

CL

An Efficient Representation of Higher Dimensional Arrays and Its Evaluation

by

Md Abu Hanif Shaikh

Roll No: 1107555

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science & Engineering



Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh

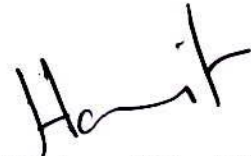
July 2016

Declaration

This is to certify that the thesis work entitled "An Efficient Representation of Higher Dimensional Arrays and Its Evaluation" has been carried out by Md Abu Hanif Shaikh in the Department of Computer Science and Engineering, Khulna University of Engineering & Technology, Khulna, Bangladesh. The above thesis work or any part of this work has not been submitted anywhere for the award of any degree or diploma.



Signature of Supervisor



Signature of Candidate

Approval

This is to certify that the thesis work submitted by Md Abu Hanif Shaikh entitled ^{An} "Efficient Representation of Higher Dimensional Arrays and It's Evaluation" has been approved by the board of examiners for the partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering in the Department of Computer Science and Engineering (CSE), Khulna University of Engineering & Technology, Khulna, Bangladesh in July 2016.

BOARD OF EXAMINERS

1. Atan 28.07.16
Dr. K.M. Azharul Hasan
Professor, Dept. of CSE
Khulna University of Engineering & Technology, Khulna
Chairman
(Supervisor)
2. Hadi 28-07-16
Head of the Department
Department of CSE
Khulna University of Engineering & Technology, Khulna
Member
3. Dr. M. M. A. Hashem 28/07/2016
Dr. M. M. A. Hashem
Professor, Dept. of CSE
Khulna University of Engineering & Technology, Khulna
Member
4. Muhammad Aminul Haque Akhand 28/7/16
Dr. Muhammad Aminul Haque Akhand
Professor, Dept. of CSE
Khulna University of Engineering & Technology, Khulna
Member
5. Abu Sayed Md. Latiful Hoque 28-07-2016
Dr. Abu Sayed Md. Latiful Hoque
Professor, Dept. of CSE
Bangladesh University of Engineering & Technology
Member
(External)

Acknowledgment

All the praise to the almighty Allah, whose blessing and mercy succeeded me to complete this thesis work fairly. I gratefully acknowledge the valuable suggestions, advice and sincere co-operation of Dr. K. M. Azharul Hasan, Professor, Department of Computer Science and Engineering, Khulna University of Engineering & Technology, under whose supervision this work was carried out. His open-minded way of thinking, encouragement and trust makes me feel confident to go through different research ideas. From him, I have learned that scientific endeavor means much more than conceiving nice algorithms and to have a much broader view at problems from different perspectives. I would like to convey my heartily ovation to all the faculty members, officials and staffs of the Department of Computer Science and Engineering as they have always extended their co-operation to complete this work. I am extremely indebted to the members of my examination committee for their constructive comments on this manuscript. Last but not least, I wish to thank my friends and my family for their constant support.

Author

Abstract

Scientific and engineering computing requires storing and operating on flooded amount of data having very high number of dimensions. Traditional multidimensional array is widely popular for implementing higher dimensional data but its performance diminishes with increased number of dimensions. On the other side, traditional row-column view of two-dimensional data is facile for implementation, imagination and visualization. This thesis represents a scheme for higher dimensional array implementation and operation with row-column abstraction which can fit an n -dimensional array into a single 2-dimensional array. A mathematical function fits odd dimensions along row-direction and even dimensions along column direction which gives lower index computation cost, higher data locality and better sequential access of memory. Performance of the proposed matricization is measured with matrix-matrix addition/subtraction and multiplication operation which give 70% and 72% improvement respectively for dense data. But most real world data is sparse and degree of data sparsity increases with increased number of dimensions. A loop transformation technique which access odd dimensions fast and then even dimensions is proposed to store any dimensional sparse arrays. In traditional scheme, n numbers of one-dimensional auxiliary arrays are necessary to store n -dimensional array but our scheme requires two one-dimensional auxiliary arrays only which gives 16 times space improvement for 32-dimensional sparse data. Traditionally, the compression ratio is inversely proportional to the number of dimensions but it is independent of number of dimensions in our scheme. The operation on stored sparse data is measured with matrix-matrix addition/subtraction and multiplication which show up to 70% improvement.

Contents

	Page No.
Title Page	i
Declaration	ii
Approval	iii
Acknowledgment	iv
Abstract	v
Contents	vi
List of Figures	viii
List of Tables	ix
List of Abbreviations	x
CHAPTER I Introduction	1
1.1 Introduction	1
1.2 Problem Statement	2
1.3 Objectives	3
1.4 Scope	4
1.5 Contributions	4
1.5 Organization of the Thesis	5
CHAPTER II Literature Review	6
2.1 Introduction	6
2.2 Multidimensional Access Methods	6
2.3 Storage Scheme for Higher Dimensional Arrays	12
2.4 Discussion	17
CHAPTER III Generalized 2-Dimensional Array	18
3.1 Introduction	18
3.2 Realization of 2-Dimensional Representation	18
3.2.1 2-Dimensional Representation of TMA(3)	19
3.2.2 2-Dimensional Representation of TMA(4)	19
3.2.3 2-Dimensional Representation of TMA(6)	20
3.2.4 2-Dimensional Representation of TMA(n)	21
3.3 Comparison of TMA and G2A for Matrix Operations	23
3.3.1 Matrix-Matrix Addition/Subtraction Algorithms	24
3.3.2 Matrix-Matrix Multiplication Algorithms	25
3.4 Theoretical Analysis	28
3.4.1 Cost of Index Computation	29
3.4.2 Cache Effect Analysis	33
3.5 Conclusion	35

CHAPTER IV	Generalized Compressed Row/Column Storage	36
	4.1 Introduction	36
	4.2 Realization of GCRS/GCCS Scheme	36
	4.2.1 Generating of GCRS/GCCS File	38
	4.3 Operation on GCRS/GCCS	42
	4.3.1 Matrix-Matrix Addition	42
	4.3.2 Matrix-Matrix Multiplication	44
	4.4 Theoretical Analysis	46
	4.4.1 Space Requirement of CRS/CCS and GCRS/GCCS	47
	4.5 Conclusion	49
CHAPTER V	Experimental Results	50
	5.1 Introduction	50
	5.2 Experimental Setup	50
	5.3 Experimental Results for G2A Operations	50
	5.4 Experimental Results for generation of CRS/CCS and GCRS/GCCS	52
	5.4.1 Time Requirement	52
	5.4.2 Space Requirement	53
	5.5 Experimental Result for GCRS/GCCS Operations	56
	5.6 Discussion	57
CHAPTER VI	Conclusions	58
	6.1 Concluding Remarks	58
	6.2 Future Scope	58
References		60

