Efficient Representation Scheme for Multidimensional Array Operations

Chun-Yuan Lin, Jen-Shiuh Liu, and Yeh-Ching Chung, *Member*, *IEEE Computer Society*

Abstract—Array operations are used in a large number of important scientific codes, such as molecular dynamics, finite element methods, climate modeling, etc. To implement these array operations efficiently, many methods have been proposed in the literature. However, the majority of these methods are focused on the two-dimensional arrays. When extended to higher dimensional arrays, these methods usually do not perform well. Hence, designing efficient algorithms for multidimensional array operations becomes an important issue. In this paper, we propose a new scheme, *extended Karnaugh map representation* (*EKMR*), for the multidimensional array representation. The main idea of the *EKMR* scheme is to represent a multidimensional array by a set of two-dimensional arrays. Hence, efficient algorithm design for multidimensional array operations becomes less complicated. To evaluate the proposed scheme, we design efficient algorithms for multidimensional array operations, matrix-matrix addition/subtraction and matrix-matrix multiplications, based on the *EKMR* and the *traditional matrix representation* (*TMR*) schemes. Both theoretical analysis and experimental test for these array operations were conducted. Since Fortran 90 provides a rich set of intrinsic functions for multidimensional array operations, in the experimental test, we also compare the performance of intrinsic functions provided by the Fortran 90 compiler and those based on the *EKMR* scheme. The experimental results show that the algorithms based on the *EKMR* scheme outperform those based on the *TMR* scheme and those provided by the Fortran 90 compiler.

Index Terms—Array operations, multidimensional arrays, data structure, extended Karnaugh map representation, traditional matrix representation.

1 Introduction

Array operations are used in a large number of important scientific codes, such as molecular dynamics [10], finite-element methods [16], climate modeling [33], etc. To implement these array operations efficiently, many methods have been proposed in the literature. For example, for two-dimensional arrays, by applying the loop repermutation [4], [28] to reorder the memory accesses for array elements of certain operations, we can obtain better performance. However, the majority of these methods are focused on the two-dimensional arrays. When extended to higher dimensional arrays, these methods usually do not perform well. The reason is that one usually uses the *traditional matrix representation* (*TMR*) that is also known as *canonical data layouts* [8] to represent higher dimensional arrays. In the *TMR* scheme, a three-dimensional array of size $5 \times 4 \times 3$ can be viewed as five 4×3 two-dimensional arrays. This scheme has two drawbacks for higher dimensional array operations. First, the costs of index computations of array elements for array operations increase as the dimension increases. Second, the cache miss rate for array operations increases as the dimension increases due to more cache lines accessed. Hence, multidimensional arrays represented by the *TMR* scheme become less manageable and difficult for programmers to design efficient algorithms.

In this paper, we propose a new scheme called *extended Karnaugh map representation* (*EKMR*) for the multidimensional array representation. The main idea of the *EKMR* scheme is to represent a multidimensional array by a set of two-dimensional arrays. This scheme is suitable for the multidimensional dense or sparse array. Hence, efficient algorithm design for multidimensional arrays based on the *EKMR* scheme becomes less complicated. To evaluate the proposed scheme, we design efficient algorithms for multidimensional array operations, matrix-matrix addition/subtraction and matrix-matrix multiplications, based on the *EKMR* and the *TMR* schemes. Both theoretical analysis and experimental testing for these array operations were conducted. From the theoretical analysis and experimental results, we can see that array operations based on the *EKMR* scheme outperform those based on the *TMR* scheme. The reasons are two-fold. First, the *EKMR* scheme can decrease the costs of index computations of array elements for array operations because it uses a set of two-dimensional arrays to represent a higher dimensional array. Second, the cache miss rate for array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme because the number of cache lines accessed by array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme. Since Fortran 90 provides a rich set of intrinsic functions for multidimensional array operations in the experimental test, we also compare the performance of intrinsic functions provided by the Fortran 90 compiler and those based on the *EKMR* scheme. The experimental results show that algorithms based on the *EKMR* scheme outperform those provided by the Fortran 90 compiler. We also present a transformation scheme called *matrix transformation method* (*MTM*) for conversion between the *TMR* and the *EKMR* schemes.

• *The authors are with the Department of Information Engineering, Feng Chia University, 100 Wen Hwa Road, Taichung, Taiwan 407, Republic of China. E-mail: {cylin, liuj, ychung}@iecs.fcu.edu.tw.*

Manuscript received 25 Sept. 2000; revised 23 May 2001; accepted 26 July 20001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112905.

This paper is organized as follows: In Section 2, a brief survey of related work will be presented. Section 3 will describe the *EKMR* scheme and the *MTM* for multidimensional arrays. Section 4 will present efficient algorithms for multidimensional array operations based on the *EKMR* scheme. We also analyze the costs for these algorithms based on the *TMR* and the *EKMR* schemes in this section. The experimental results will be given in Section 5.

2 Related Work

Many methods for improving array computation have been proposed in the literature. Carr et al. [4], [28] presented a comprehensive approach to improving data locality by using loop transformations, such as loop permutation, loop reversal, etc. They demonstrated that these transformations are useful for optimizing many array programs. They also proposed an algorithm called *LoopCost* to analyze and construct the cost models for variable loop orders of array operations. The cost model computes both temporal and spatial reuse of cache lines in order to select the best loop orders of array operations for data locality.

Kandemir et al. [17], [18] proposed a compiler technique to perform loop and data layout transformations to solve the global optimization problem on sequential and multiprocessor machines. The scope of their work focuses on dense array programs. They use loop transformations to find the best loop order of an array operation. They also use a data layout transformation scheme to change the data layout of an array, such as from the row-major data layout to the column-major data layout, and improve the performance of array operations. However, it is difficult to change the data layout of an array in programming languages, such as C and Fortran. Therefore, their work focused on finding the best loop order of an array operation to solve the global optimization problems. O'Boyle and Knijnenburg [29] presented a new algebraic framework to combine loop and data layout transformations. By integrating loop and data layout transformations, any poor spatial locality and expensive array subscripts can be eliminated. Sularycke and Ghose [31] proposed a simple sequential loop interchange algorithm that can produce a better performance than existing algorithms for array multiplication.

Chatterjee et al. [7] examined two nonlinear data layout functions (4*D* and *Morton*) for two-dimensional arrays with the tiling scheme that promises improved performance at low cost. They focus on dense matrix codes for which loop tiling is an appropriate means of high-level control flow restructuring to improve locality. In [6], they further examined the combination of five recursive data layout functions (various forms of *Morton* and *Hilbert*) with the tiling scheme for three parallel matrix multiplication algorithms. They indicate that these data layout functions with the tiling scheme for two-dimensional dense arrays can be extended to those for multidimensional dense arrays. They also indicate that the performance of array operations for multidimensional dense arrays based on these data layout functions with the tiling scheme is efficient.

Coleman and McKinley [9] presented a new algorithm *TSS* for choosing problem-size dependent tile size based on the cache size and cache line size for a direct-mapped cache. The *TSS* algorithm can eliminate both capacity and self-interference misses and reduces cross-interference misses. By integrating the *TSS* algorithm into array programs, we can improve the cache utilization and the performance of array operations. Wolf and Lam [34] proposed an algorithm that improves the locality of a loop nest by transforming the code via interchange, reversal, skewing, and tiling. In [24], they also presented a comprehensive analysis of the performance of blocked code on machines with caches. They have developed a model for understanding the cache behavior of blocked code. Through the model, they demonstrate that this cache behavior is highly dependent on the way in which a matrix interferes with itself in the cache, which in turn depends heavily on the stride of the accesses. Frens and Wise [15] presented a simple recursive algorithm with the quad-tree decomposition of matrices that has outperformed hand-optimized BLAS3 matrix multiplication. The use of quad-trees or oct-trees is known in parallel computing [2] for improving both load balance and locality. Carter et al. [5] focused on using hierarchical tiling to exploit superscalar-pipelined processor. The hierarchical tiling is a framework for applying known tiling methods to ease the burden on several compiler phases that are traditionally treated separately, such as scalar replacement, register allocation, generation of message passing calls, and storage mapping.

Callahan et al. [3] presented a source-to-source transformation scheme called *scalar replacement*. This scheme finds opportunities for reuse of subscripted variables and replaces the references involved by references to temporary scalar variables. In addition, they use transformations to improve the overall effectiveness of scalar replacement and apply these transformations in a variety of loop nest types.

Kotlyar et al. [20], [21], [22] presented a relational algebra-based framework for compiling efficient sparse array code from dense DO-Any loops and a specified sparse array. Fraguela et al. [11], [12], [13], [14] analyzed the cache effects for the array operations. They established the cache probabilistic model and modeled the cache behavior for sparse array operations. Kebler and Smith [19] described a system, SPARAMAT, for concept comprehension that is particularly suitable for sparse array codes. Their automatic program comprehension techniques for sparse array codes can be used in a sequential or a parallel environment. Ziantz et al. [35] proposed a runtime optimization technique that can be applied to a compressed row storage array for array distribution and off-processor data fetching in order to reduce both the communication and computation time.

3 The $EKMR$ and MTM Schemes

In the following, we use *TMR*(*n*) for the *TMR* scheme of an *n*-dimensional array, *EKMR*(*n*) for the *EKMR* scheme of an *n*-dimensional array, and *MTM*(*S*, *D*, *n*) for the matrix transformation method of *TMR*(*n*) and *EKMR*(*n*), where *S* and *D* are the source and the destination representation schemes, respectively. We describe the *EKMR* and *TMR* schemes based on the row-major data layout (the L_{RM} layout function used in [8]). However, with some indexing changes, the *EKMR* and *TMR* schemes are also

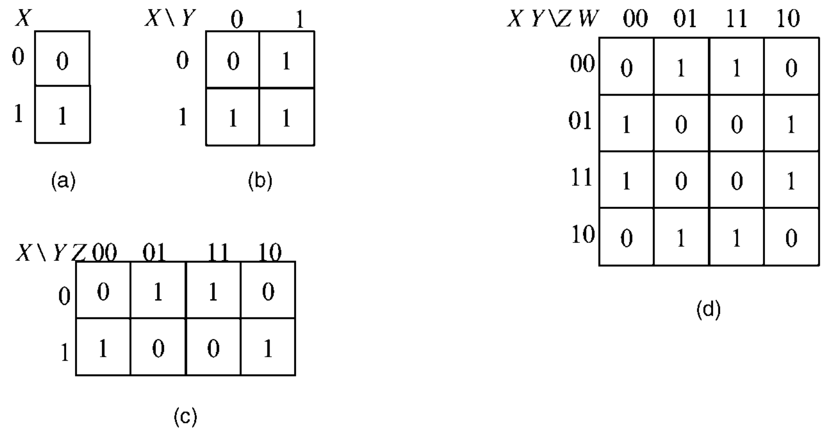

Fig. 1. Examples of the Karnaugh map. (a) 1-input for $f = X$. (b) 2-input for $f = X + Y$. (c) 3-input for $f = XZ' + X'Z$. (d) 4-input for $f = YW' + Y'W$.

suitable for the column-major data layout (the L_{CM} layout function used in [8]).

3.1 The *EKMR* and *MTM* Schemes for Three- and Four-Dimensional Arrays

In the *EKMR* scheme, a multidimensional array is represented by a set of two-dimensional arrays. The idea of the *EKMR* scheme is based on the Karnaugh map. The Karnaugh map technique is a method for minimizing a *Boolean expression*, usually aided by a rectangular map of the value of the expression for all possible input values. Input values are arranged in a *Gray code*. Fig. 1 shows examples of n-input Karnaugh maps, for $n = 1, \dots, 4$. It is clear that an n-input Karnaugh map uses n variables to reserve memory storage and represent all the 2^n possible combinations.

For 1-input Karnaugh map in Fig. 1a, we can use a variable X as a vector to store two (2^1) combinations. For the 2-input Karnaugh map in Fig. 1b, we can use two variables X and Y as a two-dimensional array (X: row, Y: column) to store four (2^2) combinations. For the 3-input Karnaugh map in Fig. 1c, we can use three variables X, Y, and Z as a two-dimensional array (X: row, $\{Y, Z\}$: column) to store eight (2^3) combinations. For the 4-input Karnaugh map in Fig. 1d, we can use four variables X, Y, Z, and W as a two-dimensional array ($\{X, Y\}$: row, $\{Z, W\}$: column) to store 16 (2^4) combinations. When n is less than or equal to 4, an n-input Karnaugh map can be drawn on a plane easily, that is, it can be represented by a two-dimensional array. Consequently, we use the concept of the Karnaugh map to represent the *EKMR* scheme. When $n = 1$, the *EKMR*(1) (1-input Karnaugh map) is simply a one-dimensional array. Similarly, when $n = 2$, the *EKMR*(2) (2-input Karnaugh map) is the traditional two-dimensional array. Therefore, the *EKMR*(n) has the same representation as the *TMR*(n) for $n = 1$ and 2. We now consider the *EKMR*(3) and the *EKMR*(4) schemes.

