ are higher than those for $k = 6$ for the same value on n and l. But, the execution time for $k = 7$ is longer because a larger number of arithmetic operations are performed.

The execution times and megaflops for $k = 6$, $l = 0$ are comparable (slightly better) to that of SGEMMS. There is a performance degradation due to a slight increase in the number of memory

Table 3. Execution Times for $k = 6$ on Two Processors

		Block Strassen				
n	*SGEMMS*	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 4$
8	.050	.047	.053	.055		
	(594)	(574)	(513)	(497)		
9	.356	.331	.371	.378	.389	
	(592)	(576)	(513)	(505)	(490)	
10	2.51			2.63	2.67	2.76
	(589)			(510)	(502)	(486)

Table 4. Execution Times for $k = 6$ on Eight Processors*

		Block Strassen				
n	*SGEMMS*	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 4$
8	.016 (84.7%)	.022 (54.2%)	.018 (81.8%)	.022 (74.6%)		
9	.10 (91.3%)	.11 (77.3%)	.14 (73.2%)	.13 (80.9%)	.15 (76.0%)	
10	.84 (78.4%)			1.04 (70.9%)	1.02 (74.0%)	1.14 (70.5%)

* Percentages of 8-cpu obtained are given in parentheses.

operations as l increases for any fixed n and $k = 6$. However, the difference is quite small, which is evident from the execution times.

Table 3 gives the performance when the program was run on two processors. A fixed value of $k = 6$ was chosen because this resulted in the best performance in the single processor case. Again, the performance when $l = 0$ is slightly better than that of *SGEMMS*. For larger values of l, the performance degrades by about 12%. Table 4 shows the performance results for eight processors. Because the programs were run in a nondedicated mode on the Cray Y-MP, we were unable to get all the eight processors for the entire execution of the program. The numbers in parentheses give the percentage of 8-cups available for execution.

The amount of extra memory required has been given in Figure 5 for different values of n and l. It can be easily seen that there is an order of magnitude improvement even for small values of l. A value of $k = 6$ was chosen because it is for this block size that the execution times are minimum.

6 CONCLUSIONS

We have shown how tensor product formulas expressing Strassen's matrix multiplication algorithm can be translated to efficient parallel programs for shared memory multiprocessors. This translation process is part of a more general programming methodology for designing high-performance block recursive algorithms for shared and distributed memory machines. The methodology uses a mathematical notation based on tensor products for expressing block recursive algorithms. Algebraic manipulation of these formulas yields mathematically equivalent formulas that result in implementations with different performance characteristics. A large nubmer of programs can be generated to search for efficient implementations. Tensor products give a powerful method to generate these equivalent implementations automatically. As was illustrated in this article, programs generated from tensor product formulas compare favorably with the best hand-coded ones.

This article presents an implementation of the Strassen's algorithm on a shared memory multiprocessor such as the Cray Y-MP. In the Y-MP, memory is organized into banks, and in the absence of bank conflicts, all memory accesses take the same amount of time. However, in distributed memory multiprocessors such as the Cray T3D, where each processor has its own local memory, a local memory access can be significantly faster than a remote access. Hence, an efficient implementation on a distributed memory machine requires partitioning the algorithm in such a manner that remote accesses are minimized.

Tensor product formulas can also be used to specify regular data distributions for arrays. Given a tensor product formula with a specified distribution of its input and output arrays, the interprocessor communication cost incurred by the implementation can be determined. If the cost of communication is high, it might be more efficient

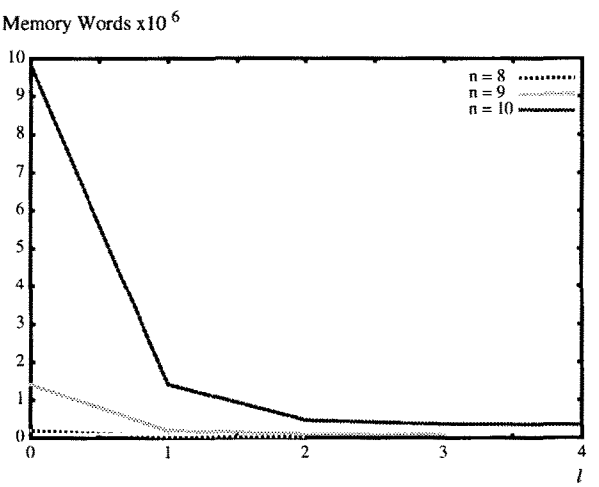

FIGURE 5 Memory requirements for working arrays.