LIST OF FIGURES

Figure No.	Description	Page No.
2.1	Vector/ one-dimensional Array	6
2.2	Matrix or Two-dimensional Array	7
2.3	Three-dimensional Array	7
2.4	Visualization of Four-dimensional Array	8
2.5	Sparse Matrix or Two-dimensional Sparse Array	9
2.6	a) TMA(3) of size $3 \times 4 \times 5$ b) EKMR(3) of TMA having size $3 \times 4 \times 5$	11
2.7	CRS/CCS for 2-D sparse Array	13
2.8	CRS/CCS for sparse 3-D Array	13
2.9	CRS/CCS for sparse 4-D Array	14
2.10	ECRS for sparse 6-D Array	15
2.11	Query to retrieve chunks	15
2.12	ArrayStore for 3-D data	16
3.1	TMA(3) and its equivalent G2A	19
3.2	G2A representation of TMA(4)	20
3.3	G2A representation of TMA(6)	21
3.4	Addition and Multiplication of Matricized TMA(4) of size $2 \times 2 \times 2 \times 2$	24
4.1	G2A representation of TMA(6) having size $2 \times 2 \times 2 \times 3 \times 3 \times 2$	37
4.2	CRS and GCRS of TMA(6) having size $2 \times 2 \times 2 \times 3 \times 3 \times 2$	37
4.3	GCRS/GCCS from TMA(3) of size $2 \times 3 \times 3$	39
4.4	GCRS/GCCS of TMA(4)	41
4.5	Addition of two GCRS/GCCS	43
4.6	Multiplication of two GCRS/GCCS	45
5.1	Experimental Results for Matrix-matrix Addition	51
5.2	Experimental Performance for Matrix-matrix Multiplication	52
5.3	Time Requirement for CRS and GCRS	53

5.4	Space Requirement for CRS and GCRS	54
5.5	Compression Ratio for CRS and GCRS	55
5.6	Improvement of GCRS over CRS	55
5.7	Experimental Results for GCRS/GCCS Addition	56
5.8	Experimental Results for GCRS/GCCS Multiplication	57

LIST OF TABLES

Table No.	Description	Page No.
2.1	LoopCost for Matrix Multiplication	10
3.1	Multiplication of Matricized TMA	28
3.2	Parameters for Theoretical Analysis	29
4.1	Generation of GCRS from TMA(3)	39
4.2	Generation of GCCS from TMA(3)	40
4.3	Generation of GCRS from TMA(4)	41
4.4	Different Parameters for Theoretical Evaluation	46
5.1	Experimental Setup	50

LIST OF ABBREVIATIONS

TMA	Traditional Multidimensional Array
HPC	High Performance Computing
EKMR	Extended Karnaugh Map Representation
G2A	Generalized 2-dimensional Array
CRS	Compressed Row Storage
CCS	Compressed Column Storage
ECRS	CRS based on EKMR
ECCS	CCS based on EKMR
EaCRS	Extendible Array based CRS
xCRS	Extended CRS
MOLAP	Multidimensional On-Line Analytical Processing
GCRS	Generalized Compressed Row Storage
GCCS	Generalized Compressed Column Storage

CHAPTER I

Introduction

1.1 Introduction

Today's advanced scientific and engineering problems require a vast amount of computing power. Compute intensive tasks in various fields, including quantum mechanics, weather forecasting, climate research, cryptanalysis, molecular modeling and physical simulations like simulations of the early moments of the universe, airplane and spacecraft aerodynamics, the detonation of nuclear weapons, and nuclear fusion etc.[1][2] entail special attention of computer scientists. High-performance computing (HPC) incorporates all these computational tasks with exceptionally high requirements for computing power and memory capacity. Traditionally, these requirements were satisfied by introducing special computing techniques namely CUDA, X10, Julia etc.

Array is the most common and widely used data structure. Generally, most real world data has wide number of dimensions and often modeled with multidimensional array. Traditional Multidimensional Array (TMA) is extensively popular for its simple addressing function, memory layout, implementation procedure and random access capability[3]-[14]. But it has some limitation to handle and operation on higher dimensional data.

- Cost of index computations increases with increase the number of dimension.[2]
- The number of cache line accessed increases for higher dimensional data.[15][16]
- Most compilers[17] have limitation for implementing multidimensional array of very large number of dimensions.

Thus the special computing techniques through comprehensive research to handle large scale higher dimensional data efficiently and effectively are cramming needs to data scientists. It emphasizes the organization and implementation schemes on parallel and distributed computing platform. Experts suggest linearization of dimensions for

implementing higher dimensional array[18][19]. In fact, multidimensional arrays are just a logical abstraction above a linear storage system. It is good enough to implement in secondary memory as compilers allocate memory linearly. But operation cost and accessing time for secondary memory data is too high as well as parallelization is seldom possible for linearized data. It is well known that the compilers replicate secondary storage data to main memory for any type of computing either it is sequential or parallel machine. Multi-dimensional array operation like matrix-matrix addition/subtraction, matrix-matrix multiplication, sparse array storage, etc. require multiple access of same element and use of cache memory reduce the access time of the element. So, the pragmatic computing techniques which support parallelism, lower index computation cost, the lower operation cost and higher data locality is an important research issue.

1.2 Problem Statement

Many techniques have been proposed in the literature for improving multidimensional data computation such as TMA[20], Extended Karnaugh Map Representation (EKMR). EKMR[2] proposed by Lin *et. al.* can convert three and four dimensional array into two-dimensional array. The EKMR representation of higher ($n > 4$) dimensions is a hierarchical structure that contains array of pointers. For large values of n ($n > 4$) there are $(n-4)$ number of pointer arrays required and there is no generalization for higher dimensions like three or four dimensions. A technique based on loop transformation to improve the data locality for multi dimensional arrays is proposed in [15][16]. They demonstrated that this transformation is useful for array operations. The chunking[21][22], reordering[15][16], redundancy[23] and partitioning of the large array are proposed to make efficient access on secondary and tertiary memory devices. Caching by chunk by chunk for improving performance is proposed by [2][4][20]. In this scheme the large multidimensional arrays are broken into smaller parts called chunks for storage and processing. All the chunks are n dimensional with smaller length than the original array. A new programming language has been proposed to serve the computational power[22]. [23] shows a technique for storing and analyzing multidimensional array by chunking but there is no generalization from higher dimensions.

Most sparse array storage schemes are based on sparse matrix i.e. 2-dimensional array[22][24]-[28]. Some of them are time effective and some are space effective. Many

programming languages and compilers (X10[29], Julia[30], CUDA[31]) provide supports for sparse array but limited to only two dimensions. Matlab sparse toolbox supports for n -way sparse tensor [33] and have promising results for n -way tensor operation but its space complexity is very high. CRS/CCS scheme for higher dimensional array is based on the idea that a multidimensional array can be viewed as a collection of two-dimensional arrays [2]. But it requires $(n+1)$ one dimensional arrays to store an array of n dimensions. ECRS/ECCS scheme [34] works well but only for four dimensions. When the number of dimensions is greater than four then it requires an abstract pointer array to support higher dimensions. That's why it will be difficult to apply in practical situation when number of dimension becomes very high. The EaCRS scheme[35][36] which has a nice characteristic of dynamic extendibility[37]-[40] and supports well for higher dimensions. But it requires n one dimensional arrays to store an array of dimension n . Gundersen et al.[41] proposed a methodology to store like PATRICIA Trie Compression Storage (PTCS), Extended Compressed Row Storage (xCRS), Bit Encoded Extended Compressed Row Storage (BxCRS) and Hybrid Approach. But all of them require $(n+1)$ one dimensional arrays. Hence the schemes are not effective enough for storing and operation on higher dimensional arrays.