3.1.1 The EKMR(3) and MTM(S, D, 3)

Let $A[k][i][j]$ denote a three-dimensional array based on the *TMR*(3). Fig. 2a shows a three-dimensional array based on the *TMR*(3) with a size of $3 \times 4 \times 5$. In practice, a multidimensional array is stored in a linear memory address space for programming languages that support multidimensional arrays. Programming languages map the array index space into the linear memory address. Therefore, array $A[k][i][j]$ can be presented by the row-major data layout function

$$L_{RM}(k, i, j; 3, 4, 5) = k \times 4 \times 5 + i \times 5 + j$$

or the column-major data layout function

$$L_{CM}(k, i, j; 3, 4, 5) = k \times 4 \times 5 + j \times 4 + i.$$

The $L_{RM}(k, i, j; 3, 4, 5)$ ($L_{CM}(k, i, j; 3, 4, 5)$) is the memory location of the array element in the third dimension k, row i, and column j relative to the starting memory location of the array with a size of $3 \times 4 \times 5$.

According to the 3-input Karnaugh map, a three-dimensional array based on the *TMR*(3) can be presented by a two-dimensional array based on the *EKMR*(3). The corresponding *EKMR*(3) of array *A*[3][4][5], is shown in

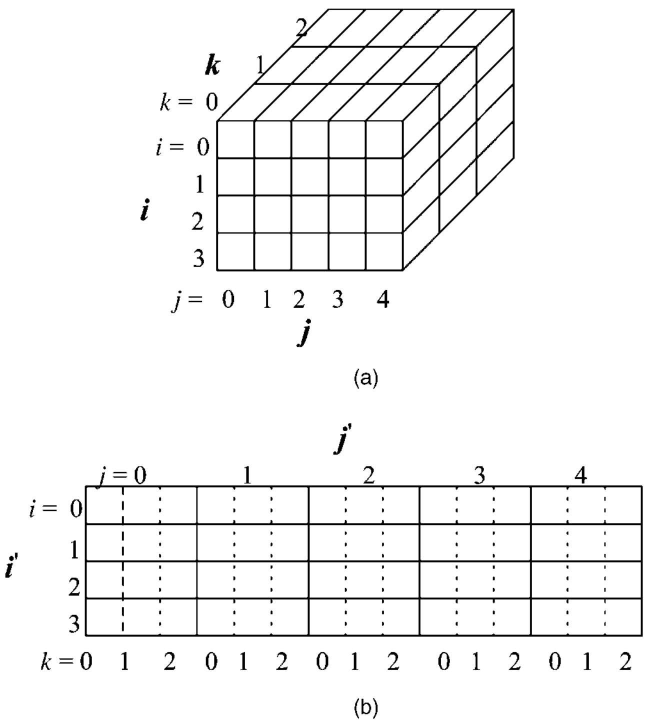

Fig. 2. (a) A $3 \times 4 \times 5$ array based on the $TMR(3)$. (b) A $4 \times (3 \times 5)$ array based on the $EKMR(3)$.

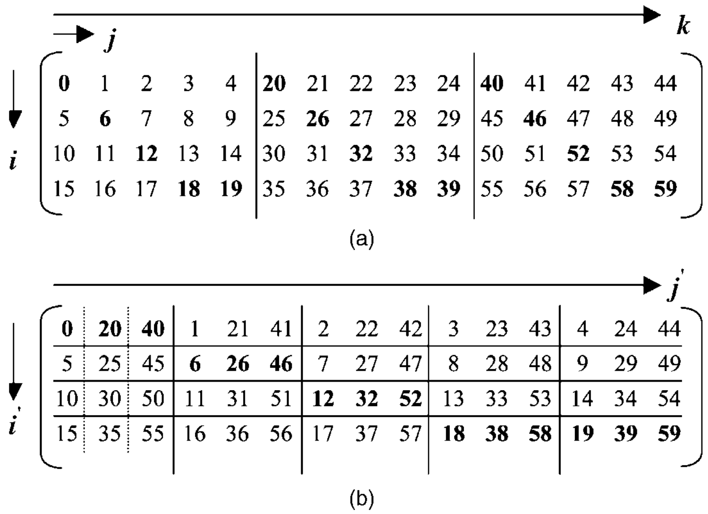

Fig. 3. (a) A three-dimensional array based on the *TMR*(3) in a 2D view. (b) The corresponding two-dimensional array based on the *EKMR*(3).

Fig. 2b. The *EKMR*(3) is represented by a two-dimensional array with the size of $4 \times (3 \times 5)$. The *EKMR*(3) can also be represented by the row-major data layout function $L'_{RM}(i', j'; 4, 15) = i' \times 15 + j'$ (or the column-major data layout function $L'_{CM}(i', j'; 5, 12) = j' \times 5 + i'$). The basic difference between the *TMR*(3) and the *EKMR*(3) is the placement of elements along the direction indexed by *k*. In the *EKMR*(3), we use the index variable i' to indicate the row direction and the index variable j' to indicate the column direction. Notice that the index variable i' is the same as the index variable *i*, whereas the index variable j' is a combination of the index variables *j* and *k* (the index variable i' is a combination of the index variables *i* and *k* and the index variable j' is the same as index variable *j* in the column-major data layout).

The analogy between the *EKMR*(3) and the 3-input Karnaugh map is that the index variables *i*, *j*, and *k* are corresponding to the variables *X*, *Y*, and *Z*, respectively (see Fig. 2 and Fig. 1c). A more concrete example based on the row-major data layout is given in Fig. 3. In Fig. 3a, a three-dimensional array based on the *TMR*(3) with a size of $3 \times 4 \times 5$ in a 2D view (three 4×5 two-dimensional arrays) is shown. Its corresponding *EKMR*(3) with a size of 4×15 is given in Fig. 3b.

Let *A*[*k*][*i*][*j*] denote a three-dimensional array based on the *TMR*(3) with a size of $r \times p \times q$, where *p*, *q*, and *r* are index variables along the row, column, and the third dimension. Let $A'[i'][j']$ denote the array based on the *EKMR*(3) corresponding to array *A*. From previous discussion, we have that A' is a two-dimensional array of size $p \times (r \times q)$. Assume that arrays *A* and A' are stored in the row-major data layout. For arrays *A* and A', they can be presented by $L_{RM}(k, i, j; r, p, q) = k \times (p \times q) + i \times (q) + j$ and $L'_{RM}(i', j'; p, r \times q) = i' \times (r \times q) + j'$, respectively. The *MTM*(*S*,*D*, 3) is defined as the mapping function for $L_{RM}(k, i, j; r, p, q)$ and $L'_{RM}(i', j'; p, r \times q)$ and is given as follows (the *MTM*(*S*,*D*, 3) in the column-major data layout can be obtained in a similar way):

$$L_{RM}(k, i, j; r, p, q) \rightarrow L'_{RM}(i', j'; p, rq),$$
$$\text{where} \begin{cases} i' = i'; \\ j' = j \times r + k; \\ A[k][i][j] \rightarrow A'[i][j \times r + k]; \end{cases} \quad (1)$$

$$L'_{RM}(i', j'; p, r \times q) \rightarrow L_{RM}(k, i, j; r, p, q),$$
$$\text{where} \begin{cases} i = i'; \\ k = j' \% r; \\ j = j'/r; \\ A'[i'][j'] \rightarrow A[j' \% r][i'][j'/r]; \end{cases} \quad (2)$$

We can apply the *MTM*(*S*,*D*, 3) to translate a three-dimensional array based on the *TMR*(3) to a two-dimensional array based on the *EKMR*(3) and vice versa. For example, for an array element $A[1][0][0]$ with value 20 in Fig. 3a, it can be presented by the row-major data layout function $L_{RM}(1, 0, 0; 3, 4, 5) = 1 \times (4 \times 5) + 0 \times (5) + 0 = 20$ (to map the array index space $A[1][0][0]$ into the linear memory address space *A*[20]). According to (1), the corresponding array element of *A*[1][0][0] in the *TMR*(3) is $A'[0][1]$ in the *EKMR*(3). On the other hand, for an array element $A'[2][6]$ with value 12 in the *EKMR*(3) as shown in Fig. 3b, it can be presented by the row-major data layout function $L'_{RM}(2, 6; 4, 15) = 2 \times 15 + 6 = 36$ (to map the array index space $A'[0][1]$ into the linear memory address space $A'[36]$). According to (2), the corresponding array element of $A'[2][6]$ in *EKMR*(3) is *A*[0][2][2] in the *TMR*(3).

3.1.2 The EKMR(4) and MTM(S, D, 4)

Let *A*[*l*][*k*][*i*][*j*] denote a four-dimensional array with a size of $2 \times 3 \times 4 \times 5$ based on the *TMR*(4). Array *A* can be presented by the row-major data layout function

$$L_{RM}(l, k, i, j; 2, 3, 4, 5) = l \times 3 \times 4 \times 5 + k \times 4 \times 5 + i \times 5 + j$$

or the column-major data layout function

$$L_{CM}(l, k, i, j; 2, 3, 4, 5) = l \times 3 \times 4 \times 5 + k \times 4 \times 5 + j \times 4 + i.$$

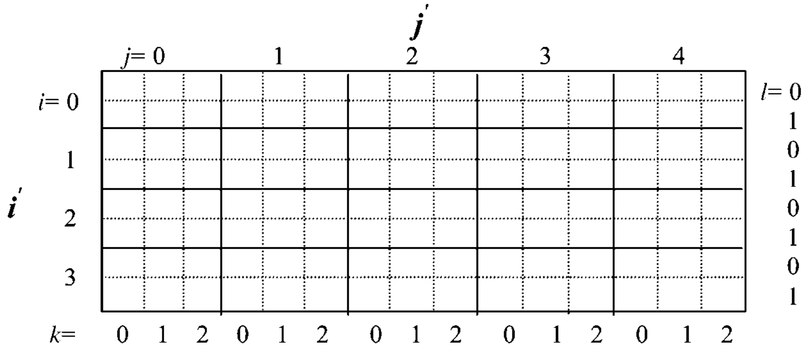

Fig. 4. A $(2 \times 4) \times (3 \times 5)$ array based on the *EKMR*(4).

The way to obtain the *EKMR*(4) based on the 4-input Karnaugh map is similar to that of the *EKMR*(3). Fig. 4 illustrates a corresponding *EKMR*(4) of array $A[2][3][4][5]$ with a size of $(2 \times 4) \times (3 \times 5)$.

The *EKMR* (4) can also be represented by the row-major data layout function

$$L'_{RM}(i', j'; 8, 15) = i' \times 15 + j'$$

(or the column-major data layout function

$$L'_{CM}(i', j'; 10, 12) = j' \times 10 + i').$$

The basic difference between the *TMR*(4) and the *EKMR*(4) is the placement of elements along the direction indexed by *l* and *k*. In the *EKMR*(4), we also use the index variable i' to indicate the row direction and the index variable j' to indicate the column direction. Notice that the index variable i' is a combination of the index variables *l* and *I* and the index variable j' is a combination of the index variables *j* and *k* (the index variable i' is a combination of the index variables *i* and *k* and the index variable j' is a combination of the index variables *j* and *l* in the column-major data layout).

