Efficient Parallel Algorithms for Multi-Dimensional Matrix Operations[1]

Jen-Shiuh Liu, Jiun-Yuan Lin, and Yeh-Ching Chung

Department of Information Engineering

Feng Chia University, Taichung, Taiwan 407, ROC

Tel: 886-4-4517250 x3700

Fax: 886-4-4515517

Email: {liuj, jylin, ychung} @iecs.fcu.edu.tw

Abstract

Matrix operations are the core of many linear systems. Efficient matrix multiplication is critical to many numerical applications, such as climate modeling, molecular dynamics, computational fluid dynamics and etc. Much research work has been done to improve the performance of matrix operations. However, the majority of these works is focused on two-dimensional (2D) matrix. Very little research work has been done on three or higher dimensional matrix. Recently, a new structure called Extended Karnaugh Map Representation (EKMR) for n-dimensional (nD) matrix representation has been proposed, which provides better matrix operations performance compared to the Traditional matrix representation (TMR). The main idea of EKMR is to represent any nD matrix by 2D matrices. Hence, efficient algorithms design for nD matrices becomes less complicated. Parallel matrix operation algorithms based on EKMR and TMR are presented. Analysis and experiments are conducted to assess their performance. Both our analysis and experimental result show that parallel algorithms based on EKMR outperform those based on TMR.

Keywords: Parallel algorithm, Compiler, Matrix operation, Multi-dimensional matrix, Data Structure.

1. Introduction

Matrix operations are the core of many linear systems. Efficient matrix multiplication is critical to many numerical applications [4, 9, 16]. A significant part of scientific codes consists of matrix computation, which results in poor efficiency on today's supercomputers. Much research work has been done to improve the performance of matrix operations [1-3, 5-8, 14, 11-13].

However, the majority of these works is focused on two-dimensional (2D) matrix [1-3, 17]. Very little research work, for example [10] has been done on three or higher dimensional matrix. Fortran 90 provides a rich set of array intrinsic functions, which operate on elements of multi-dimensional array objects concurrently. Hence, efficient matrix operations become an important issue.

A multi-dimensional matrix can be viewed as a collection of 2D matrices. Hence, 2D matrices are used as building blocks for multi-dimensional matrices. For example, we can use 5 separate 4×3 2D matrices to represent a 3D matrix of size 5×4×3. This scheme is called *Traditional Matrix Representation* (*TMR*(*n*)), where *n* is the dimension of the matrix. Researchers have proposed many algorithms or techniques [1-3, 18] in order to gain better performance for *TMR*(2). For example, by applying re-permutation concept [2, 18] to reorder the sequence of certain operations, we can obtain better performance. Similarly, by using compression schemes in sparse matrix operations, we can reduce storage requirement substantially. However, these proposed algorithms or techniques for 2D matrix usually do not perform well when applied to higher dimensional matrix. The main reason is that the size of the matrix represented by *TMR*(*n*) increases exponentially as the dimension increases. Hence, multi-dimensional matrices represented by *TMR*(*n*) become less manageable and difficult for programmers to design efficient algorithms for matrix operations.

A new structure called *Extended Karnaugh Map*

1.The work of this paper was partially supported by NSC under contract NSC89-2213-E-035-007.

Representation (*EKMR*(*n*)) for *n*D matrix representation was introduced recently [15], which provides better matrix operations performance compared to the *TMR*(*n*). The main idea of *EKMR*(*n*) is to represent any *n*D matrix by a set of 2D matrices. This structure is suitable for dense or sparse matrix without using compress scheme. Hence, efficient algorithms design for *n*D matrices becomes less complicated. In this work, we present parallel algorithms for matrix operations in both *EKMR* and *TMR*. We compare the performance of matrix-matrix addition/subtraction, matrix-vector multiplication, and matrix-matrix multiplication in both *EKMR* and *TMR*. Our experimental results show that operations in the form of *EKMR* outperform that in the form of *TMR* for most cases.