This thesis is going to describe a scheme to represent higher dimensional array (both dense and sparse) with a single two dimensional array which is facile for implementation, imagination and visualization. The generalized addressing function returns row-column abstractions which ensure lower index computation cost and higher data locality. Thus the simple algorithms for higher dimensional array operations like matrix-matrix addition, subtraction and multiplication would be introduce for both sparse and dense array. The proposed scheme can be applied to wide area including data mining[9][13][14], numerical analysis[13][26], GPU computing[2][42], MOLAP[18][38] and multi way data analysis[10][12].

1.3 Objectives

High Performance Computing, MOLAP or various scientific applications use multidimensional array as a basic data structure to represent high dimensional data. This is because multidimensional array has an inherent facility to compute indexes and aggregation operation. Supporting for very high dimensional data is an important

requirement of those applications since data widely vary in today's computing. Hence, a generalized array model or realization scheme is strong requirement of current era.

The main objective of this research topic can be summarized as –

- Overview of multi-dimensional array representation, its index computation, data locality and limitations of computing for multi-dimensional data.
- Proposed generalized two-dimensional representation of higher dimensional data, its formal notation and implementation. Different operation on generalized two-dimensional data and its evaluation with traditional representation.
- Sparse array overview and evaluation of different sparse array storage scheme and hence propose an array of storage scheme for higher dimensional sparse
- Comparing theoretical evaluation of proposed algorithms with equivalent traditional algorithms.
- Experimental results analysis to prove the soundness of theoretical evaluation.

1.4 Scope

The important scopes under this thesis are as follows:

- Computation can be done independently in each converted two-dimensional sub-array block which is very significant for parallel and distributed computing and applications.
- Evaluate the proposed algorithms time requirement with existing traditional array computation technique and EKMR[2].
- Different techniques of sparse data storage and evaluate the new storage scheme with existing schemes.
- Theoretical evaluation of space and time requirement for sparse data storage, depending on compression ratio, range of usability etc.
- Operations on stored data based on generalized row/column storage scheme.

1.5 Contribution

The contribution of this thesis can be summarized as follows:

- Generalization of higher dimensional array representation with row-column view.

- Algorithms for operations on stored data like matrix-matrix addition/subtraction and multiplication. Theoretical analysis is verified with experimental results.
- Generalization of higher dimensional sparse array storage with loop transformation. Details of theoretical analysis for compression ratio, range of usability with operations on stored data.

1.6 Organization of the Thesis

- **Chapter II** presents Literature Review that describes some of the traditional and prominent array organization and realization scheme that already exists. Some of these high dimensional data representation and compression methods will be described.
- **Chapter III** proposes a new multidimensional array representation scheme called Generalized Two-dimensional Array (G2A). It also explains the basic two-dimensional array/ matrix operations like addition, subtraction, multiplication, retrieval etc. over the proposed G2A scheme.
- **Chapter IV** illustrates the details of a generalized sparse array storage scheme called GCRS/GCCS based on Chapter III. Traditional addition, subtraction and multiplication operation on store sparse data are also described in this section.
- The experimental outcomes of proposed scheme and its evaluation are discussed in **Chapter V** which shows the technical soundness compelling with theoretical analysis.
- The future direction of work on the proposed model and the conclusive words about the model are outlined in **Chapter VI**.

CHAPTER II

Literature Review

2.1 Introduction

Array is the most common and widely used data structure. Most real world compute incentive data of weather forecasting, climate research, medical image processing, etc have wide number of dimensions and often modeled with multidimensional array. The location of each element can be computed with a single mathematical formula called addressing function. Array is also used to implement other data structures, such as lists, strings, heaps, hash tables, queues, stacks and VLists. But very few of them support for higher dimensional data. There are some other data structure or technique to increase the performance of higher dimensional data computation like loop transformation, optimal chunking, ArrayStore, SciDB, tensor decomposition, EKMR, CRS/CCS and ECRS/ECCS

2.2 Multidimensional Access Methods [1]-[14][26][33]-[35]

Array[2][5][34][35] is a collection of similar elements which is identified by at least one array index or key. The simplest form of array is a linear array called one-dimensional array or vector. For example an array of 20 32-bit integer variables, with indices 0 though 19, may be stored as 20 words at memory address 1000, 1004, 1008, ..., 1076, so that the element with index i has the address $1000+4\times i$ as shown in figure 2.1.

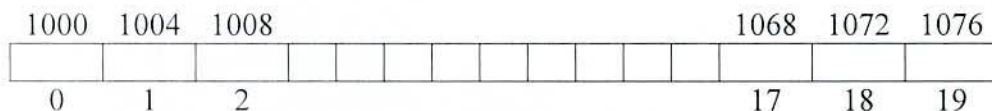


Figure 2.1 Vector/one-dimensional array

The mathematical concept of a matrix can be represented as a two-dimensional array having two index or keys where first key represent row number and second key represent column number. Typical graphical view/image on a plane is just a matrix or row-column view of a two-dimensional data. Figure 2.2 illustrates a matrix or two dimensional array,

A[l_1][l_2] of size 5×5 where l_1 and l_2 are length of dimensions. An element, A[x_1][x_2] can be identified either row major or column major order in linearized memory by below addressing function.

$$f_{\text{row major}}(x_1, x_2) = x_1 \times l_1 + x_2 \quad \text{and} \quad f_{\text{column major}}(x_1, x_2) = x_1 \times l_2 + x_2$$

	$x_2=0$	1	2	3	5
$x_1=0$	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

Figure 2.2 Matrix or Two-dimensional array

Lets us consider a three dimensional array, A[l_1][l_2][l_3] having three key where length of dimensions are l_1 , l_2 and l_3 . The set of continuous memory location into which the array maps is denoted by A[0: l] where $l = l_1 \times l_2 \times l_3 - 1$. A three dimensional coordinate system or cube having X, Y and Z axis can be represent as three dimensional data as shown in figure 2.3.