Let *A*[*l*][*k*][*i*][*j*] be a four-dimensional array based on the *TMR*(4) with a size of $s \times r \times p \times q$, where the index variable *l* indicates the fourth dimension with a size of *s*. Let $A'[i'][j']$ be the corresponding array based on the *EKMR*(4), which is of the size of $(s \times p) \times (r \times q)$. Assume that arrays *A* and A' are stored in the row-major data layout. For arrays *A* and A', they can be presented by $L_{RM}(l, k, i, j; s, r, p, q) = l \times r \times p \times q + k \times p \times q + i \times q + j$ and $L'_{RM}(i', j'; s \times p, r \times q) = i' \times r \times q + j'$, respectively. The *MTM*(*S*, *D*, 4) is defined as the mapping function of $L_{RM}(l, k, i, j; s, r, p, q)$ and $L'_{RM}(i', j'; s \times p, r \times q)$ and is given as follows (the $MTM(S, D, 4)$ in column-major data layout can be obtained in a similar way):

$$L_{RM}(l, k, i, j; s, r, p, q) \rightarrow L'_{RM}(i', j'; s \times p, r \times q),$$
$$\text{where} \begin{cases} i' = i \times s + l; \\ j' = j \times r + k; \\ A[l][k][i][j] \rightarrow A'[i \times s + l][j \times s + k]; \end{cases} \quad (3)$$

$$L'_{RM}(i', j'; s \times p, r \times q) \rightarrow LRM(l, k, i, j; s, r, p, q),$$
$$\text{where} \begin{cases} l = i'\%; \\ k = j'\%r; \\ i = i'/s; \\ j = j'/r; \\ A'[i'][j'] \rightarrow A[i'\%s][j'\%r][i'/s][j'/r]; \end{cases} \quad (4)$$

3.2 The $EKMR(n)$ and $MTM(S, D, n)$

Based on the *EKMR*(4), we can generalize our result to the *n*-dimensional array. In general, we can use 2^{n-4} 4-input Karnaugh maps to represent an *n*-input $(n \geq 4)$ one. Similarly, we can use a set of the *EKMR*(4) to construct the *EKMR*(*n*). Assume that there is an *n*-dimensional array with a size of *m* along each dimension, i.e., an m^n array based on the *TMR*(*n*). Since the *EKMR*(*n*) can be represented by m^{n-4} *EKMR*(4), we need a structure to link all arrays based on the *EKMR*(4). Here, we use a one-dimensional array *X* with a size of m^{n-4} to link these *EKMR*(4). By applying a data layout function, such as the row-major data layout function or the column-major data layout function, we can determine the one-to-one mapping between *X* and m^{n-4} *EKMR*(4). Assume that there is a six-dimensional array *A*[*n*][*m*][*l*][*k*][*i*][*j*] with a size of $3 \times 2 \times 2 \times 3 \times 4 \times 5$ based on the *TMR*(6). Fig. 5 shows the corresponding *EKMR*(6), represented by six (3×2) arrays based on the *EKMR*(4) each with a size of $(2 \times 4) \times (3 \times 5)$, of array *A*[*n*][*m*][*l*][*k*][*i*][*j*]. In Fig. 5, a one-dimensional array *X* with a size of six is used to link these *EKMR*(4). If the row-major data layout function is used for the array, $X[0]$, $X[1]$, $X[2]$, $X[3]$, $X[4]$, and $X[5]$ are linked to *EKMR*(4)

$$A[0][0][l][k][i][j],$$
$$A[0][1][l][k][i][j],$$
$$A[1][0][l][k][i][j],$$
$$A[1][1][l][k][i][j],$$
$$A[2][0][l][k][i][j], \text{ and}$$
$$A[2][1][l][k][i][j],$$

respectively.

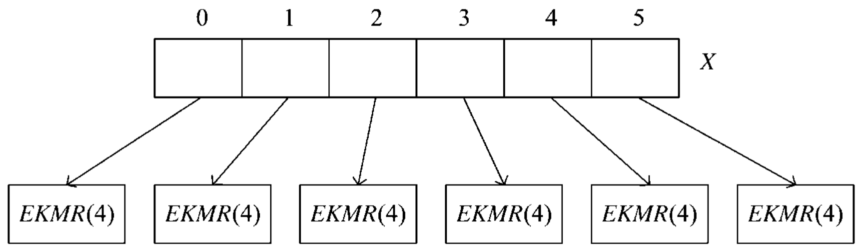

Fig. 5. An example of the *EKMR*(6).

Let $A[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j]$ be an n-dimensional array in the *TMR*(n) with a size of

$$t_{n-4} \times t_{n-3} \times \ldots t_1 \times s \times r \times p \times q,$$

where the index variable m_{n-4} indicates the nth dimension with a size of t_{n-4}. Let $A'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i'][j']$ be the corresponding $EKMR(n)$ of $A[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j]$ and $A'_x[i'][j']$ be an $EKMR(4)$ of $A'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i'][j']$ with a size of $(s \times p) \times (r \times q)$, where $A'_x[i'][j']$ is linked by array element $X[x]$ in array X. For array A, it can be presented by the row-major data layout function

$$\begin{aligned}&L_{RM}(m_{n-4}, m_{n-3}, \ldots, l, k, i, j; t_{n-4}, t_{n-3}, \ldots, s, r, p, q)\\ &\quad = m_{n-4} \times t_{n-3} \times \ldots \times s \times r \times p \times q + \ldots + l \times r \times p \times q\\ &\qquad + k \times p \times q + i \times q + j.\end{aligned}$$

For array $A'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i'][j']$, it can be presented by the row-major data layout function

$$\begin{aligned}&L'_{RM(M_{n-4},M_{n-3},\ldots,M_1)}(i', j'; s \times p, r \times q)\\ &\quad = x \times ((s \times p) \times (r \times q)) + i' \times (r \times q) + j',\end{aligned}$$

where

$$\begin{aligned}x =\ &m_{n-4} \times t_{n-3} \times \ldots \times s \times r \times p \times q + \ldots +\\ &m_1 \times s \times r \times p \times q.\end{aligned}$$

The $MTM(S, D, n)$ is defined as the mapping function of

$$L_{RM}(m_{n-4}, m_{n-3}, \ldots, l, k, i, j; t_{n-4}, t_{n-3}, \ldots, s, r, p, q)$$

and

$$L'_{RM(M_{n-4},M_{n-3},\ldots,M_1)}(i', j'; s \times p, r \times q)$$

and is given as follows (the $MTM(S, D, n)$ in the column-major data layout can be obtained in a similar way):

$$\begin{aligned}&L_{RM}(m_{n-4}, m_{n-3}, \ldots, l, k, i, j; t_{n-4}, t_{n-3}, \ldots, s, r, p, q)\\ &\quad \rightarrow L'_{RM(M_{n-4},M_{n-3},\ldots,M_1}(i', j'; s \times p, r \times q),\\ &\quad\quad \text{where}\begin{cases} i' = i \times s + l;\\ j' = j \times r + k;\\ A(m_{n-4}, m_{n-3}, \ldots, m_1, l, k, i, j)\\ \rightarrow A'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i \times s + l][j \times r + k];\end{cases}\end{aligned} \tag{5}$$

$$\begin{aligned}&L'_{RM(M_{n-4},M_{n-3},\ldots,M_1)}(i', j'; s \times p, r \times q)\\ &\quad \rightarrow L_{RM}(m_{n-4}, m_{n-3}, \ldots, l, k, i, j; t_{n-4}, t_{n-3}, \ldots, s, r, p, q),\\ &\quad\quad \text{where}\begin{cases} l = i'\%s;\\ i = i'/s;\\ k = j'\%r;\\ j = j'/r;\\ A'_{(m_{n-4},m_{n-3},\ldots)}[i'][j']\\ \rightarrow A[m_{n-4}][m_{n-3}]\ldots[m_1][i'\%s][j'\%r][i'/s][j'/r];\end{cases}\end{aligned} \tag{6}$$

4 Comparisons of the *TMR* and *EKMR* Schemes

The *TMR* and the *EKMR* are both representation schemes for multidimensional arrays. Different data layout functions can be applied to them to get different data layouts. To compare the *TMR* and the *EKMR* schemes, we design algorithms of multidimensional array operations, matrix-matrix addition/subtraction, and matrix-matrix multiplication, according to the row-major data layout function for both schemes. Algorithms based on the column-major data layout function can be obtained by changing the order of array indices. Based on these algorithms, we analyze their theoretical performance. We do not consider algorithms based on the recursive data layout functions [6], [7]. The reason is that, how to select a recursive data layout function, such that a multidimensional array operation algorithm based on the *TMR* scheme has the best performance, is an open question [6], [7]. In the following, we will first present algorithms for three-dimensional arrays. Then, extend them to higher dimensional arrays.

4.1 Matrix-Matrix Addition/Subtraction Algorithms

Let A and B be two $n \times n \times n$ three-dimensional arrays based on the $TMR(3)$. The algorithm for $C = A \pm B$ based on the $TMR(3)$ can be illustrated as follows:

*Algorithm matrix-matrix_addition/subtraction_*$TMR(3)$
1. *for* $(k = 0; k < n; k++)$
2. *for* $(i = 0; i < n; i++)$
3. *for* $(j = 0; j < n; j++)$
4. $C[k][i][j] = A[k][i][j] \pm B[k][i][j];$

end of matrix-matrix addition/subtraction $TMR(3)$

Let A' and B' be the corresponding arrays of A and B based on the $EKMR(3)$. The algorithm for $C' = A' \pm B'$ based on the $EKMR(3)$ is given as follows, where a dummy variable r is used for summation over the j' direction:

Algorithm matrix-matrix_addition/subtraction_$EKMR(3)$
$r = n \times n;$
for $(i' = 0; i' < n; i' + +)$
  for $(j' = 0; j' < r; j' + +)$
    $C'[i'][j'] = A'[i'][j'] \pm B'[i'][j'];$
end of matrix-matrix addition/subtraction $EKMR(3)$

Let

$$A[m_{n-4}][m_{n-3}] \dots [m_1][l][k][i][j] \text{ and }$$
$$B[m_{n-4}][m_{n-3}] \dots [m_1][l][k][i][j]$$

be two m^n n-dimensional arrays. Let $A'_{(m_{n-4},m_{n-3},\dots,m_1)}[i'][j']$ and $B'_{(m_{n-4},m_{n-3},\dots,m_1)}[i'][j']$ be two corresponding $EKMR(n)$ whose $EKMR(4)$ has a size of $(m \times m) \times (m \times m)$. The algorithms for the matrix-matrix addition/subtraction based on the $EKMR(n)$ and the $TMR(n)$ is given as follows:

Algorithm matrix-matrix_addition/subtraction_$TMR(n)$
for $(m_{n-4} = 0; m_{n-4} < m; m_{n-4} + +)$
  for $(m_{n-3} = 0; m_{n-3} < m; m_{n-3} + +)$
    $\dots$ /*From loop m_{n-4} to loop m_1*/
      for $(l = 0; l < m; l + +)$
    . *for* $(k = 0; k < m; k + +)$
  . *for* $(i = 0; i < m; i + +)$
  . *for* $(j = 0; j < m; j + +)$
  . $C[m_{n-4}][m_{n-3}] \dots [m_1][l][k][i][j]$
    $= A[m_{n-4}][m_{n-3}] \dots [m_1][l][k][i][j]$
    $+B[m_{n-4}][m_{n-3}] \dots [m_1][l][k][i][j];$
end_of_matrix-matrix_addition/subtraction_$TMR(n)$

Algorithm matrix-matrix_addition/subtraction_$EKMR(n)$
$r = m \times m;$
for $(x = 0; x < m^{n-4}; x + +)$
  for $(i' = 0; i' < r; i' + +)$
    for $(j' = 0; j' < r; j' + +)$
      $C'_x[i'][j'] = A'_x[i'][j'] \pm B'_x[i'][j'];$
end_of_matrix-matrix_addition/subtraction_$EKMR(n)$

4.2 Matrix-Matrix Multiplication Algorithms

Let A and B be two $n \times n \times n$ three-dimensional arrays. An algorithm of the matrix-matrix multiplication $C = A \times B$ based on the $TMR(3)$ in $KIJM$ order is depicted as follows:

Algorithm matrix-matrix_multiplication_KIJM_order_TMR(3)
for $(k = 0; k < n; k + +)$
  for $(i = 0; i < n; i + +)$
    for $(j = 0; j < n; j + +)$
      for $(m = 0; m < n; m + +)$
        $C[k][i][j] = C[k][i][j] + A[k][i][m] \times B[k][m][j];$
end_of_matrix-matrix_multiplication_$KIJM$_order_$TMR(3)$

By using the $MTM(S, D, 3)$, we can translate the algorithm for $C = A \times B$ based on the $TMR(3)$ to the naive algorithm for $C' = A' \times B'$ based on the $EKMR(3)$. The naive algorithm for $C' = A' \times B'$ based on the $EKMR(3)$ is given as follows:

Algorithm naive_matrix-matrix_multiplication_$EKMR(3)$
$r = n \times n;$
for $(i' = 0; i' < n; i' + +)$
  for $(j' = 0; j' < r; j' + +)$
    for $(m = 0; m < n; m + +)$
      $v = m \times n;$
    $C'[i'][j'] = C'[i'][j'] + A'[i'][v + j'\%n] \times B'[m][j'];$
end_of_nave_matrix-matrix_multiplication_$EKMR(3)$