The remainder of paper is organized as follows. In Section 2, we briefly review the *EKMR* scheme. Section 3 presents the parallel algorithms for matrix-matrix addition/subtraction, matrix-vector multiplication, and matrix-matrix multiplication operations in *EKMR*. Analytic and experimental results are given in Section 4. Finally, conclusions and future work is presented in Section 5.

2. Multi-dimensional Matrix Representation

The *EKMR* was first proposed in [15]. Here, we give a brief overview. The Karnaugh Map technique is thought to be the most efficient tool to deal with Boolean functions. It provides instant recognition of basic patterns, can be used to represent all possible combination minimal terms. Figure 1 displays some examples of Karnaugh Map. It is clear that *n*-input Karnaugh Map uses *n* variables to reserve memory storage and represent all the 2^n possible combinations. Furthermore, any *n*-input Karnaugh Map can be drawn on a plane easily, where $n \leq 4$.

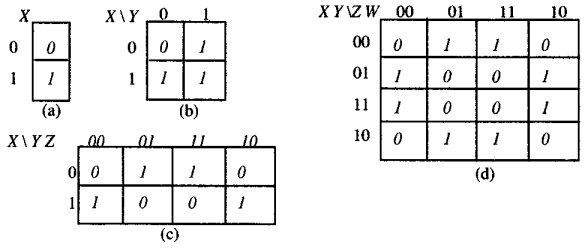


Figure 1: Karnaugh Map: (a) 1-input for $f=X$. (b) 2-input for $f=X+Y$. (c) 3-input for $f=XZ'+X'Z$. (d) 4-input for $f=YW'+Y'W$.

We use the concept of Karnaugh Map to propose our new matrix representation in *EKMR*. Since *EKMR*(1) is simply a 1D array, no new definition is needed. Similarly, *EKMR*(2) is the traditional 2D matrix. Therefore, *EKMR*(*n*) has the same representation as *TMR*(*n*) for n=1,2. We now consider *EKMR*(3) and *EKMR*(4) for these are basic building blocks of *EKMR*(*n*).

We use an example to illustrate the structure of *EKMR*(3). Let *A*[*k*][*i*][*j*] denote a 3D matrix in *TMR*(3). Figure 2 displays two ways to view a 3D matrix with a size of 3×4×5. The corresponding *EKMR*, denoted by *A*[*i*'][*j*'], is shown in Figure 3, where it is represented by a 2D matrix with the size of 4×(3×5). The basic difference between *TMR*(3) and *EKMR*(3) is the placement of elements along the direction indexed by *k*. In *EKMR*(3), we use index variable *i*' to indicate the row direction and *j*' to indicate the column direction. Notice that index *i*' is the same as *i*, whereas *j*' is a combination of *j* and *k*. The analogy between *EKMR*(3) and Karnaugh Map with three-input is as follows(cf. Figure 3 and Figure 1(c)): index variables *i*, *j* and *k* corresponds to variable *X*, *Y* and *Z*, respectively.

The way to obtain *EKMR*(4) from *EKMR*(3) is similar to obtain a four-input Kaunaugh map from a three-input one. Figure 4 illustrates that a (2×4)×(3×5) matrix in *EKMR*(4) can be used to represent a 2×3×4×5 matrix in *TMR*(4). The structure of *EKMR*(n) can be found in [15].

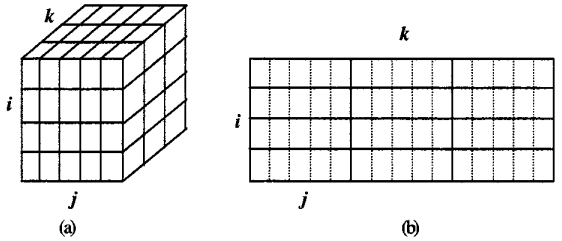


Figure 2: A 3×4×5 matrix in *TMR*(3): (a) A 3-D view, (b) A 2-D view.