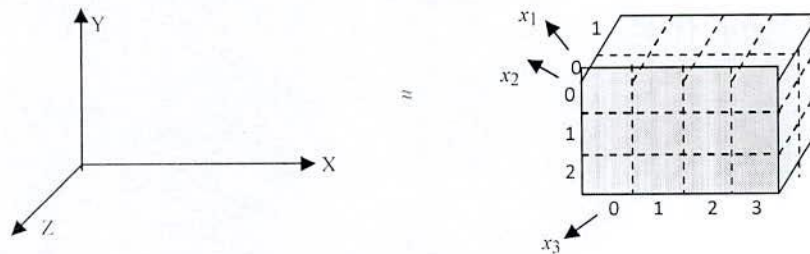


Figure 2.3 Three-dimensional Array

Each element of this TMA can be addressed in row major order as below

$$f(x_1, x_2, x_3) = x_1 \times l_2 \times l_3 + x_2 \times l_3 + x_3 \dots\dots\dots (2.1)$$

Similarly a four dimensional array, A[l_1][l_2][l_3][l_4] having four key to indentify each element. The continuous memory location to map with four keys is A[0: l] where $l = l_1 \times l_2 \times l_3 \times l_4 - 1$. A four dimensional array can be view as a l_1 number of three dimensional array, B [l_2][l_3][l_4] as shown in Figure 2.4. The element of this four dimensional array can map into continuous memory location with below addressing function.

$$f(x_1, x_2, x_3, x_4) = x_1 \times l_2 \times l_3 \times l_4 + x_2 \times l_3 \times l_4 + x_3 \times l_4 + x_4 \dots\dots\dots (2.2)$$

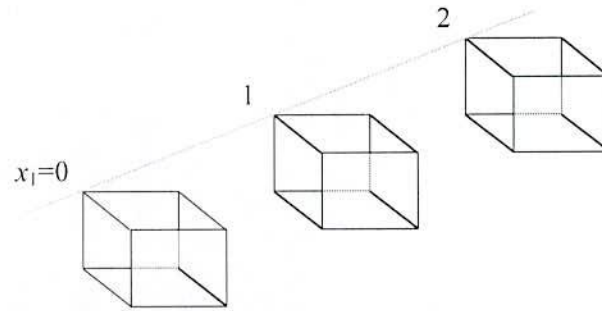


Figure 2.4 Visualization of Four-dimensional Array

If we consider a six dimensional array, $A[l_1][l_2][l_3][l_4][l_5][l_6]$ having six key to identify each element. The continuous memory location to map is $A[0:l]$ where $l = l_1 \times l_2 \times l_3 \times l_4 \times l_5 \times l_6 - 1$. Further we can say that above six dimensional array can be view as a l_1 number of five dimensional array namely $B[l_2][l_3][l_4][l_5][l_6]$. The row major addressing function for six dimensional array is mentioned below.

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = x_1 \times l_2 \times l_3 \times l_4 \times l_5 \times l_6 + x_2 \times l_3 \times l_4 \times l_5 \times l_6 + x_3 \times l_4 \times l_5 \times l_6 + x_4 \times l_5 \times l_6 + x_5 \times l_6 + x_6 \dots \quad (2.3)$$

Therefore a multidimensional array $\mathbf{A}[l_1][l_2] \dots [l_n]$ is an association between n -tuples of integer indices $\langle x_1, x_2, \dots, x_n \rangle$ and the elements of a set of E such that, to each n -tuples given by the ranges $0 < x_1 < l_1, 0 < x_2 < l_2, \dots, 0 < x_n < l_n$ correspond to an element of E . The domain from which the elements are chosen is immaterial and the assumption is made that only one memory location need be assigned to each n -tuples. Each array may be visualized as the lattice points in a rectangular region of n -space. The set of continuous memory locations into which the array maps is denoted by $\mathbf{A}[0:l]$ where

$$l = \left(\prod_{i=1}^n l_i \right) - 1$$

Any element in the multidimensional array is determined by addressing function as follows,

$$f(x_1, x_2, x_3, \dots, x_{n-1}, x_n) = x_1 l_2 l_3 \dots l_n + x_2 l_3 l_4 \dots l_n + \dots + x_{n-1} l_n + x_n \dots \dots \dots \quad (2.4)$$

An array is sparse [5][27][34][35] when most of its elements have default value (usually 0 or null). Sparse array may be any dimensional data. The sparsity problem becomes serious when the number of dimensions increases. This is because the number of all possible combinations of dimension values exponentially increases, whereas the number of actual data values would not increase at such a rate. In the case of sparse arrays, one can ask for a

value from an "empty" array position. If one does this, then for an array of numbers, a value of zero should be returned, and for an array of objects, a value of null should be returned. A naive implementation of an array may allocate space for the entire array, but in the case where there are few non-default values, this implementation is inefficient. Figure 2.5 illustrates a sparse matrix or two-dimensional array where only eight locations have non-zero value among total 25 locations.

	$x_2=0$	1	2	3	5
$x_1=0$	0	1	0	0	2
1	3	0	0	4	0
2	0	0	5	0	0
3	0	6	7	0	0
4	0	0	0	8	0

Figure 2.5 Sparse matrix or Two-dimensional sparse array

Loop transformation [2][15][16] is a compiler optimization technique to increase the performance. There are different types of loop transformation but this thesis only consider about loop permutation or re-organization to improve the memory performance. The fascinating characteristic of loop transformation is data locality. References to the same memory location or adjacent locations are reused within a short period of time. As of Steve Carr[15][16], data locality is measured with the algorithm called *LoopCost(l)*. The *LoopCost(l)* algorithms compute the costs of various loop orders of an array operation. The *LoopCost(l)* finds the number of cache line accessed by a loop l . The value of *LoopCost(l)* indicates the cache miss rate for a loop l and hence smaller the *LoopCost(l)* indicates the smaller the cache miss rate. Therefore the *LoopCost(l)* determines the best loop orders for nested loops with a specific innermost loop l . If the consideration is of loop cost for below matrix multiplication algorithm then the loop cost is listed at Table 2.1 with different loop order for cache line length r .

{ KJI ordering }

Do K=1, N

Do J=1, N

Do I=1, N

$$C(I,J) = C(I,J) + A(I,K) * B(K,J)$$

Table 2.1 LoopCost for Matrix Multiplication

Refs	J	K	I
C(I,J)	$n \times n^2$	$1 \times n^2$	$n/r \times n^2$
A(I,K)	$1 \times n^2$	$n \times n^2$	$n/r \times n^2$
B(K,J)	$n \times n^2$	$n/r \times n^2$	$1 \times n^2$
total	$2n^3+n^2$	$(r+1)n^3/r+n^2$	$2n^3/r+n^2$

Multidimensional array has an inherent facility of random accessing – the reason of becoming the most popular. There are many data structures already exist to represent multidimensional data. Some of the well-known and prominent data structures are discussed below.