to perform a data redistribution before the computation, to bring the arrays into a form where the computation is local to the processors, if the overhead of data distribution is lower than the benefit gained due to the communication cost reducing to zero. We are currently examining these issues and are working on an implementation on the Cray T3D.

Both formula modification and program generation are capable of being automated. We are currently implementing this methodology in an expert system EXTENT (Expert System for Tensor Formula Translation) that assists in the development of parallel programs for numerical algorithms on various computer architectures. Currently, the system generates Fortran programs for the Cray Y-MP. The expert system employs various heuristics to automatically generate alternative tensor product formulas, translate tensor product formulas to programs for various parallel architectures, test the produced programs, and analyze the test results.

ACKNOWLEDGMENTS

This work was supported in part by ARPA and monitored by NIST.

REFERENCES

[1] D. H. Bailey, "Extra high speed matrix multiplication on the Cray-2," *SIAM J. Sci. Stat. Comput.*, vol. 9, pp. 603–607, 1988.

[2] D. H. Bailey, K. Lee, and H. D. Simon, "Using Strassen's algorithm to accelerate the solution of linear systems," *J. Supercomput., vol. 4, pp. 357–371, Jan. 1991.*

[3] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems.* New York: American Elsevier Publishing Co., 1975.

[4] R. P. Brent, "Algorithms for matrix multiplication," Computer Science Department Stanford University, Palo Alto, CA, Tech. Rep. CS 157, 1970.

[5] J. W. Brewer, "Kronecker products and matrix calculus in system theory," *IEEE Trans. Circuits Systems*, vol. 25, pp. 772–781, 1978.

[6] J. Granta, M. Conner, and R. Tolimieri, "Recursive fast algorithms and the role of tensor products," *IEEE Trans. Signal Processing*, vol. 40, pp. 2921–2930, Dec. 1992.

[7] F. A. Graybill, *Matrices, with Applications in Statistics.* Belmont, CA: Wadsworth International Group, 1983.

[8] H. V. Henderson and S. R. Searle, "The vec-permutation matrix, the vec operator and kronecker products: A review," *Linear Multilinear Algebra*, vol. 9, pp. 271–288, 1981.

[9] N. J. Higham, "Exploiting fast matrix multiplication within the level 3 BLAS," *ACM Trans. Mathematical Software*, vol. 16, pp. 352–368, Dec. 1990.

[10] C.-H. Huang, J. R. Johnson, and R. W. Johnson, "A tensor product formulation of Strassen's matrix multiplication algorithm," *App. Math. Lett.*, vol. 3, pp. 67–71, 1990.

[11] C.-H. Huang, J. R. Johnson, and R. W. Johnson, "Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm, in *International Conference on Parallel Processing*, vol. 3, 1992, p. 104.

[12] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying and implementing fourier transform algorithms on various architectures," *Circuits Systems Signal Process*, vol. 9, pp. 45–500, 1990.

[13] B. Kumar, C. H. Huang, J. Johnson, R. W. Johnson, and P. Sadayappan, "A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction," in *Seventh International Parallel Processing Symposium*, 1993, p. 582.

[14] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform.* New York: SIAM, 1992.

[15] P. A. Regalia and S. K. Mitra, "Kronecker products, unitary matrices and signal processing applications," SIAM Rev. vol. 31, pp. 586–613, Dec. 1989.

[16] G. X. Ritter and P. D. Gader, "Image algebra techniques and parallel image processing," *J. Parallel Distrib. Comput.* vol. 4, pp. 7–44, 1987.

[17] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, pp. 354–356, 1969.