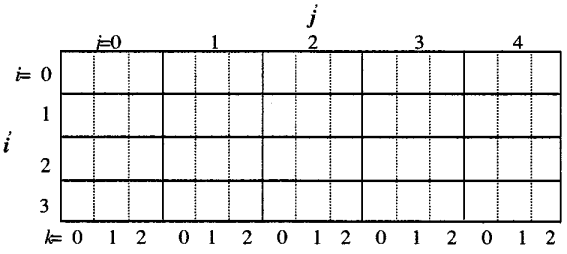


Figure 3: The structure of *EKMR*(3).

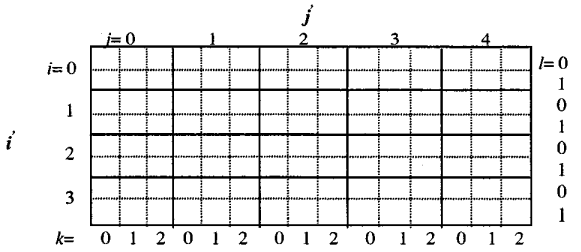


Figure 4: *EKMR*(4) for a (2×4)×(3×5) matrix.

3. Parallel Algorithms for Matrix Operations in *EKMR*(3)

Sequential algorithms for matrix operations based on *EKMR* have been presented in our previous work [15]. In this Section, we study data parallel algorithms for 3D matrix operations. In general, a data parallel algorithm can be divided into three phases: data partition, local computation, and results collection. We briefly examine these three phases and then present our parallel algorithms.

3.1 Three Phases for Parallel Algorithms

•Data Partition: To achieve parallelism, data is partitioned and distributed into processors for parallel operations. Row, column and 2D [11-13] are three common schemes for data partition in matrix algorithm. We have seen that matrices in *TMR*(3) are stored in a 3D fashion. On the other hand, matrices in *EKMR*(3) are stored in a 2D fashion. Figure 5 shows the three common ways to perform data partition in *TMR*(3) and *EKMR*(3). If non-continuous data are partitioned into a processor, we must collect each blocks, store them in a buffer before sending data to the processor. Usually, the cost for collecting data and sending them to processors cannot be ignored. Hence, data partition is an important issue in designing parallel algorithm. Many researches have worked on reducing the cost for collecting data.

•Local Computation: The next step is to perform the computation based on the algorithm and partial data. In general, the work in this phase is the same as the original sequential algorithm. However, there might be some changes in the scope of operation data, i.e., changes of indices for matrix operations.

•Results Collection: Results computed and scattered among processors must be collected for a final report. This phase is similar to data partition. If the partial results computed are non-continuous, they must be broken into blocks and then placed in their appropriate final locations.

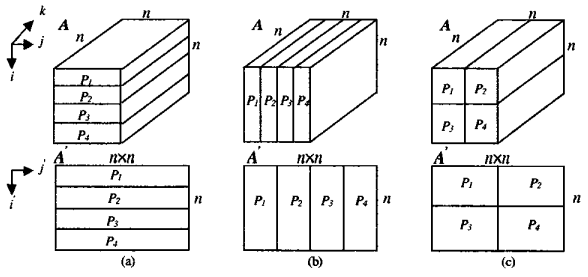


Figure 5: Partition schemes for the matrices A and A' in *TMR*(3) and *EKMR*(3). (a) Row partition. (b) Column partition. (c) 2D partition.

3.2 Row Partition Algorithms

Since both *TMR*(*n*) and *EKMR*(*n*) employ row-major storage scheme, we choose to use row partition in designing our parallel algorithms. Other partition schemes will be explored latter. To get better performance, we duplicate vector x or matrix B on all processors and do partition on matrix A in all our design. The data partition and results collection procedures are pretty much the same for all our matrix algorithms. We examine these two first, and then focus on local computation.