The EKMR scheme[2][5][35] is based on the Karnaugh map[45] representation for minimizing Boolean expression. It can represent a three and four dimensional array with two dimensional arrays. Representation of higher dimensional (greater than 4) is abstract pointer array of EKMR of four dimensional data. Let a three dimensional TMA, $A[l_1][l_2][l_3]$ of size $3 \times 4 \times 5$ shown in Figure 2.6.a). The EKMR(3) of this TMA is a two-dimensional array, $A'[l_1'][l_2']$ where $l_1' = l_2 = 4$ and $l_2' = l_1 \times l_3 = 3 \times 5 = 15$ as shown in Figure 2.6.b). The representation ($A[x_1, x_2, x_3; l_1, l_2, l_3] \rightarrow A'[x_1', x_2'; l_1', l_2']$) is a permutation of elements where index tuple $\langle x_1', x_2' \rangle$ can be derived by $x_1' = x_2$ and $x_2' = x_3 \times l_1 + x_1$. EKMR can also returns to its original TMA ($A'[x_1', x_2'; l_1', l_2'] \rightarrow A[x_1, x_2, x_3; l_1, l_2, l_3]$) according to below backward mapping equation $x_2 = x_1'$, $x_1 = x_2' / l_1$ and $x_3 = x_2' \% l_1$. Similarly, a four-dimensional TMA, $A[l_1][l_2][l_3][l_4]$ can also be represented as a two-dimensional array, $A'[l_1'][l_2']$ where $l_1' = l_1 \times l_3$ and $l_2' = l_2 \times l_4$ in EKMR(4) scheme.

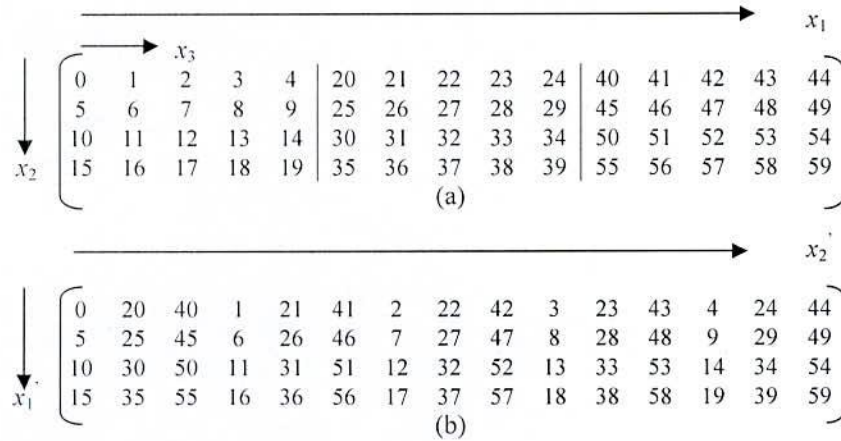


Figure 2.6 a) TMA(3) of size $3 \times 4 \times 5$ b) EKMR(3) of TMA having size $3 \times 4 \times 5$

The representation of n -dimensional TMA to EKMR(n) is based on EKMR(4) when $n > 4$ i.e. set of EKMR(4) construct EKMR(n). If the length of each dimension is l then EKMR(n) is represented by l^{n-4} EKMR(4) which introduce a structure to link all EKMR(4). The new structure is a one-dimensional array X of size l^{n-4} for one-to-one mapping with each EKMR(4). Consider a six-dimensional TMA, $A[l_1][l_2][l_3][l_4][l_5][l_6]$ of size $3 \times 2 \times 2 \times 3 \times 4 \times 5$. Equivalent EKMR(6) is represented by six (3×2) EKMR(4) where each EKMR(4) is a $(2 \times 4) \times (3 \times 5)$ two dimensional array.

A tensor $[1][13][26][32][33]$ is a multidimensional array. More formally, an N -way or N^{th} -order tensor is an element of the tensor product of N vector spaces, each of which has its own coordinate system. This notion of tensors is not to be confused with tensors in physics and engineering (such as stress tensors), which are generally referred to as tensor fields in mathematics. A third-order tensor has three indices, as shown in Figure 2.3. A first-order tensor is a vector, a second-order tensor is a matrix and tensors of order three or higher are called higher-order tensors.

Matricization, also known as unfolding or flattening, is the process of reordering the elements of an N -way array into a matrix. For instance, a $2 \times 3 \times 4$ tensor can be arranged as a 6×4 matrix or a 3×8 matrix, and so on. The mode- n matricization of a tensor $X \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ is denoted by $X_{(n)}$ and arranges the mode- n fibers to be the columns of the resulting matrix. Though conceptually simple, the formal notation is clunky. Tensor element (i_1, i_2, \dots, i_N) maps to matrix element (i_n, j) , where

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^N (i_k - 1)J_k \text{ with } J_k = \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m$$

The concept is easier to understand using an example. Let the frontal slices of $X \in \mathbb{R}^{3 \times 4 \times 2}$

$$X_1 = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix}$$

Then the three mode- n unfolding are

$$X_{(1)} = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{bmatrix}$$

$$X_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 18 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{bmatrix}$$

$$X_{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \dots & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & \dots & 21 & 22 & 23 & 24 \end{bmatrix}$$

It is also possible to vectorize a tensor. Once again the ordering of the elements is not important so long as it is consistent. In the example above, the vectorized version is

$$\text{vec}(X) = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 23 \\ 24 \end{bmatrix}$$

2.3 Storage Scheme for Higher Dimensional Arrays [5][7][18]-[24][27][32][33]

Multidimensional array are the basic data structure used in many applications such as MOLAP and compute intensive task. But in many cases, they are found to be sparse in nature – i.e. many of the array cells contain null values and consume unnecessary space. So it is important to design a technique, ‘The Storage/Compression’, to store such arrays. Some common storage schemes are reviewed below.

CRS/CCS[5][27][34][35] scheme is based on sparse matrix storage where three vectors namely RO, CO and VL are needed. Let us consider a matrix, $A_{m \times n}$ where m is the number of rows and n is the number of column. In CRS scheme, the length of RO is $m+1$ which is

$n+1$ in CCS. First element of RO is zero and later elements store cumulative sum of total number of non-zero elements in each row (each column in CCS). VL stores non-zero elements itself and CO store respective column index (row index for CCS) of non-zero values. Thus, the CCS is the CRS on transpose of targeted matrix. Figure 2.7 illustrates a two-dimensional array/matrix of 4×4 and equivalent CRS/CCS. There are seven (7) non-zero element out of total sixteen (16) elements. To store this matrix in CRS/CCS scheme, it requires to store a total of $(4+1)+2 \times 7=19$ elements. First element of RO is initialized with 0 and rests are the cumulative sum of total number of non-zero elements in each row (column for CCS). The number of non-zero element(s) in j 'th Row (Column for CCS) can be found by $RO[j+1]-RO[j]$.