However, the performance of the naive algorithm for $C' = A' \times B'$ based on the $EKMR(3)$ is worse than that based on the $TMR(3)$. There are two reasons. First, the access patterns of array elements for matrix-matrix multiplication based on the $EKMR(3)$ and the $TMR(3)$ are the same. Therefore, the performance of algorithms for matrix-matrix multiplication based on the $TMR(3)$ and the $EKMR(3)$ is the same. Second, the naive algorithm for $C' = A' \times B'$ based on the $EKMR(3)$ has poorer spatial locality and more expensive array subscripts than that based on the $TMR(3)$. Since we do not exploit advantages for the structure of the $EKMR(3)$ in the naive algorithm of $C' = A' \times B'$, based on O'Boyle and Knijnenburg [29], a redesigned efficient algorithm of $C' = A' \times B'$ based on the $EKMR(3)$ is given as follows:

Algorithm matrix-matrix_multiplication_row-major order_$EKMR(3)$
for $(i = 0; i < n; i + +)$
  for $(j = 0; j < n; j + +)$
    $v = j \times n;$
      for $(m = 0; m < n; m + +)$
        $r = m \times n;$
          for $(k = 0; k < n; k + +)$
$C'[i][k + r] = C'[i][k + r] + A'[i][k + v] \times B'[j][k + r];$
end_of_matrix-matrix_multiplication_row-major order_$EKMR(3)$

There are two advantages for the row-major order algorithm of matrix-matrix multiplication based on the $EKMR(3)$. First, the row-major order algorithm can decrease the access numbers of different elements in array B'. Second, the structure of $EKMR(3)$ can aggregate array elements that have the same values of index variables j and i. These array elements in array A' will be operated with the same array element in array B'. Therefore, the cache miss rate for array operations based on the $EKMR(3)$ may be less than that based on the $TMR(3)$. The algorithms for the matrix-matrix multiplication based on the $TMR(n)$ and the $EKMR(n)$ is given as follows:

Algorithm matrix-matrix_multiplication_$TMR(n)$
for $(m_{n-4} = 0; m_{n-4} < m; m_{n-4} + +)$
  for $(m_{n-3} = 0; m_{n-3} < m; m_{n-3} + +)$
    /*From loop m_{n-4} to loop m_1*/
      for $(l = 0; l < m; l + +)$
        for $(k = 0; k < m; k + +)$
  . *for* $(i = 0; i < m; i + +)$
  . *for* $(j = 0; j < m; j + +)$
  . *for* $(m_0 = 0; m_0 < m_0; m + +)$
  . $C[m_{n-4}][m_{n-3}] \dots [m_1][l][k][i][j]$
    $= C[m_{n-4}][m_{n-3}] \dots [m_1][l][k][i][j]$

```
                    + A[m_{n-4}][m_{n-3}]...[m_1][l][k][I][m]
                    × B[m_{n-4}][m_{n-3}]...[m_1][l][k][m][j];
end_of_matrix-matrix_multiplication_TMR(n)
```

Algorithm matrix-matrix_multiplication_row-major order_EKMR(n)

```
for (x = 0; x < m_{n-4}; x++)
  for (i = 0; i < m; i++)
    t = i × m;
    for (j = 0; j < m; j++)
      v = j × m;
      for (l = 0; l < m; l++)
        w = t + l;
        u = l + v;
        for (m_0 = 0; m_0 < m; m_0++)
          r = m_0 × m;
          for (k = 0; k < m; k++)
            C'_x[w][k + r] = C'_x[w][k + r]
                             + A'_x[w][k + v] × B'_x[u][k + r];
end_of_matrix-matrix_multiplication_row-major order_EKMR(n)
```

4.3 Theoretical Analysis

In the following, we analyze the theoretical performance for algorithms presented in this section in two aspects, the cost of addition/subtraction/multiplication operators and the cache effect. For the cost of addition/subtraction/multiplication operators, we analyze the numbers of the addition/subtraction/multiplication operators for the index computations of array elements and array operations in these algorithms. In this aspect, we use the full indexing cost for each array element to analyze the performance of algorithms based on the *TMR* and *EKMR* schemes. It is no doubt that the compiler optimization techniques do achieve incremental addressing. However, we do not consider any compiler optimization technique in the theoretical analysis. The reason is that it is difficult to analyze the effects of compiler optimization techniques since the effects of the optimization may depend on the addressing mode of a machine, the way to write the programs, the efficiency of the optimization techniques, etc. To see the optimization effects, in the experimental tests, we will show both results for all *C* programs with and without compiler optimization techniques.

To analyze the cache effect, an algorithm called *LoopCost* that was proposed by Carr et al. [4], [28] is used to compute the costs of various loop orders of an array operation. In the algorithm, *LoopCost(l)* is the number of cache line accessed by the innermost loop *l*. The value of *LoopCost(l)* reflects the cache miss rate. The smaller the *LoopCost(l)*, the smaller the cache miss rate. According to *LoopCost(l)*, the best loop orders with a specific innermost loop *l*can be determined. In the analysis, we assume that the cache line size used in algorithm *LoopCost* is *r*.

4.3.1 Costs of Matrix-Matrix Addition/Subtraction Algorithms

A. The Costs of Addition/Subtraction/Multiplication Operators. Algorithms for matrix-matrix addition/subtraction based on the $TMR(3)$ in *KIJ* order and the $EKMR(3)$ were described in Section 4.1. Assume that A and B are two $m \times m \times m$ three-dimensional arrays based on the $TMR(3)$ and that A' and B' are the corresponding arrays of A and B based on the $EKMR(3)$ with the size of $m \times m^2$. For arrays A and A', they can be presented by

$$L_{RM}(k, i, j; m, m, m) = k \times (m \times m) + i \times (m) + j$$

and

$$L'_{RM}(i', j'; m, m^2) = i' \times m^2 + j',$$

respectively. Assume that the cost for an addition/subtraction operator and a multiplication operator is β and α, respectively. For the $TMR(3)$ and the $EKMR(3)$, the cost of index computation of an array element is $(3\alpha + 2\beta)$ and $(\alpha + \beta)$, respectively. Similarly, for the $TMR(4)$ and the $EKMR(4)$, the cost of index computation of an array element is $(6\alpha + 3\beta)$ and $(\alpha + \beta)$, respectively. Assume that

$$A[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j] \text{ and}$$
$$B[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j]$$

are two m^n n-dimensional arrays and $A'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i'][j']$ and $B'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i'][j']$ are two corresponding $EKMR(n)$ whose $EKMR(4)$ has a size of m^4. For arrays A and A', they can be presented by

$$\begin{aligned} &L_{RM}(m_{n-4}, m_{n-3}, \ldots, l, k, i, j; m, m, \ldots, m, m, m, m) \\ &= m_{n-4} \times m \times \ldots \times m \times m \times m \times m + \ldots \\ &\quad + l \times m \times m \times m + k \times m \times m + i \times m + j \end{aligned}$$

and

$$L'_{RM(M_{n-4},M_{n-3},\ldots,M_1)}(i', j'; m^2, m^2) = x \times m^4 + i' \times m^2 + j',$$

respectively. For the $TMR(n)$ and the $EKMR(n)$, where $n \geq 5$, the cost of index computation of an array element is $\frac{n(n-1)}{2}\alpha + (n-1)\beta$ and $(2\alpha + 2\beta)$, respectively. Therefore, we can see that the cost of index computations of array elements based on the *EKMR* scheme is less than that based on the *TMR* scheme. The reason is that the $EKMR(n)$ is presented by a set of two-dimensional arrays.

For the matrix-matrix addition/subtraction algorithm based on the $TMR(3)$, there are three arrays *A*, *B*, and *C* involved in an addition/subtraction operation. The costs of index computations of array elements and the array operation are $(9\alpha + 6\beta)m^3$ and βm^3, respectively. In the matrix-matrix addition/subtraction algorithm based on the $EKMR(3)$, there are three arrays A', B', and C' involved in an addition/subtraction operation. Besides, there is an extra cost for the cost of array operations to compute the value of *r* in the algorithm. The costs of index computations of array elements and array operations are $(3\alpha + 3\beta)m^3$ and $\beta m^3 + \alpha$, respectively. Table 1 shows the costs of index computations of array elements and array operations for algorithms of matrix-matrix addition/subtraction based on the TMR(n) and the EKMR(n). In Table 1, the improved rate is defined as follows:

$$Improved\ Rate(\%) = \frac{Total(TMR) - Total(EKMR)}{Total(TMR)} \times 100. \tag{7}$$

TABLE 1
The Costs of Index Computations of Array Elements and Array Operations for Algorithms of Matrix-Matrix Addition/Subtraction Based on the *TMR*(*n*) and the *EKMR*(*n*)

3-*D*			
Schemes \ *Costs*	*Index computations*	*Array operations*	*Total*
TMR	$(9\alpha+6\beta)m^3$	βm^3	$(9\alpha+7\beta)m^3$
EKMR	$(3\alpha+3\beta)m^3$	$\beta m^3+\alpha$	$(3\alpha+4\beta)m^3+\alpha$
Improved Rate (%)	$(\frac{2}{3}-\frac{1}{9m^3})\times 100$		
4-*D*			
Schemes \ *Costs*	*Index computations*	*Array operations*	*Total*
TMR	$(18\alpha+9\beta)m^4$	βm^4	$(18\alpha+10\beta)m^4$
EKMR	$(3\alpha+3\beta)m^4$	$\beta m^4+\alpha$	$(3\alpha+4\beta)m^4+\alpha$
Improved Rate (%)	$(\frac{5}{6}-\frac{1}{18m^4})\times 100$		
n-*D* ($n \geq 5$)			
Schemes \ *Costs*	*Index computations*	*Array operations*	*Total*
TMR	$(\frac{3n(n-1)}{2}\alpha+(3n-3)\beta)m^n$	βm^n	$(\frac{3n(n-1)}{2}\alpha+(3n-2)\beta)m^n$
EKMR	$(6\alpha+6\beta)m^n$	$\beta m^n+\alpha$	$(6\alpha+7\beta)m^n+\alpha$
Improved Rate (%)	$(1-\frac{4}{n^2-n})\times 100$		

Since the cost of α is much larger than that of β for the improved rate in Table 1, we only consider the effect of α. From Table 1, we can see that the costs of index computations of array elements and array operations for algorithms of matrix-matrix addition/subtraction based on the $EKMR(n)$ are less than those based on the $TMR(n)$. In Table 1, for $n = 3$ and 4, the improved rate increases as the size m increases. For $n \geq 5$, the improved rate is $(1 - \frac{4}{n^2-n}) \times 100$ that is independent of the size m. The improved rate increases as the array dimension n increases.

B. The Cost of Cache Effect Table 2 shows the $LoopCost(l)$ for algorithms of matrix-matrix addition/subtraction based on the $EKMR(3)$ and the $TMR(3)$ with various innermost loop indices *K*, *I*, and *J*. From Table 2, we can see that the algorithm for matrix-matrix addition/subtraction based on the $EKMR(3)$ has the smallest $LoopCost(l)$. In Table 2, for the $TMR(3)$, we can see that the algorithm whose innermost loop index is *J* has smallest $LoopCost(l)$.