3.2.1 Data Partition and Results Collection

Let $A[k][i][j]$ be an $n \times n \times n$ matrix represented in *TMR*(3) and there are P processors in the parallel system. We have seen that *TMR*(3) can be viewed as a collection (along the k direction) of 2D matrices (along the $i \times j$ direction). Therefore, row partition for *TMR*(3) can be obtained by partitioning each 2D matrix row-wise and repeating the partition along the k direction. More precisely, let $r = n \% P$, and *row_size* denote the number of data rows assigned to each processor. With some arithmetic, it can be seen that *row_size*= $\lceil n/p \rceil$ for the first r processors and *row_size*= $\lfloor n/p \rfloor$ for the remaining $(P-r)$ processors.

Let $A'[i'][j']$ be an $n \times (n \times n)$ matrix represented in *EKMR*(3). It can be seen that matrix A' consists of n rows and each row contains n^2 elements. Partition A' by row means that we should assign *row_size* rows to each processor, where *row_size* is the same as that in the case of *TMR*(3). Since *EKMR*(3) is basically a 2D matrix, elements in the same row are stored continuously. There is no collection involved in the data partition phase.

3.2.2 Local Computation

In general, the local computation is the same as the original sequential algorithm presented in [15], except that there might be some changes in the scope of operation data. For briefness, we do not repeat our work and list the matrix-matrix multiplication algorithm using P processors for *TMR* and *EKMR* in Figures 6 and 7, respectively.

3.3 Column and 2D Partition Algorithms

Since data in each row is divided among P processors in the column partition, we need to collect pieces of data in the beginning of data partition phase for both *TMR* and *EKMR* based algorithms. Moreover, after the local computation each processor has only partial result for matrix-vector or matrix-matrix multiplication.

Hence, some extra work is needed in order to obtain final result during the results collections phase. For briefness, the details are omitted here.

Data Partition Procedure algorithm
/*Local Computation Procedure*/
for ($p\_id = 0$; $p\_id < P$; p_id ++)
  $row\_size = \lfloor n/p \rfloor$ *or* $\lceil n/p \rceil$. /*To change the scope for the index of sequential algorithm*/
  for ($k = 0$; $k < n$; k++)
    for ($i = 0$; $i < row\_size$; i++)
      for ($j = 0$; $j < n$; j++)
        for ($m = 0$; $m < n$; m++)
          $C[k][i][j] = C[k][i][j] + A[k][i][m] \times B[k][m][j]$;
      /*local result matrix size is $row\_size \times n^2$ */
Results Collection Procedure algorithm

Figure 6: Row partition matrix-matrix multiplication algorithm in *TMR*(3).

Data Partition Procedure algorithm
/*Local Computation Procedure*/
for ($p\_id = 0$; $p\_id < P$; p_id ++)
  $row\_size = \lfloor n/p \rfloor$ *or* $\lceil n/p \rceil$. /*To change the scope for the index of sequential algorithm*/
  for ($j = 0$; $j < n$; j++)
    $v = j \times n$;
    for ($i = 0$; $i < row\_size$; i++)
      for ($m = 0$; $m < n$; m++)
        $r = m \times n$;
        for ($k = 0$; $k < n$; k++)
          $C'[i][k+r] = C'[i][k+r] + A'[i][k+v] \times B'[j][k+r]$;
        /*local result matrix size is $row\_size \times n^2$ */
Results Collection Procedure algorithm

Figure 7: Row partition matrix-matrix multiplication algorithm in *EKMR*(3).

The 2D partition is a combination of row and column partition. Hence, piecewise data needs to be collected in buffer before distributed among processors in both representation schemes. Similarly, extra work needs to be done during the results collection phase. These algorithms are also omitted here.

4. Performance Evaluation

Parallel algorithms based on both matrix representations are analyzed in this Section. Moreover, some experiments were conducted in order to gain further insights. All our results indicate that algorithms based on *EKMR* are more efficient than that of *TMR*.

4.1 Analysis

In previous Sections, we have seen that all our parallel algorithms consist of three phases. In comparison algorithms based on two representation schemes, the amount of work for local computation are pretty much the same. This makes data partition and result collection become two important roles in deciding final performance.