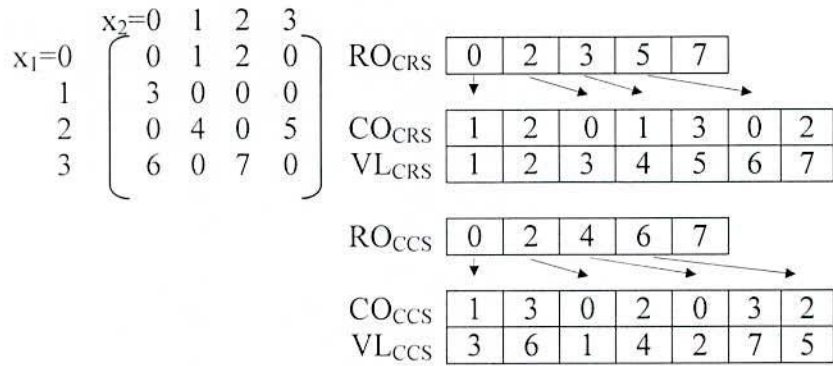


Figure 2.7 CRS/CCS for 2-D sparse array

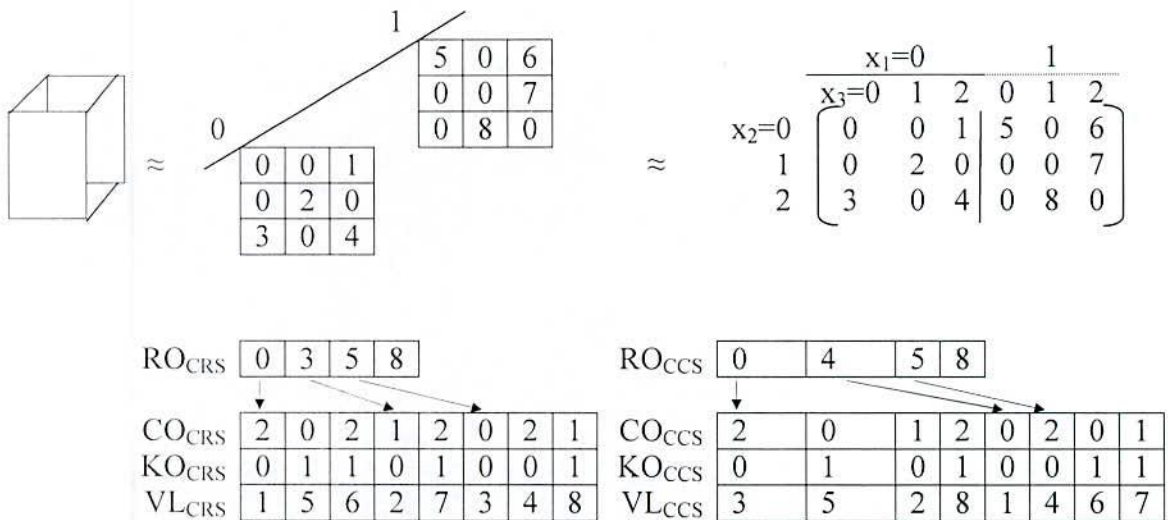


Figure 2.8 CRS/CCS for sparse 3-D array

Figure 2.8 shows the storage of a three dimensional array or cube of size $2 \times 3 \times 3$. This cube can be viewed as two two-dimensional arrays having size 3×3 for each. An additional array named KO is introduced which stores indices of first dimension x_1 along with RO, CO and VL. The number of non-zero elements is eight (8) out of eighteen (18) elements

and it requires to store $(1+3)+3 \times 8=28$ elements. Similarly it needs two KO to store a four-dimensional array of Figure 2.9 of size $2 \times 2 \times 3 \times 3$ where KO^0 and KO^1 store indices of x_1 and x_2 respectively. In this case it requires storing total of $(1+3) + 4 \times 12 = 52$ elements. Thus, it needs n number of auxiliary one-dimensional arrays to store a n -dimensional array.

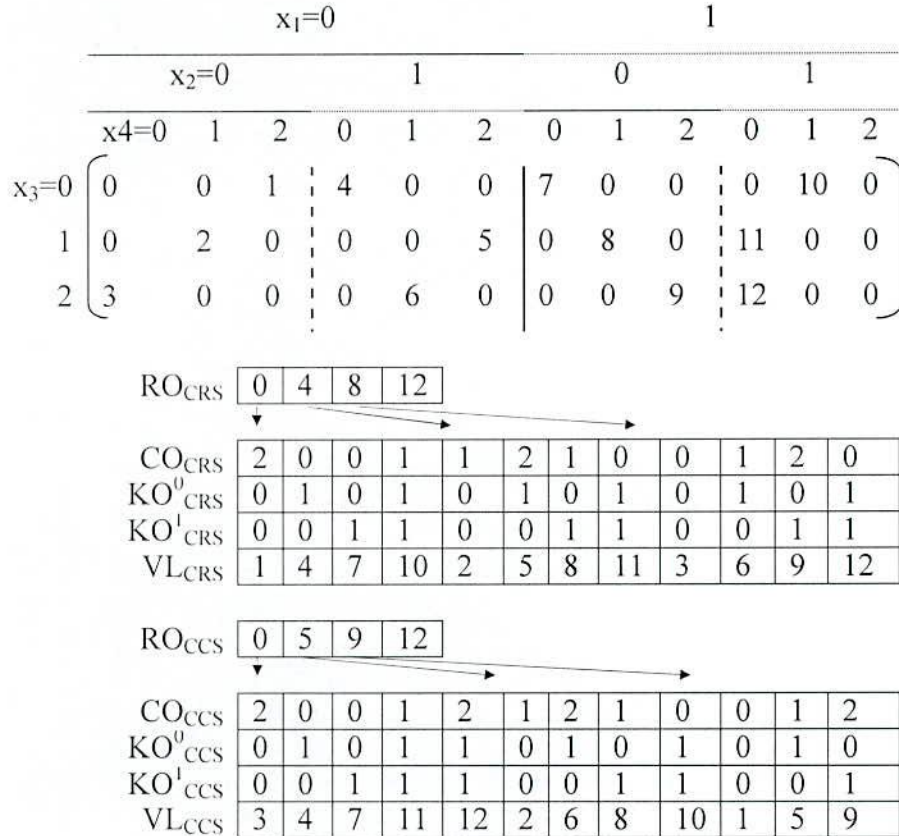


Figure 2.9 CRS/CCS for sparse 4-D array

The ECRS/ECCS scheme [5][27][34] use one one-dimensional floating-point array V and two one-dimensional integer arrays R and CK to compress a multidimensional sparse array based on the EKMR scheme. Given a sparse array based on the EKMR(3), the ECRS (ECCS) scheme compresses all of nonzero array elements along the rows (columns for ECCS) of the sparse array. Array R stores information of nonzero array elements of each row (column for ECCS). The number of nonzero array elements in the i^{th} row (j^{th} column for ECCS) can be obtained by subtracting the value of $R[i]$ from $R[i+1]$. Array CK stores the column (row for ECCS) indices of nonzero array elements of each row (column for ECCS). Array V stores the values of nonzero array elements. Similarly, It can be used arrays R , CK , and V to compress a sparse array based on the EKMR(4) in the ECRS/ECCS schemes. Since EKMR(k) can be represented by m^{k-4} EKMR(4), in the

ECRS/ECCS schemes, each EKMR(4) is first compressed by using arrays R, CK, and V. Then, an abstract pointer array with a size of m^{k-4} is used to link arrays R, CK, and V in each EKMR(4). For example, assume that there is a $3 \times 2 \times 2 \times 3 \times 4 \times 5$ sparse array A based on the TMR(6). The sparse array A' based on the EKMR(6) can be represented by six EKMR(4) with a size of 8×15 . The ECRS/ECCS scheme compress each EKMR(4) to arrays R, CK, and V. Then, use an abstract pointer array with a size of 6 to link arrays R, CK, and V of each EKMR(4) shown in Figure 2.10.