Algorithms for matrix-matrix addition/subtraction based on the $TMR(n)$ in $M_{n-4}M_{n-3}\dots M_1LKIJ$ order and the $EKMR(n)$ were described in Section 4.1. Table 3 shows the $LoopCost(l)$ for algorithms of matrix-matrix addition/subtraction based on the $EKMR(n)$ and the $TMR(n)$ with various innermost loop indices $M_{n-4}, M_{n-3}, \dots, M1, L, K, I,$ and *J*. From Table 3, we can see that the algorithm for matrix-matrix addition/subtraction based on the $EKMR(n)$ has the smallest $LoopCost(l)$. In Table 3, for the $TMR(n)$, we can see that the algorithm whose innermost loop index is *J* has smallest $LoopCost(l)$. The improved rate of the $EKMR(n)$ with respect to the $TMR(n)$ whose innermost loop is *j* is

$$\left(1 - \left\lceil\frac{m^2}{r}\right\rceil \div \left(\left\lceil\frac{m}{r}\right\rceil \times m\right)\right) \times 100,$$

for $n \geq 3$. The improved rate is independent of the array dimension *n*. When *m* is divisible by *r*, the improved rate is 0, that is, the number of cache line accessed for the $EKMR(n)$ is the same as that of the $TMR(n)$. When *m* is not divisible by *r*, the improved rate is

$$\left(1 - \left\lceil\frac{m^2}{r}\right\rceil \div \left(\left\lceil\frac{m}{r}\right\rceil \times m\right)\right) \times 100$$
$$= \left(1 - \frac{m\delta + r + 1}{m(\delta+1)}\right) \times 100,$$

TABLE 2
The $LoopCost(l)$ for Algorithms of Matrix-Matrix Addition/Subtraction Based on the $TMR(3)$ and the $EKMR(3)$

LoopCost(*l*)					
RefGroup		$C[k][i][j]$	$A[k][i][j]$	$B[k][i][j]$	*Total*
TMR(3)	*K*	$m \times m^2$	$m \times m^2$	$m \times m^2$	$3m \times m^2$
	I	$m \times m^2$	$m \times m^2$	$m \times m^2$	$3m \times m^2$
	J	$\left\lceil\frac{m}{r}\right\rceil \times m^2$	$\left\lceil\frac{m}{r}\right\rceil \times m^2$	$\left\lceil\frac{m}{r}\right\rceil \times m^2$	$3\left\lceil\frac{m}{r}\right\rceil \times m^2$
RefGroup		$C'[i'][j']$	$A'[i'][j']$	$B'[i'][j']$	*Total*
EKMR(3)		$\left\lceil\frac{m^2}{r}\right\rceil \times m$	$\left\lceil\frac{m^2}{r}\right\rceil \times m$	$\left\lceil\frac{m^2}{r}\right\rceil \times m$	$3\left\lceil\frac{m^2}{r}\right\rceil \times m$

TABLE 3
The $LoopCost(l)$ for Algorithms of Matrix-Matrix Addition/Subtraction Based on the $TMR(n)$ and the $EKMR(n)$

LoopCost(l)					
RefGroup		$C[m_{n-4}]\ldots[i][j]$	$A[m_{n-4}]\ldots[i][j]$	$B[m_{n-4}]\ldots[i][j]$	*Total*
TMR(n)	*Others*	$m \times m^{n-1}$	$m \times m^{n-1}$	$m \times m^{n-1}$	$3m \times m^{n-1}$
	J	$\lceil \frac{m}{r} \rceil \times m^{n-1}$	$\lceil \frac{m}{r} \rceil \times m^{n-1}$	$\lceil \frac{m}{r} \rceil \times m^{n-1}$	$3\lceil \frac{m}{r} \rceil \times m^{n-1}$
RefGroup		$C_x[i][j]$	$A_x[i][j]$	$B_x[i][j]$	*Total*
EKMR(n)		$\lceil \frac{m^2}{r} \rceil \times m^{n-2}$	$\lceil \frac{m^2}{r} \rceil \times m^{n-2}$	$\lceil \frac{m^2}{r} \rceil \times m^{n-2}$	$3\lceil \frac{m^2}{r} \rceil \times m^{n-2}$

where δ is the quotient of $m \div r$. If m is much lager than r, the improved rate

$$1 - \frac{m\delta + r + 1}{m(\delta + 1)} \approx 0.$$

C. Discussions. The overall performance of these algorithms should consider the costs of addition/subtraction/multiplication operators and the cache effect. Assume that the ratios of the cost of addition/subtraction/multiplication operators and the cost of the cache effect to the overall cost of an algorithm are $p : 1$ and $(1 - p) : 1$, respectively. From the above analysis, for the $TMR(n)$, the time complexities of the addition/subtraction/multiplication operators and the cache effect for the matrix-matrix addition/subtraction algorithm are $O(m^n)$ and $O(\lceil \frac{m}{r} \rceil m^{n-1})$, respectively. The time complexity of the addition/subtraction/multiplication operators is larger than that of the cache effect. For a fixed array dimension n, the ratio of the cost of addition/subtraction/multiplication operators increases as the array size m increases. For a fixed array size m, the ratio of the cost of addition/subtraction/multiplication operators is independent of the array dimension n. The overall improved rate for algorithms of matrix-matrix addition/subtraction based on the $EKMR(n)$ with respect to those of the $TMR(n)$ is given in Table 4.

From Table 4, for three- and four-dimensional arrays, we have two remarks.

Remark 1. If m is divisible by r, the overall improved rates for three- and four-dimensional arrays are determined by $\left(\frac{2}{3} - \frac{1}{9m^3}\right) \times p \times 100$ and $\left(\frac{5}{6} - \frac{1}{18m^4}\right) \times p \times 100$, respectively. The overall improved rate increases as the array size m increases (the ratio p increases as the array size m increases).

Remark 2. If m is not divisible by r, the overall improved rates for three- and four-dimensional arrays depend on the array size m and the ratio p. When the array size increases from m_1 to m_2, the overall improved rate for three-dimensional arrays increase if the ratio of the cost of addition/subtraction/multiplication operators increases from p_1 to p_2 and

$$\begin{aligned}\Delta p &= p_2 - p_1 \\ &> \left(\left(\frac{m_2\delta_2 + r + 1}{m_2(\delta_2 + 1)} - \frac{m_1\delta_1 + r + 1}{m_1(\delta_1 + 1)}\right)(1 - p_1)\right) \\ &\div \left(\frac{m_2\delta_2 + r + 1}{m_2(\delta_2 + 1)} - \frac{1}{3}\right)\end{aligned}$$

is satisfied. The overall improved rate for three-dimensional arrays are constant if

$$\begin{aligned}\Delta p &= p_2 - p_1 \\ &= \left(\left(\frac{m_2\delta_2 + r + 1}{m_2(\delta_2 + 1)} - \frac{m_1\delta_1 + r + 1}{m_1(\delta_1 + 1)}\right)(1 - p_1)\right) \\ &\div \left(\frac{m_2\delta_2 + r + 1}{m_2(\delta_2 + 1)} - \frac{1}{3}\right)\end{aligned}$$

is satisfied. For other cases, the overall improved rates for three-dimensional arrays will decrease. For example, for three-dimensional arrays, when the values of m, r, and p are 10, 4, and p_1, respectively, the overall improved rate is $\frac{1}{2}p_1 + \frac{1}{6}$. When the values of m, r, and p are 30, 4, and p_2, respectively, the overall improved rate is $\frac{27}{48}p_2 + \frac{5}{48}$. If $p_1 = 0.1$ and $p_2 = 0.30$,

$$\left(\frac{27}{48}p_2 + \frac{5}{48}\right) - \left(\frac{1}{2}p_1 + \frac{1}{6}\right) = \frac{2.7}{48} < 0.$$

The overall improved rate increases. If $p_1 = 0.1$ and $p_2 = 0.20$,

$$\left(\frac{27}{48}p_2 + \frac{5}{48}\right) - \left(\frac{1}{2}p_1 + \frac{1}{6}\right) = 0.$$

TABLE 4
The Overall Improved Rate for Algorithms of Matrix-Matrix Addition/Subtraction Based on the $EKMR(n)$ with respect to those of the $TMR(n)$

Matrix-matrix addition/subtraction operation	
Dimension	*Improved Rate (%)*
3-*D*	$\left(\left(\frac{m\delta + r + 1}{m(\delta + 1)} - \frac{1}{3} - \frac{1}{9m^3}\right) \times p + 1 - \frac{m\delta + r + 1}{m(\delta + 1)}\right) \times 100$
4-*D*	$\left(\left(\frac{m\delta + r + 1}{m(\delta + 1)} - \frac{1}{6} - \frac{1}{18m^4}\right) \times p + 1 - \frac{m\delta + r + 1}{m(\delta + 1)}\right) \times 100$
n-*D* ($n \geq 5$)	$\left(\left(\frac{m\delta + r + 1}{m(\delta + 1)} - \frac{4}{n^2 - n}\right) \times p + 1 - \frac{m\delta + r + 1}{m(\delta + 1)}\right) \times 100$

TABLE 5
The Costs of Index Computations of Array Elements and Array Operations for Algorithms of Matrix-Matrix Multiplication Based on the $TMR(n)$ and the $EKMR(n)$

3-*D*			
Schemes \ *Costs*	*Index computations*	*Array operations*	*Total*
TMR	$(9\alpha+6\beta)m^4$	$\alpha m^4+\beta m^4$	$(10\alpha+7\beta)m^4$
EKMR	$(3\alpha+3\beta)m^4$	$4\beta m^4+\alpha m^4+\alpha m^3+\alpha m^2$	$(4\alpha+7\beta)m^4+\alpha m^3+\alpha m^2$
Improved Rate (%)	$(\frac{3}{5}-\frac{1}{10m})\times 100$		
4-*D*			
Schemes \ *Costs*	*Index computations*	*Array operations*	*Total*
TMR	$(18\alpha+9\beta)m^5$	$\alpha m^5+\beta m^5$	$(19\alpha+10\beta)m^5$
EKMR	$(3\alpha+3\beta)m^5$	$4\beta m^5+\alpha m^5+\alpha m^4+2\beta m^3+\alpha m^2+\alpha m$	$(4\alpha+7\beta)m^5+\alpha m^4+2\beta m^3+\alpha m^2+\alpha m$
Improved Rate (%)	$(\frac{15}{19}-\frac{1}{19m})\times 100$		
n-*D* ($n \geq 5$)			
Schemes \ *Costs*	*Index computations*	*Array operations*	*Total*
TMR	$\frac{3n(n-1)}{2}\alpha m^{n+1}+(3n-3)\beta m^{n+1}$	$\alpha m^{n+1}+\beta m^{n+1}$	$\frac{3n^2-3n+2}{2}\alpha m^{n+1}+(3n-2)\beta m^{n+1}$
EKMR	$(6\alpha+6\beta)m^{n+1}$	$4\beta m^{n+1}+\alpha m^{n+1}+\alpha m^n+2\beta m^{n-1}+\alpha m^{n-2}+\alpha m^{n-3}$	$(7\alpha+10\beta)m^{n+1}+\alpha m^n+2\beta m^{n-1}+\alpha m^{n-2}+\alpha m^{n-3}$
Improved Rate (%)	$(1-\frac{14m^4+2m^3}{m^4(3n^2-3n+2)})\times 100$		

TABLE 6
The $LoopCost(l)$ for Algorithms of Matrix-Matrix Multiplication Based on the $TMR(3)$ and the $EKMR(3)$

LoopCost(*l*)					
RefGroup		$C[k][i][j]$	$A[k][i][m]$	$B[k][m][j]$	*Total*
TMR(3)	*K*	$m\times m^3$	$m\times m^3$	$m\times m^3$	$3m^4$
	I	$m\times m^3$	$m\times m^3$	$1\times m^3$	$2m^4+m^3$
	M	$1\times m^3$	$\lceil\frac{m}{r}\rceil\times m^3$	$m\times m^3$	$m^4+\lceil\frac{m}{r}\rceil\times m^3+m^3$
	J	$\lceil\frac{m}{r}\rceil\times m^3$	$1\times m^3$	$\lceil\frac{m}{r}\rceil\times m^3$	$2\lceil\frac{m}{r}\rceil\times m^3+m^3$
RefGroup		$C'[i'][k+r]$	$A'[i'][k+v]$	$B'[j'][k+r]$	*Total*
EKMR(3)		$\lceil\frac{m^2}{r}\rceil\times m^2$	$\lceil\frac{m}{r}\rceil\times m^2$	$\lceil\frac{m^2}{r}\rceil\times m^2$	$2\lceil\frac{m^2}{r}\rceil\times m^2+\lceil\frac{m}{r}\rceil\times m^2$

The overall improved rate is constant. If p_2 is less than 0.20,

$$\left(\frac{27}{48}p_2+\frac{5}{48}\right)-\left(\frac{1}{2}p_1+\frac{1}{6}\right)<0.$$

The overall improved rate decreases. For four-dimensional arrays, we have similar observations as those of three-dimensional arrays.

From Table 4, for n-dimensional arrays where $n \geq 5$, we have two remarks.

Remark 3. If m is divisible by r, the overall improved rate is determined by $\left(1-\frac{4}{n^2-n}\right)\times p\times 100$. For a fixed array dimension n, the overall improved rate increases as the array size m increases (the ratio p increases as the array size m increases). For a fixed array size m, the overall improved rate increases as the array dimension n increases (the ratio p is independent of the array dimension n).