In practice, there is data or task dependency [11-13] in parallel algorithms. Some extra works are needed to eliminate dependency. Fortunately, there is no data dependency in matrix operations we studied in this paper. We have mentioned that buffer and collection is needed if original data partitioned to a processor is not stored in continuous memory. We choose (1) number of data elements to be collected and (2) number of non-continuous data blocks as our evaluation metrics. This is because the first number indicates how many copy operations are needed to move data elements to buffer. The second number indicates how many jumps are needed during the copy operations, which affects cache usage (respectively, disk seek time) for in-of-core (respectively, out-of core) algorithms. We now proceed to find these numbers. Assuming that there are n^3 elements in matrix A, which is to be partitioned into P processors based on row or column partition. For simplicity, we further assume that n is a multiple of P.

•Row partition: For *TMR*(3), the first n/P rows in the first plane is assigned to processor 1, then the second n/P rows assigned to processor 2 and so on(c.f. Fig 5(a) for the case of P=4). There are n planes along the k direction. Hence, there are n non-continuous data blocks for processor 1, so are other processors. Therefore, the total number of non-continuous data segments would be summed to Pn. Since data assigned to each processor is not continuous, a buffer is needed for temporary storage and all the n^3 elements will be copied to buffer. The structure of *EKMR*(3) is exactly a 2D matrix in row-major storage. Therefore, all elements assigned to each processor are in continuous memory locations. Hence, no buffer is needed and both metrics are 0.

•Column partition: For *TMR*(3), there are n non-continuous data blocks for elements in the first plane assigned to processor 1(c.f. Fig 5(b)). Again, there are n planes along the k direction. Hence, there are n^2 non-continuous blocks for data assigned to each processor. This gives a total of Pn^2 non-continuous data blocks. For *EKMR*(3), there are n non-continuous blocks(along the i' direction) for each processor. Hence, there are Pn non-continuous data blocks for all processor. Since in both cases data are stored in non-continuous blocks, there are n^3 data elements to be collected.

•2D partition: Assuming that n^3 elements are to be partitioned into $P \times Q$ processors. For *TMR*(3), there are n/P non-continuous blocks assigned to processor 1 in the first plane and n total planes. Hence, the number of non-continuous data blocks is $P \times Q \times n \times (n/P)$, which gives Qn^2. Similarly, there is only one plane in *EKMR*(3). Therefore, the number of non-continuous data blocks is Qn. All n^3 elements need to be collected in both cases, since they are stored in non-continuous memory.

The analytical results for three data partition

schemes in *TMR*(3) and *EKMR*(3) are summarized in Table 1. It can be seen that our proposed parallel algorithms based on *EKMR*(3) should perform better than those based on *TMR*(3) regardless of in- or our-of-core environment. Results for *TMR*(4) and *EKMR*(4) can be obtained similarly. Table 1 also summarizes their lost of a matrix with size $n \times n \times n \times n$.

Table 1. Cost for partition procedure in three- and four-dimensional matrices.

	# of elements to be collected				# of Non-Continuous blocks			
	TMR(3)	*EKMR*(3)	*TMR*(4)	*EKMR*(4)	*TMR*(3)	*EKMR*(3)	*TMR*(4)	*EKMR*(4)
Row Partition (*P* processor)	n^3	0	n^4	0	Pn	0	Pn^2	0
Column Partition (*P* processor)	n^3	n^3	n^4	n^4	Pn^2	Pn	Pn^3	Pn
2D Partition (*P*x*Q* processor)	n^3	n^3	n^4	n^4	Qn^2	Qn	Qn^3	Qn

4.2 Experimental results

To assess the performance of our proposed algorithms, we have implemented them on an IBM SP2 machine. Parallel algorithms are implemented in *C+MPI* using SPMD model. Experiments for parallel algorithms consist of two parts. In the first part, we study effects of matrix size, which were run on SP2 with three nodes. In the second part, we investigate effect of number of processors in the parallel machine, which were run with a fixed matrix size 100x100x100 for 3D matrix and size 30x30x30x30 for 4D matrix and processor number varying from 2 to 16.