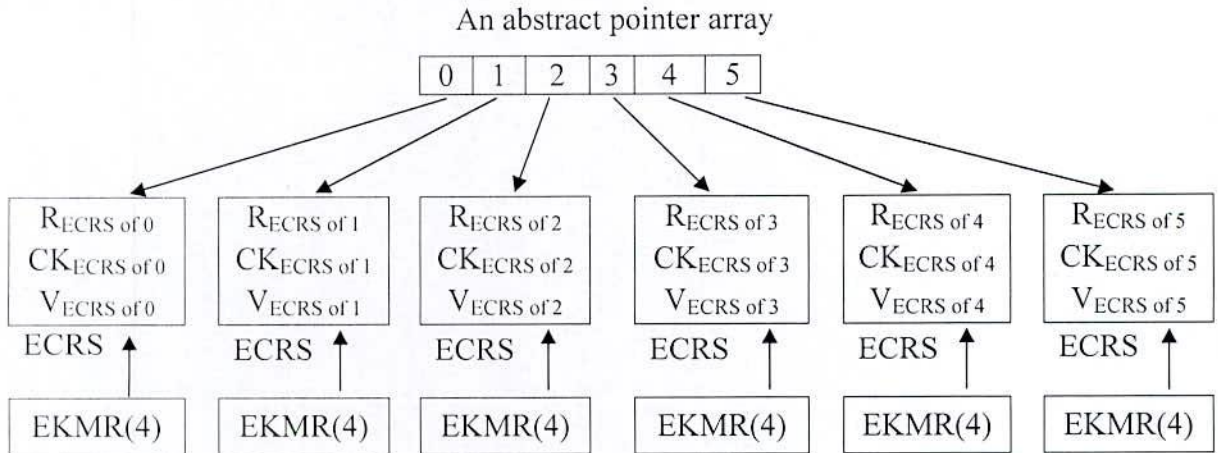


Figure 2.10 ECRS for sparse 6-D array

The optimal chunking technique [23] partitions multidimensional array into coarse grained hyper-rectangular blocks called chunks. A chunk is defined by the index range of values along each dimension. A query over the dataset retrieves either the entire array or sub-array based on overlapping the query result. An optimal chunking is characterized as chunk size and chunk shape. Suppose a n -dimensional array $A[l_1][l_2] \dots [l_n]$, consists of $\prod_{i=1}^n l_i$ elements. The storage of A is done by partitioning A into equal shape rectangular chunks such that each chunk fits on a disk block, i.e., if each chunk has dimensions $\langle c_1, c_2, c_3, \dots, c_n \rangle$ then $\prod_{i=1}^n c_i < C$.

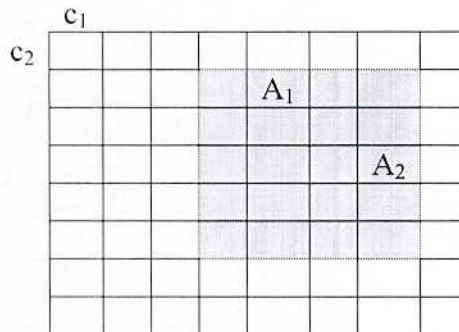


Figure 2.11 Query to retrieve chunks

The system supports queries that retrieve rectangular sub-arrays of A . A query $q = \langle [u_1 : v_1], [u_2 : v_2], [u_3 : v_3], \dots, [u_n : v_n] \rangle$ specifies a lower bound u_i and upper bound v_i on each of the n dimensions. The query retrieves all elements $\langle x_1, x_2, x_3, \dots, x_n \rangle$ of A such that $u_i < x_i < v_i$ for $1 \leq i < n$. Figure 2.11 illustrates a 2-dimensional query of shape $\langle A_1, A_2 \rangle$ operating on a chunked array where each chunk has the shape $\langle c_1, c_2 \rangle$.

ArrayStore[7] is a storage manager which supports range query as well as binary operations like join and complex user defined function. ArrayStore takes the approach of breaking an array into fragments called chunks and storing these chunks on disk. Figure 2.12 shows array A_1 of size $4 \times 4 \times 4$ which is divided into eight $2 \times 2 \times 2$ chunks. Each chunk is a unit of I/O (a disk block or larger). Each X-Y, X-Z, or Y-Z slice needs to load 4 I/O units. The array A_2 is laid out linearly through nested traversal of its axes without chunking. X-Y needs to load only one I/O unit, while X-Z and Y-Z need to load the entire array.

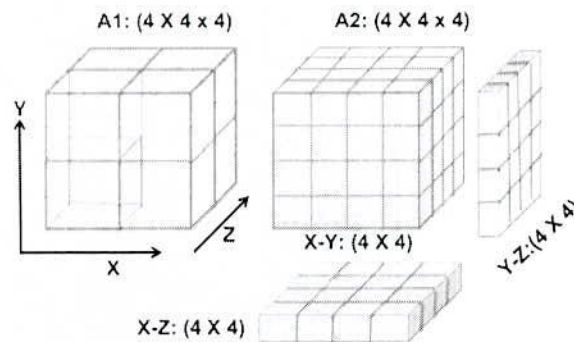


Figure 2.12 ArrayStore for 3-D data

ArrayStore use two types of chunking scheme namely Regular Chunks (REG) and Irregular Chunks (IREG). Each array in ArrayStore is represented with one data file and one metadata file. The data file contains the actual array values. The metadata file contains array meta information such as number of dimensions, total number of chunks, and in the case of regular chunking the number of chunks.

SciDB[24] is a multidimensional array data model which supports both functional and SQL-like query language. It was pretty obvious that SciDB had to run on a grid (or cloud) of computers. It should chunk arrays to storage blocks using some (or even all) of the dimensions. SciDB should chunk arrays across the nodes of a grid, as well as locally in

storage. Hence, it distributes chunks to nodes using hashing, range partitioning, or a block-cyclic algorithm.

2.4 Discussion

Every array models described in this chapter have some pros and cons. TMA is still the standard to implement higher dimensional data and good for random accessing but the performance of TMA drastically decrease with the increase of number of dimensions. EKMR is a prominent technique to represent higher dimensional data with two-dimensional data. But when the number of dimension is greater than four then it uses an abstract pointer array to point each EKMR(4) which make EKMR clunky to handle data with very high dimensions. Tensor decomposition slices the tensor into vector which is prominent for operation but very costly to store.