Remark 4. If m is not divisible by r, for a fixed array dimension n, the overall improved rate depends on the array size m and the ratio p. We have similar observations as those of Remark 2. For a fixed array size m, the overall improved rate increases as the array dimension n increases (the ratio p is independent of the array dimension n).

TABLE 7
The $LoopCost(l)$ for Algorithms of Matrix-Matrix Multiplication Based on the $TMR(n)$ and the $EKMR(n)$

LoopCost(l)					
RefGroup		$C[m_{n-4}]\ldots[i][j]$	$A[m_{n-4}]\ldots[i][m_0]$	$B[m_{n-4}]\ldots[m_0][j]$	*Total*
$TMR(n)$	*Others*	m / m^n	m / m^n	$m \backslash m^n$	$3m / m^n$
	I	m / m^n	m / m^n	$1 \backslash m^n$	$2m^{n+1}+m^n$
	M_0	$1 / m^n$	$\left\lceil \frac{m}{r} \right\rceil \times m^n$	$m \backslash m^n$	$m^{n+1}+\left\lceil \frac{m}{r} \right\rceil \times m^n + m^n$
	J	$\left\lceil \frac{m}{r} \right\rceil \times m^n$	$1 \backslash m^n$	$\left\lceil \frac{m}{r} \right\rceil \times m^n$	$2\left\lceil \frac{m}{r} \right\rceil \times m^n + m^n$
RefGroup		$C'_x[w][k+r]$	$A'_x[w][k+v]$	$B'_x[u][k+r]$	*Total*
$EKMR(n)$		$\left\lceil \frac{m^2}{r} \right\rceil \times m^{n-1}$	$\left\lceil \frac{m}{r} \right\rceil \times m^{n-1}$	$\left\lceil \frac{m^2}{r} \right\rceil \times m^{n-1}$	$(2\left\lceil \frac{m^2}{r} \right\rceil + \left\lceil \frac{m}{r} \right\rceil) \times m^{n-1}$

TABLE 8
The Overall Improved Rate for Algorithms of Matrix-Matrix Multiplication Based on the $EKMR(n)$ with Respect to Those of the $TMR(n)$

Matrix-matrix multiplication operation	
Dimension	*Improved Rate (%)*
3-*D*	$((\frac{2m\delta+\delta+2r+2}{m(2\delta+3)} - \frac{2}{5} - \frac{1}{10m}) \times p + 1 - \frac{2m\delta+\delta+2r+2}{m(2\delta+3)}) \times 100$
4-*D*	$((\frac{2m\delta+\delta+2r+2}{m(2\delta+3)} - \frac{4}{19} - \frac{1}{19m}) \times p + 1 - \frac{2m\delta+\delta+2r+2}{m(2\delta+3)}) \times 100$
n-*D* ($n \geq 5$)	$((\frac{2m\delta+\delta+2r+2}{m(2\delta+3)} - \frac{14m^4+2m^3}{m^4(3n^2-3n+2)}) \times p + 1 - \frac{2m\delta+\delta+2r+2}{m(2\delta+3)}) \times 100$

4.3.2 Costs for Matrix-Matrix Multiplication Algorithms

A. The Costs of Addition/Subtraction/Multiplication Operators. Algorithms for matrix-matrix multiplication based on the *TMR*(3) in *KIJM* order and the *EKMR*(3) were described in Section 4.2. Table 5 shows the costs of index computations of array elements and array operations for algorithms of matrix-matrix multiplication based on the *TMR*(*n*) and the *EKMR*(*n*). In Table 5, for the improve rate, we only consider the effect of the α.

From Table 5, we can see that the costs of index computations of array elements and array operations for algorithms of matrix-matrix multiplication based on the $EKMR(n)$ are less than those based on the $TMR(n)$. In Table 5, for $n = 3$ and 4, the improved rate increases as the array size *m* increases. For $n \geq 5$, the improved rate is

$$\left(1 - \frac{14m^4 + 2m^3}{m^4(3n^2 - 3n + 2)}\right) \times 100.$$

The improved rate increases as the array dimension *n* or the array size *m* increases.

B. The Cost of Cache Effect. Table 6 shows the $LoopCost(l)$ for algorithms of matrix-matrix multiplication based on the $EKMR(3)$ and the $TMR(3)$ with various innermost loop indices *K*, *I*, *J*, and *M*. From Table 6, we have similar observations as those of Table 2.

Algorithms for matrix-matrix multiplication based on the $TMR(n)$ in $M_{n-4}M_{n-3}\ldots M_1LKIJM_0$ order and the $EKMR(n)$ were described in Section 4.2. Table 7 shows the $LoopCost(l)$ for algorithms of matrix-matrix multiplication based on the $EKMR(n)$ and the $TMR(n)$ with various innermost loop indices $M_{n-4}, M_{n-3}, \ldots, M_1, L, K, I, J$ and M_0. The improved rate of the $EKMR(n)$ with respect to the $TMR(n)$ whose innermost loop is *j* is

$$\left(1 - \left(2\left\lceil\frac{m^2}{r}\right\rceil + \left\lceil\frac{m}{r}\right\rceil\right) \div \left(\left(2\left\lceil\frac{m}{r}\right\rceil + 1\right) \times m\right)\right) \times 100,$$

for $n \geq 3$. The improved rate is independent of the array dimension *n*. When *m* is divisible by *r*, the improved rate is not 0, which is different from the algorithms of matrix-matrix addition/subtraction operation. Let δ be the quotient of $m \div r$. We have

$$\left(1 - \left(2\left\lceil\frac{m^2}{r}\right\rceil + \left\lceil\frac{m}{r}\right\rceil\right) \div \left(\left(2\left\lceil\frac{m}{r}\right\rceil + 1\right) \times m\right)\right) \times 100$$
$$= \left(1 - \frac{2m\delta + \delta + 2r + 2}{m(2\delta + 3)}\right) \times 100.$$

If *m* is much lager than *r*, the improved rate

$$1 - \frac{2m\delta + \delta + 2r + 2}{m(2\delta + 3)} \approx 0.$$

C. Discussions. From the above analysis, for the $TMR(n)$, the time complexities for the addition/subtraction/multiplication operators and the cache effect are $O(m^{n+1})$ and $O(\lceil\frac{m}{r}\rceil m^n)$, respectively. The time complexity of the addition/subtraction/multiplication operators is larger than that of the cache effect. For a fixed array dimension *n*, the ratio of the cost of addition/subtraction/multiplication

TABLE 9
The Execution Time of Algorithms for the Matrix-Matrix Addition Based on the $TMR(3)$ and the $EKMR(3)$ with/without the Compiler Optimization

Array Sizes \ Methods		TMR(3) KIJ	TMR(3) KJI	TMR(3) IKJ	TMR(3) IJK	TMR(3) JKI	TMR(3) JIK	EKMR(3)
Sun Sparc 20								
10×10×10	*	0.466	0.472	0.461	0.462	0.461	0.461	0.368
	**	0.215	0.205	0.119	0.112	0.118	0.117	0.098
50×50×50	*	69.204	71.615	70.569	92.227	79.027	102.288	53.837
	**	22.508	23.319	27.609	53.324	42.451	67.800	19.731
100×100×100	*	545.187	599.843	578.356	1121.259	971.615	1450.361	424.348
	**	181.324	206.936	203.647	1052.891	884.855	1322.829	158.529
150×150×150	*	4036.25	4090.10	4412.34	9468.14	6647.67	11510.03	2990.13
	**	603.23	659.88	649.45	6002.90	3497.50	8138.36	534.59
200×200×200	*	10683.69	11784.38	12234.63	26428.42	16092.11	25828.06	7541.69
	**	1534.27	3829.70	1491.12	18701.37	8971.80	18772.69	1386.43
Intel Pentium III 800 *PC*								
10×10×10	*	0.054	0.053	0.052	0.053	0.052	0.052	0.042
	**	0.010	0.009	0.009	0.009	0.009	0.008	0.005
50×50×50	*	11.451	11.508	11.354	13.908	13.435	15.691	8.839
	**	4.421	4.369	4.349	5.309	6.627	7.548	4.069
100×100×100	*	76.533	76.637	76.296	92.875	307.694	326.276	58.598
	**	31.192	30.087	33.986	51.126	175.111	226.006	27.216
150×150×150	*	329.26	542.72	331.92	724.68	1701.94	2052.30	244.00
	**	110.50	130.81	138.61	166.83	648.36	822.62	108.90
200×200×200	*	845.13	1313.68	869.05	1474.11	4034.10	4422.22	625.42
	**	284.58	277.19	271.70	406.38	1588.26	1927.11	260.81
IBM RS/6000								
10×10×10	*	0.171	0.171	0.169	0.169	0.169	0.169	0.133
	**	0.026	0.029	0.024	0.025	0.025	0.025	0.023
50×50×50	*	21.865	23.016	22.562	22.131	65.511	66.258	17.479
	**	3.961	3.546	2.957	2.986	11.461	8.292	2.740
100×100×100	*	176.375	179.129	179.125	178.068	629.211	667.122	144.563
	**	24.524	28.625	28.593	44.106	292.602	318.651	21.850
150×150×150	*	593.05	606.71	611.17	1053.14	2269.52	2738.98	484.31
	**	75.97	114.35	135.58	224.96	1173.06	1808.66	65.77
200×200×200	*	1411.77	4390.57	1465.41	8361.49	5169.73	8659.84	1141.11
	**	515.39	902.24	622.84	1011.63	3179.72	4510.17	423.40

*: Without the compiler optimization
**: With the compiler optimization
Time: ms

operators increases as the array size m increases. For a fixed array size m, the ratio of the cost of addition/subtraction/multiplication operators is independent of the array dimension n. The overall improved rate for algorithms of matrix-matrix multiplication based on the $EKMR(n)$ with respect to those of the $TMR(n)$ is given in Table 8. From Table 8, for three- and four-dimensional arrays, we have the following remark:

Remark 5. The overall improved rates for three- and four-dimensional arrays depend on the array size m and the ratio p. We have similar observations as those of Remark 2.

From Table 8, for n-dimensional arrays where $n \geq 5$, we have two remarks.

Remark 6. For a fixed array dimension n, the overall improved rate depends on the array size m and the ratio p. We have similar observations asthose of Remark 2.

Remark 7. For a fixed array size m, the overall improved rate increases as the array dimension n increases (the ratio p is independent of the array dimension n).

5 Experimental Results

To evaluate the performance of algorithms for matrix-matrix addition/subtraction and matrix-matrix multiplication array operations, we have implemented those algorithms on three platforms, an IBM RS/6000 with 256MB main memory, a Sun Sparc 20 with 180MB main memory, and an Intel Pentium III 800 PC with 512MB main memory. The algorithms were implemented in C. For the Sun Sparc 20 and Intel Pentium III 800 PC platforms, all C programs were compiled by gcc compilers with/without the -O3 option. For the IBM RS/6000 platform, we used the cc compiler to compile all C programs with/without the -O4 option. The array size is set from $10 \times 10 \times 10$ to $200 \times 200 \times 200$ for the three-dimensional array and from $10 \times 10 \times 10 \times 10$ to $50 \times 50 \times 50 \times 50$ for the four-dimensional array. Since Fortran 90 provides a rich set of intrinsic functions for multidimensional array operations, in the experimental test, we also compare the performance of intrinsic functions provided by the Fortran 90 compiler and those based on the *EKMR* scheme on an IBM RS/6000.