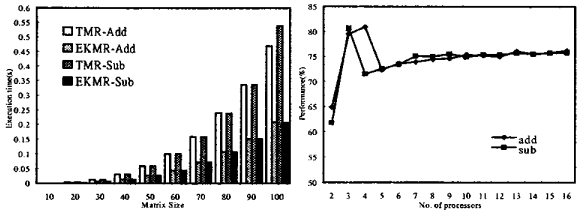
4.2.1 Performance of Parallel Algorithms in *EKMR*(3)

Performance of sequential algorithms in *EKMR*(3) have presented in [15]. We now study performance of parallel algorithms. In addition to execution time, we use a second comparison metric called relative performance, which is defined as

$$performance(\%) = \frac{Time(TMR\_A\lg) - Time(EKMR\_A\lg)}{Time(TMR\_A\lg)} \times 100$$

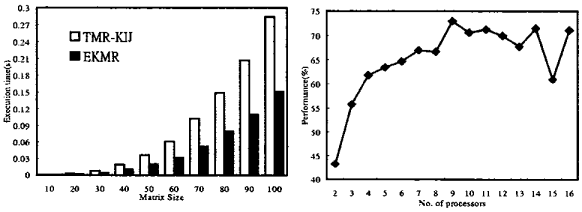
4.2.1.1 Row partition

From Figure 8(a), we can find that our proposed addition/subtraction algorithms outperform those in *TMR*(3) by about 50~85% for different matrix sizes on an SP2 with three nodes. The relative performance for different processor numbers is shown in Figure 8(b). We can see that *EKMR* algorithms are better than *TMR* algorithm by about 60~85% in relative performance. Results for matrix-vector multiplication is shown in Figure 9, where we can see that *EKMR* algorithm is faster than *TMR* algorithm by about 35~60% in execution time and about 40~75% better in relative performance. Results for matrix-matrix multiplication is displayed in Figure 10. It can seen that *EKMR* algorithm is faster than *TMR* algorithm by about 6~18% in execution time and about 8~17% better in relative performance.



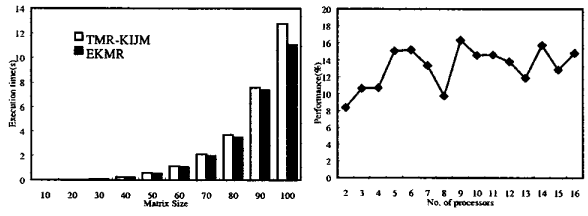
(a) (b)

Figure 8: (a) Execution time for matrix-matrix addition/subtraction with 3 processors. (b) Relative performance for matrix-matrix addition/subtraction.



(a) (b)

Figure 9: (a) Execution time for matrix-vector multiplication operations with 3 processors. (b) Relative performance for matrix-vector multiplication.



(a) (b)

Figure 10: (a) Execution time for matrix-matrix multiplication with 3 processors. (b) Relative performance for matrix-matrix multiplication.

4.2.1.2 Column and 2D partitions

Experiments for column partition were also conducted. Results indicate that algorithm based on *EKMR* outperforms that based on *TMR*. More precisely, the difference is about 10~25% in execution time and 10~20% in relative performance for addition/subtraction operation, about 6~10% in execution time and 6~15% in relative performance for matrix-vector multiplication, and about 3~8% in execution and 3~12% in relative performance for matrix-matrix multiplication.

The relative performance of 120x120x120 matrix-matrix multiplication with different numbers of processors has been studied. Results show that the *EKMR* algorithm is about 5~10% faster than the TMR algorithm.

4.2.2 Performance of Algorithms in *EKMR*(4)

To see how well the *EKMR* scheme can be extended to higher dimensions, we move our focus to *EKMR*(4). We have seen, in Table 1, that the row partition scheme should perform better than column and 2D partition. We have implemented matrix operation algorithms based on row partition and conducted some experiments in order to gain further insight. As we expected experimental results indicate that algorithm based on *EKMR* outperforms that based on *TMR*. In fact, the difference is about 55~62% in execution time and 55~63% in relative performance for addition/subtraction operation, about 40~50% in execution time and 40~60% in relative performance for matrix-vector multiplication, and about 25~30% in execution and 25~40% in relative performance for matrix-matrix multiplication.