TABLE 10
The Execution Time of Algorithms for the Matrix-Matrix Multiplication Based on the $TMR(3)$ and the $EKMR(3)$ with/without the Compiler Optimization

Methods / Array Sizes		TMR(3)							EKMR(3)
		KIJM	KIMJ	KMIJ	IKMJ	IMKJ	MKIJ	MIKJ	
Sun Sparc 20									
10×10×10	*	0.0087	0.0089	0.0090	0.0087	0.0090	0.0090	0.0090	0.0080
	**	0.0054	0.0052	0.0059	0.0052	0.0051	0.0052	0.0052	0.0050
50×50×50	*	6.115	6.130	6.215	6.215	6.248	6.194	6.270	5.232
	**	3.31	3.33	3.22	3.35	3.93	3.69	4.20	3.07
100×100×100	*	98.50	102.09	99.15	100.05	100.44	99.34	100.18	81.68
	**	57.88	53.85	61.24	53.90	60.89	58.95	64.98	51.20
150×150×150	*	1111.0	1108.0	1116.5	1107.6	1138.0	1116.1	1139.3	922.1
	**	286.1	275.3	277.6	274.8	314.7	277.5	314.9	257.1
200×200×200	*	3385.8	3381.5	3393.5	3380.5	3443.0	3392.1	3443.6	2703.8
	**	1154.5	1081.4	1105.4	1089.4	1209.7	1102.8	1219.4	958.6
Intel Pentium III 800 PC									
10×10×10	*	0.0008	0.0008	0.0008	0.0008	0.0008	0.0008	0.0008	0.0006
	**	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
50×50×50	*	0.658	0.649	0.658	0.678	0.684	0.717	0.722	0.526
	**	0.072	0.057	0.057	0.152	0.155	0.203	0.148	0.045
100×100×100	*	10.33	10.35	10.30	11.19	11.26	11.44	11.46	8.29
	**	1.091	0.849	0.841	2.358	2.435	3.116	3.058	0.752
150×150×150	*	53.96	54.18	53.65	62.45	63.79	63.71	65.07	43.22
	**	4.610	4.148	4.137	11.993	13.320	16.347	16.141	3.516
200×200×200	*	158.9	159.6	158.5	185.5	204.3	190.9	204.0	126.4
	**	16.989	14.735	12.899	37.170	58.092	49.992	60.319	11.654
IBM RS/6000									
10×10×10	*	0.0021	0.0021	0.0025	0.0023	0.0021	0.0021	0.0021	0.0019
	**	0.0008	0.0003	0.0003	0.0003	0.0004	0.0003	0.0003	0.0003
50×50×50	*	1.569	1.567	1.560	1.542	1.518	1.533	1.501	1.316
	**	0.507	0.316	0.315	0.332	0.333	0.350	0.353	0.296
100×100×100	*	23.467	23.495	23.607	23.902	24.895	23.887	24.935	20.880
	**	7.744	3.555	3.876	3.953	5.388	3.956	5.316	2.744
150×150×150	*	120.51	119.53	119.60	121.02	125.68	121.13	125.65	105.11
	**	35.300	17.765	18.859	19.183	27.011	20.581	26.615	15.813
200×200×200	*	383.64	380.61	381.30	382.62	393.75	383.10	393.81	335.79
	**	143.25	70.46	73.73	75.75	105.04	80.31	102.41	62.25

*: Without the compiler optimization Time: s
**: With the compiler optimization

5.1 Performance Comparisons of Array Operations Based on the $EKMR(3)$ and the $TMR(3)$

Table 9 shows the execution time of algorithms for the matrix-matrix addition based on the $EKMR(3)$ and the $TMR(3)$. From Table 9, we can see that the execution time of the algorithm based on the $EKMR(3)$ is less than that based on the$TMR(3)$ for all test samples with/without the compiler optimization. In the following discussion, we only consider the cases without the compiler optimization.

For the $TMR(3)$, we can see that the execution time of algorithms whose innermost loop index is *J* (KIJ and IKJ orders) is less than that of algorithms whose innermost loop index is *K* or *I*. These results match the theoretical analysis described in Section 4.3.1. In general, the cache line size is a multiple of 4, such as $4, 8, \ldots, 4n$. From Table 9, we can see that the overall improved rate of the array size $200 \times 200 \times 200$ is larger than that of the array size $100 \times 100 \times 100$ and this result matches Remark 1. When the array size increases, the overall improved rate for the Sun Sparc 20 increases, the overall improved rate for the Intel PentiumIII 800PC is constant, and the overall improved rate for the IBM RS/6000 decreases. Although it is very difficult to obtain the ratios of the cost of addition/subtraction/multiplication operators and the cost of the cache effect to the overall cost of an algorithm, for this phenomenon, the possible reason was described in Remark 2.

For the matrix-matrix multiplication, based on the $TMR(3)$, there are 24 loop orders. From the theoretical analysis, we have shown that algorithms whose innermost loop index is *J* have the best performance. Therefore, in Table 10, for the $TMR(3)$, we show the execution time of algorithms whose innermost loop index is *J*. For other innermost loop indices, we only show the one that has the smallest execution time (without the compiler optimization). From Table 10, we can see that the execution time of algorithm based on the $EKMR(3)$ is less than that based on the $TMR(3)$ for all test samples with/without the compiler optimization. In the following discussion, we only consider the cases without the compiler optimization.

For the $TMR(3)$, in general, the execution time of algorithms whose innermost loop index is *J* is less than that of algorithms whose innermost loop index is not *J*. These results match the theoretical analysis described in Section 4.3.2. For the overall improved rates, we have similar observations as those of Table 9. These observations match Remark 5. However, for the $TMR(3)$, there are some exceptions. For example, for Sun Sparc 20, the execution

TABLE 11
The Execution Time of Algorithms for the Matrix-Matrix Addition Based on the $TMR(4)$ and the $EKMR(4)$ with/without the Compiler Optimization

Array sizes \ Methods		TMR(4) (LKIJ)	EKMR(4)
Sun Sparc 20			
10×10×10×10	*	6.456	3.921
	**	2.256	1.996
20×20×20×20	*	105.239	65.668
	**	29.490	26.744
30×30×30×30	*	533.083	292.330
	**	146.927	129.612
40×40×40×40	*	3422.274	1862.289
	**	507.067	444.144
50×50×50×50	*	9803.730	5197.671
	**	1137.829	990.796
Intel Pentium III 800 *PC*			
10×10×10×10	*	0.713	0.448
	**	0.143	0.126
20×20×20×20	*	13.665	8.083
	**	5.399	5.087
30×30×30×30	*	78.563	45.678
	**	24.778	21.852
40×40×40×40	*	260.543	151.468
	**	84.600	81.882
50×50×50×50	*	815.700	484.779
	**	224.051	200.487
IBM RS/6000			
10×10×10×10	*	2.105	1.368
	**	0.345	0.314
20×20×20×20	*	35.444	23.681
	**	22.569	19.717
30×30×30×30	*	172.904	117.966
	**	18.112	16.766
40×40×40×40	*	552.005	365.832
	**	61.926	52.609
50×50×50×50	*	1338.657	897.834
	**	156.365	147.727

*: Without the compiler optimization Time: s
**: With the compiler optimization

time of algorithms whose innermost loop index is *J* is larger than that of the algorithm in $KIJM$ order for the case where the array size is $10 \times 10 \times 10$ or $100 \times 100 \times 100$. The reason is that algorithm *LoopCost* assumes that there will be no cache conflict problem in algorithms [4], [28]. In practice, the cache conflict may be encountered in algorithms and will influence the overall performance of algorithms.

5.2 Performance Comparisons of Array Operations Based on the $EKMR(4)$ and the $TMR(4)$

For the matrix-matrix addition/subtraction, based on the $TMR(4)$, there are 24 loop orders. In Table 11, we only show the one that has the smallest execution time for the $TMR(4)$. From Table 11, we can see that the execution time of algorithm based on the $EKMR(4)$ is less than that based on the $TMR(4)$ for all test samples with/without the compiler optimization. In the following discussion, we only consider the cases without the compiler optimization.

For the $TMR(4)$, we can see that the algorithm whose innermost loop index is *J* has the smallest execution time. These results match the theoretical analysis described in Section 4.3.1. In addition, we can see that the overall improved rate of the array size $20 \times 20 \times 20$ is larger than that of the array size $40 \times 40 \times 40$. This result matches Remark 3. For the overall improved rates, we have similar observations as those of Table 9. These results match Remark 4. From Table 9 and Table 11, we can see that the overall improved rates for four-dimensional arrays are better than those for three-dimensional arrays. These results match Remarks 3 and 4.

For the matrix-matrix multiplication, based on the $TMR(4)$, there are 120 loop orders. From the theoretical analysis, we have shown that algorithms whose innermost loop index is *J* have the best performance. Therefore, in Table 12, for the $TMR(4)$, we show the execution time of some algorithms whose innermost loop index is *J*. For other innermost loop indices, we only show the one that has the smallest execution time (without the compiler optimization). From Table 12, we can see that the execution time of algorithm based on the $EKMR(4)$ is less than that based on the $TMR(4)$ for all test samples with/without the compiler optimization. In the following discussion, we only consider the cases without the compiler optimization.

For the $TMR(4)$, we can see that the algorithm whose innermost loop index is *J* has the smallest execution time. These results match the theoretical analysis described in

TABLE 12
The Execution Time of Algorithms for the Matrix-Matrix Multiplication Based on the $TMR(4)$ and the $EKMR(4)$ with/without the Compiler Optimization

Array Sizes \ Methods		TMR(4)					EKMR(4)
		LKIJM	MLKIJ	LMKIJ	LKMIJ	LKIMJ	
Sun Sparc 20							
10×10×10×10	*	0.1263	0.1529	0.1458	0.1107	0.1139	0.0852
	**	0.0632	0.0665	0.0612	0.0627	0.0753	0.0514
20×20×20×20	*	3.515	3.621	3.654	3.511	3.599	2.359
	**	1.976	2.072	2.053	1.922	1.920	1.784
30×30×30×30	*	28.578	29.008	29.478	28.528	29.508	18.994
	**	14.556	15.728	15.610	13.327	13.093	11.296
40×40×40×40	*	243.65	247.52	247.36	242.85	242.627	159.193
	**	63.246	65.444	65.505	62.703	61.275	58.156
50×50×50×50	*	868.64	850.77	850.44	838.91	836.93	562.00
	**	233.65	206.54	206.30	175.73	174.62	164.63
Intel Pentium III 800 *PC*							
10×10×10×10	*	0.0116	0.0115	0.0115	0.0114	0.0114	0.0075
	**	0.0012	0.0015	0.0014	0.0015	0.0014	0.0011
20×20×20×20	*	0.378	0.466	0.381	0.381	0.377	0.240
	**	0.044	0.117	0.045	0.044	0.045	0.043
30×30×30×30	*	3.340	3.686	3.342	3.311	3.334	2.072
	**	0.261	0.912	0.350	0.341	0.325	0.244
40×40×40×40	*	11.584	14.274	14.227	11.495	11.585	7.301
	**	1.263	3.406	3.256	1.231	1.238	1.058
50×50×50×50	*	50.066	59.050	58.592	50.513	49.988	30.112
	**	3.594	11.083	11.086	3.687	3.740	3.296
IBM RS/6000							
10×10×10×10	*	0.0275	0.0288	0.0282	0.0277	0.0277	0.0232
	**	0.0046	0.0036	0.0035	0.0035	0.0034	0.0030
20×20×20×20	*	0.914	0.986	0.900	0.901	0.901	0.723
	**	0.139	0.139	0.116	0.117	0.115	0.109
30×30×30×30	*	6.948	7.105	7.001	6.934	7.041	5.538
	**	1.959	1.256	1.396	1.199	1.236	1.060
40×40×40×40	*	28.674	29.138	31.005	28.762	28.739	22.489
	**	8.034	5.189	5.549	4.714	5.712	4.297
50×50×50×50	*	87.427	88.787	88.878	88.376	88.115	69.092
	**	24.675	18.606	17.980	16.217	16.769	14.629

*: Without the compiler optimization Time: s
**: With the compiler optimization

Section 4.3.2. For the overall improved rates, we have similar observations as those of Table 9. These results match Remark 6. From Table 10 and Table 12, we can see that the overall improved rates for four-dimensional arrays are better than those for three-dimensional arrays. These results match Remark 7.

5.3 Performance Comparisons of Fortran 90 Array Intrinsic Functions

Fortran 90 [1] provides a rich set of array intrinsic functions, which operate on elements of multidimensional array objects. These array intrinsic functions are useful in a large number of scientific codes. In general, they can be divided into two categories. In the first category, the array intrinsic functions, such as *ALL*, *MAXVAL*, *PACK*, *SUM*, etc., focus on the operations in an array. They are usually used to find the maximum or minimum value, do logic operations, and collect some array elements in an array.

In the second category, the array intrinsic functions, such as +, -, *MERGE*, etc., focus on element-to-element operations between two arrays. They are usually used to perform matrix-matrix addition/subtraction, matrix-matrix multiplication, etc. Fortran 90 adopts the column-major data layout to store array elements based on the *TMR* scheme. To implement these array intrinsic functions based on the *EKMR* scheme, the *EKRM* scheme presented in Section 3 needs a slightly modifications. For the $EMKR(3)$, the index variable i' is a combination of the index variables i and k and the index variable j' is the same as index variable j.

For the $EMKR(4)$, the index variable i' is a combination of the index variables i and k and the index variable j' is a combination of the index variables l and j. Based on the modified *EKMR* scheme, we design algorithms for Fortran 90 array intrinsic functions, including *ALL*, *MAXVAL*, *MERGE*, *PACK*, *SUM*, and +.