5. Conclusions and Further Work

A new structure called *EKMR* for representation of multi-dimensional matrix has been proposed recently. The structure consists of a tree with 2D matrices in its leaves. Parallel matrix operation algorithms based on *EKMR* are developed. Our analysis and experimental results show that algorithms based on *EKMR* outperform those based on *TMR*. We plan to work on the following directions in the future: (1) study cache effect on different algorithms, (2) develop compression schemes for sparse matrices in the form of *EKMR* and *TMR*, (3) apply our proposed algorithms to research in multi-dimensional data cube, and (4) add other schemes, such as tensor product or Strassen's method, in our matrix multiplication algorithm to further improve performance.

References

[1] Aart J.C. Bik and Harry A.G. Wijshoff, "Compilation Techniques for Sparse Matrix Computations," *In Proceedings of Supercomputing*, pages 430-439, 1993.

[2] Aart J.C. Bik, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff, "Reshaping Access Patterns for Generating Sparse Codes," *In Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, 1994.

[3] Aart J.C. Bik and Harry A.G Wijshoff, "Automatic Data Structure Selection and Transformation for Sparse Matrix Computations," *Technical Report*, 1992.

[4] J.K. Cullum and R.A. Willoughby, "Algorithms for Large Symmetric Eignenvalue Computations," (birkhauser, Boston 1985).

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

[6] B. B. Fraguela, R. Doallo, E. L. Zapata, "Cache Misses Prediction for High Performance Sparse Algorithms, " 4th *International Euro-Par Conference* (*Euro-Par*'98), pp.224-233, September, 1998.

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

[8] B. B. Fraguela, R. Doallo, E. L. Zapata, "Automatic Analytical Modeling for the Estimation of Cache Misses, " *International Conference on Parallel Architectures and Compilation Techniques* (*PACT*'99), October, 1999.

[9] G.H. Golub and C.F. Van Loan, *Matrix Computations*, 2nd ed. (Johns Hopkins Univ.Press, Baltimore, 1989)

[10] Mahmut Kandemir, J. Ramanujam, Alok Choudhary, "Improving Cache Locality by a Combination of Loop and Data Transformations, " *IEEE Trans. on Computers*, vol 48, no. 2, February 1999.

[11] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill, "Compiling Parallel Sparse Code for User-Defined Data Structures," *In Proceedings of* 8th *SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[12] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill, "A Relation Approach to the Compilation of Sparse Matrix Programs," *In Euro Par*, August 1997.

[13] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill, "Compiling Parallel Code for Sparse Matrix Applications," *In Proceedings of the Supercomputing Conference*, August 1997.

[14] B. Kumar, C.-H. Huang, R. W. Johnson, and P. Sadayappan, " A Tensor Product Formulation of Strassen's Matrix Multiplication Algorithm with Memory Reduction," *In Proceedings of the* 7th *International Parallel Processing Symposium*, 1998.

[15] Jen-Shiuh Liu, Jiun-Yuan Lin, and Yeh-Ching Chung, "Efficient Representation for Multi-Dimensional Matrix Operations," *Proceedings of Workshop on Compiler Techniques for High Performance Computing* (*CTHPC*), pp. 133-142, March 2000, Taiwan.

[16] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes*, The Art of Scientific Computing, 2nd ed. (Cambridge University Press, 1992)

[17] Peter D. Sulatycke and Kanad Ghose, "Caching Efficient Multithreaded Fast Multiplication of Sparse Matrices," *In Proceedings of the* 1st *Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998.

[18] James B. White, III and P.Sadayappan, "On Improving the Performance of Sparse Matrix-Matrix Vector Multiplication," *IEEE Conference*, 1997.