To evaluate the performance of these algorithms based on the *EKMR* scheme, we implemented these algorithms in Fortran 90, executed them on an IBM RS/6000 machine, and compared the execution time of these algorithms with those provided by the Fortran 90 compiler. Table 13 shows the execution time of the array intrinsic functions provided by the Fortran 90 compiler and based on the $EKMR(3)$ with different array size. From Table 13, we can see that the execution time of algorithms based on the $EKMR(3)$ is less than that provided by the Fortran 90 compiler for all test intrinsic functions. Table 14 shows the execution time of the array intrinsic functions provided by the Fortran 90 compiler and based on the $EKMR(4)$ with different array size. From Table 14, we have similar observation as that of Table 13.

TABLE 13
The Execution Time of the Array Intrinsic Functions Provided by the Fortran 90 Compiler and Based on the $EKMR(3)$ with Different Array Size on IBM *RS/*6000 Machine

Array Intrinsic Functions	*Methods* \ *Array Sizes*	50/50/50	100^100\100	200\200/200
+	*TMR*(3)	24	509	5075
(*C*=*A*+*B*)	*EKMR*(3)	18	397	4615
ALL	*TMR*(3)	20	156	3893
(*ALL*(*A*>0))	*EKMR*(3)	16	147	3521
MAXVAL	*TMR*(3)	22	171	3957
(*b*=*MAXVAL*(*A*))	*EKMR*(3)	16	154	3621
MERGE	*TMR*(3)	21	350	3052
(*c*=*MERGE*(*A*,*B*,*A*>*B*))	*EKMR*(3)	16	150	2790
PACK	*TMR*(3)	22	149	2428
(*c*=*PACK*(*A*,*A*>3))	*EKMR*(3)	14	105	2256
SUM	*TMR*(3)	22	169	2523
(*b*=*SUM*(*A*))	*EKMR*(3)	17	148	2367

Time: ms

TABLE 14
The Execution Time of the Array Intrinsic Functions Provided by the Fortran 90 Compiler and Based on the $EKMR(4)$ with Different Array Size on IBM *RS/*6000 Machine

Array Intrinsic Functions	*Methods* \ *Array Sizes*	10/10/10/10	30/30\30\30	50\50/50/50
+	*TMR*(3)	4	192	1458
(*C*=*A*+*B*)	*EKMR*(3)	2	130	1014
ALL	*TMR*(3)	4	171	1215
(*ALL*(*A*>0))	*EKMR*(3)	2	115	868
MAXVAL	*TMR*(3)	4	159	1137
(*b*=*MAXVAL*(*A*))	*EKMR*(3)	1	114	817
MERGE	*TMR*(3)	5	169	1536
(*c*=*MERGE*(*A*,*B*,*A*>*B*))	*EKMR*(3)	2	114	1262
PACK	*TMR*(3)	4	156	2451
(*c*=*PACK*(*A*,*A*>3))	*EKMR*(3)	1	90	1389
SUM	*TMR*(3)	4	148	2307
(*b*=*SUM*(*A*))	*EKMR*(3)	1	107	1603

Time: ms

6 Conclusions and Future Work

In this paper, we have proposed a new scheme, *EKMR*, for the multidimensional array representation. The main idea of the *EKMR* scheme is to represent a multidimensional array by a set of two-dimensional arrays. To evaluate the proposed scheme, we designed efficient algorithms for multidimensional array operations, matrix-matrix addition/subtraction, and matrix-matrix multiplications, based on the *EKMR* and *TMR* schemes. Both theoretical analysis and experimental test for these array operations were conducted. From the theoretical analysis and experimental results, we can see that array operations based on the *EKMR* scheme outperform those based on the *TMR* scheme. The reasons are two-fold. First, the *EKMR* scheme can decrease the costs of index computations of array elements for array operations because it uses a set of two-dimensional arrays to represent a higher dimensional array. Second, the cache miss rate for array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme because the number of cache lines accessed by array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme. Since Fortran 90 provides a rich set of intrinsic functions for multidimensional array operations, in the experimental test, we also compared the performance of intrinsic functions provided by the Fortran 90 compiler and those based on the *EKMR* scheme. The experimental results showed that algorithms based on the *EKMR* scheme outperform those based on the *TMR* scheme and those provided by the Fortran 90 compiler.

In the future, we plan to work on the following directions:

1. Develop efficient parallel algorithms of array operations based on the *EKMR* scheme. Some preliminary results can be found in [27].
2. Develop compression schemes for sparse arrays in the form of the *EKMR* scheme on sequential and multiprocessor machines.
3. Apply recursive data layout functions to the *EKMR* scheme to obtain other efficient data layouts for array operations.
4. Develop efficient algorithms of array operations based on the *EKMR* scheme by using the tiling technique.

We believe that these directions are of importance in array operations.

ACKNOWLEDGMENTS

The work in this paper was partially supported by National Science Council of the Republic of China under contract NSC89-2213-E-035-007.

REFERENCES

[1] J.C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith, and J.L. Wagener, *Fortran 90 Handbook.* Intertext Publications/McGraw-Hill Inc. 1992.

[2] I. Banicescu and S.F. Hummel, "Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations," *Proc. 1995 ACM/IEEE Supercomputing Conf.,* Dec. 1995.

[3] D. Callahan, S. Carr, and K. Kennedy, "Improving Register Allocation for Subscripted Variables," *Proc. ACM SIGPLAN 1990 Conf. Programming Language Design and Implementation,* pp. 53-65, June 1990.

[4] S. Carr, K.S. McKinley, and C.-W. Tseng, "Compiler Optimizations for Improving Data Locality," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 252-262, Oct. 1994.

[5] L. Carter, J. Ferrante, and S.F. Hummel, "Hierarchical Tiling for Improved Superscalar Performance," *Proc. Nineth Int'l Symp. Parallel Processing,* pp. 239-245, Apr. 1995.

[6] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi, "Recursive Array Layouts and Fast Parallel Matrix Multiplication," *Proc. Eleventh Ann. ACM Symp. Parallel Algorithms and Architectures,* pp. 222-231, June 1999.

[7] S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems," *Proc. 1999 ACM Int'l Conf. Supercomputing,* pp. 444-453, June 1999.

[8] M. Cierniak and W. Li, "Unifying Data and Control Transformations for Distributed Shared Memory Machines," Technical Report TR-542, Dept. of Computer Science, Univ. of Rochester, Nov. 1994.

[9] S. Coleman and K.S. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. ACM SIGPLAN '95 Conf. Programming Language Design and Implementation,* pp. 279-290, June 1995.

[10] J.K. Cullum and R.A. Willoughby, *Algorithms for Large Symmetric Eignenvalue Computations,* vol. 1. Boston, Mass.: Birkhauser, 1985.

[11] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Cache Misses Prediction for High Performance Sparse Algorithms," *Proc. Fourth Int'l Euro-Par Conf. (Euro-Par '98),* pp. 224-233, Sept. 1998.

[12] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Cache Probabilistic Modeling for Basic Sparse Algebra Kernels Involving Matrices with a Non-Uniform Distribution," *Proc. 24th IEEE Euromicro Conf.,* pp. 345-348, Aug. 1998.

[13] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Modeling Set Associative Caches Behaviour for Irregular Computations," *ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '98),* pp. 192-201, June 1998.

[14] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Automatic Analytical Modeling for the Estimation of Cache Misses," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '99),* Oct. 1999.

[15] J.D. Frens and D.S. Wise, "Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code," *Proc. Sixth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* June 1997.

[16] G.H. Golub and C.F. Van Loan, *Matrix Computations, Second ed.* Baltimore, Md: Johns Hopkins Univ. Press, 1989.

[17] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving Cache Locality by a Combination of Loop and Data Transformations," *IEEE Trans. Computers,* Feb. 1999.

[18] M. Kandemir, J. Ramanujam, and A. Choudhary, "A Compiler Algorithm for Optimizing Locality in Loop Nests," *Proc. 1997 ACM Int'l Conf. Supercomputing,* pp. 269-276, July 1997.

[19] C.W. Kebler and C.H. Smith, "The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations," *Proc. Seventh Int'l Workshop Program Comprehension,* pp. 200-207, 1999.

[20] V. Kotlyar, K. Pingali, and P. Stodghill, "Compiling Parallel Sparse Code for User-Defined Data Structures," *Proc. Eighth SIAM Conf. Parallel Processing for Scientific Computing,* Mar. 1997.

[21] V. Kotlyar, K. Pingali, and P. Stodghill, "A Relation Approach to the Compilation of Sparse Matrix Programs," *Euro Par,* Aug. 1997.

[22] V. Kotlyar, K. Pingali, and P. Stodghill, "Compiling Parallel Code for Sparse Matrix Applications," *Proc. Supercomputing Conf.,* Aug. 1997.

[23] B. Kumar, C.-H. Huang, R.W. Johnson, and P. Sadayappan, "A Tensor Product Formulation of Strassen's Matrix Multiplication Algorithm with Memory Reduction," *Proc. Seventh Int'l Parallel Processing Symp.,* pp. 582-588, Apr. 1993.

[24] M.S. Lam, E.E. Rothberg, and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 63-74, Apr. 1991.

[25] W. Li and K. Pingali, "A Singular Loop Transformation Framework Based on Non-Singular Matrices," *Proc. Fifth Workshop Languages and Compilers for Parallel Computers,* pp. 249-260, 1992.

[26] J.-S. Liu, J.-Y. Lin, and Y.-C. Chung, "Efficient Representation for Multi-Dimensional Matrix Operations," *Proc. Workshop Compiler Techniques for High Performance Computing (CTHPC),* pp. 133-142, Mar. 2000.

[27] J.-S. Liu, J.-Y. Lin, and Y.-C. Chung, "Efficient Parallel Algorithms for Multi-Dimensional Matrix Operations," *Proc. IEEE Int'l Symp. Parallel Architectures, Algorithms and Networks (I-SPAN),* pp.224-229, Dec. 2000.

[28] K.S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Lanaguages and Systems,* July 1996.

[29] M.F.P. O'Boyle and P.M.W. Knijnenburg, "Integrating Loop and Data Transformations for Global Optimization," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '98),* pp. 12-19, Oct. 1998.

[30] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing.* Cambridge Univ. Press, 1996.

[31] P.D. Sulatycke and K. Ghose, "Caching Efficient Multithreaded Fast Multiplication of Sparse Matrices," *Proc. First Merged Int'l Parallel Processing Symp. and Symp. Parallel and Distributed Processing,* pp. 117-123, 1998.

[32] M. Thottethodi, S. Chatterjee, and A.R. Lebeck, "Turing Strassen's Matrix Multiplication for Memory Efficiency," *Proc. ACM/IEEE SC98 Conf. High Performance Networking and Computing,* Nov. 1998.

[33] M. Ujaldon, E.L. Zapata, S.D. Sharma, and J. Saltz, "Parallelization Techniques for Sparse Matrix Applications," *J. Parallel and Distribution Computing,* 1996.

[34] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation,* pp. 30-44, June 1991.

[35] L.H. Ziantz, C.C. Ozturan, and B.K. Szymanski, "Run-Time Optimization of Sparse Matrix-Vector Multiplication on SIMD Machines," *Proc. Int'l Conf. Parallel Architectures and Languages,* pp. 313-322, July 1994.

Chun-Yuan Lin received the BS degree in computer science from Feng Chia University in 1999 and the MS degree in computer science from Feng Chia University in 2000,respectively. He is currently a PhD student in the Department of Information Engineering at Feng Chia University. His research interests are in the areas of parallel and distributed computing, parallel algorithms, high performance compilers for data parallel programming languages, and array operations.

Jen-Shiuh Liu received the BS and MS degrees in nuclear engineering from National Tsing Hua University and the MS and PhD degrees in computer science from Michigan State University in 1979, 1981, 1987 and 1992, respectively. Since 1992, he has been an associate professor in the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan. His research interests include parallel and distributed processing, computer networks, and computer system security.

Yeh-Ching Chung received the BS degree in computer science from Chung Yuan Christian University in 1983, and the MS and PhD degrees in computer and information science from Syracuse University in 1988 and 1992, respectively. Currently, he is a professor in the Department of Information Engineering at Feng Chia University, where he directs the Parallel and Distributed Processing Laboratory. His research interests include parallel compilers, parallel programming tools, mapping, scheduling, load balancing, embedded systems and virtual reality. He is a member of the IEEE Computer Society.