Most storage technique of higher dimensional sparse data is based on CRS/CCS scheme which is excellent to store sparse matrix. But using CRS/CCS for higher dimensional sparse data is hardly possible for its storage size. ECRS/ECCS is better than CRS/CCS which works well until four dimensional arrays. When the dimensional is greater than four then it stores as a collection of ECRS/ECCS(4). Generating the collection of ECRS/ECCS(4) is inefficient when the data dimension is very high. Optimal chunking, ArrayStore and SciDB based on the splitting the whole array into smaller size. But defining the size is widely affect the performance for higher dimensional data

Though, there are a lot of research has been done on array model, but only a few researches have been made on the generalization the scheme to any number of dimensions. Hence the proposed generalized representation scheme will outperform over TMA and EKMR scheme. The detail of the proposed scheme is presented in the next chapter.

CHAPTER III

Generalized 2-Dimensional Array

3.1 Introduction

Traditional multidimensional array is widely popular for implementing higher dimensional data but its performance diminishes with the increase of the number of dimensions. On the other side, traditional row-column view is facile for implementation, imagination and visualization. It is well known that the multidimensional array is the logical abstraction and linearized when stored on the memory. Compilers/ programming languages map the array index into the linearized memory address. So, it needs to compute the specified dimensional indices. If we consider a TMA(3) array, $A[l_1][l_2][l_3]$ then a tuple $\langle x_1, x_2, x_3 \rangle$ can be linearized and identified by array linearization function as follows

$$f(x_1, x_2, x_3) = x_1/l_3 + x_2/l_3 + x_3$$

This chapter represents an implementation procedure for n -dimensional array with row-column abstraction which named **Generalized Two-dimensional Array (G2A)**. Odd dimensions contribute along row-direction and even dimensions along column direction which gives lower cost of index computation and higher data locality. It is not related with dimension reduction depending upon eigen value. It is a permutation on higher dimensional data to fit into a new two-dimensional array. Thus the length and indices of new 2-Dimensional array is determined based on n -Dimensional arrays' length and indices. To do this, G2A fits $\lceil n/2 \rceil$ number of dimensions along row direction and the rest $n/2$ number of dimensions along column direction. Odd dimensions contribute for rows and even dimensions contribute for column.

3.2 Realization of 2-Dimensional Representation

G2A is the way of representing an n -dimensional array ($n > 2$) with a two-dimensional array. Let, $A[l_1][l_2] \dots [l_n]$ be a TMA(n) of size $[l_1, l_2, \dots, l_n]$ and $\langle x_1, x_2, \dots, x_n \rangle$ be the subscripts of

an element of A; where l_1, l_2, \dots, l_n is the length of each dimension d_1, d_2, \dots, d_n and $x_i = 0, 1, 2, 3, \dots, (l_i-1)$ ($0 \leq i \leq l$). The representation of the TMA(n), A into a G2A $A'[l'_1][l'_2]$ of size $[l'_1, l'_2]$ and subscripts $\langle x'_1, x'_2 \rangle$ where l'_1 and l'_2 are the length of each dimension d'_1 and d'_2 ; $x'_1 = 0, 1, 2, \dots, (l'_1-1)$ and $x'_2 = 0, 1, 2, \dots, (l'_2-1)$. In the following the conversion of TMA(3), TMA(4) and TMA(6) to G2A are shown and then the generalization for TMA(n) is described.

3.2.1 2-Dimensional Representation of TMA(3)

Consider a three dimensional TMA, $A[l_1][l_2][l_3]$ of length $[l_1, l_2, l_3] \approx [2, 3, 4]$. It can be represented as a two-dimensional array, $A'[l'_1][l'_2]$ having length $[l'_1, l'_2] \approx [8, 3]$ as

$$l'_1 = l_1 \times l_3 \text{ and } l'_2 = l_2$$

and the G2A elements index tuple $\langle x'_1, x'_2 \rangle$ can be derived from TMA tuple $\langle x_1, x_2, x_3 \rangle$ by

$$x'_1 = x_1 \times l_3 + x_3 \text{ and } x'_2 = x_2$$

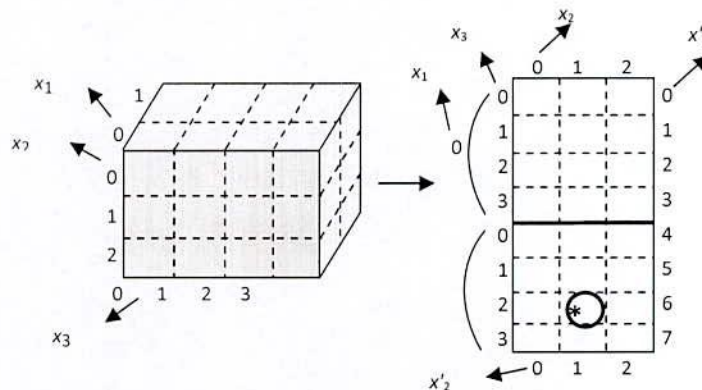


Figure 3.1 TMA(3) and its equivalent G2A

Thus a TMA index tuple $\langle 1, 1, 2 \rangle$ is equivalent to G2A tuple $\langle 6, 1 \rangle$. In reverse, it is also possible to reconstruct a TMA from its corresponding G2A. This is done by finding corresponding TMA tuple which is called backward mapping. Backward mapping to construct a TMA(3) from G2A can be as follows

$$x_1 = x'_1 / l_3, x_3 = x'_1 \% l_3 \text{ and } x_2 = x'_2$$

Here % indicates the 'modulus' operation and / indicates 'division' operation. Thus, a G2A index tuple $\langle 6, 1 \rangle$ is equivalent to TMA tuple $\langle 1, 1, 2 \rangle$.

3.2.2 2-Dimensional Representation of TMA(4)

Let, a four-dimensional TMA, A of length $[l_1, l_2, l_3, l_4] \approx [2, 3, 3, 2]$. Its corresponding G2A (Figure 5), $A'[l'_1][l'_2]$ where l'_1 and l'_2 can be found as follows:

$$l_1' = l_1 \times l_3 = 2 \times 3 = 6 \text{ \& } l_2' = l_2 \times l_4 = 3 \times 2 = 6$$

and any G2A index $\langle x_1', x_2' \rangle$ can be found with corresponding TMA index $\langle x_1, x_2, x_3, x_4 \rangle$ as

$$x_1' = x_1 \times l_3 + x_3 \text{ \& } x_2' = x_2 \times l_4 + x_4$$

As example if a TMA index is $\langle 1, 1, 2, 0 \rangle$, then its equivalent G2A index will be $\langle 1 \times 3 + 2, 1 \times 2 + 0 \rangle = \langle 5, 2 \rangle$. In reverse, if a G2A index $\langle 5, 2 \rangle$ is known then its equivalent TMA index can be found by

$$x_3 = x_1' \% l_3 = 5 \% 3 = 2, x_1 = x_1' / l_3 = 5 / 3 = 1 \text{ and}$$

$$x_4 = x_2' \% l_4 = 2 \% 2 = 0, x_2 = x_2' / l_4 = 2 / 2 = 1$$

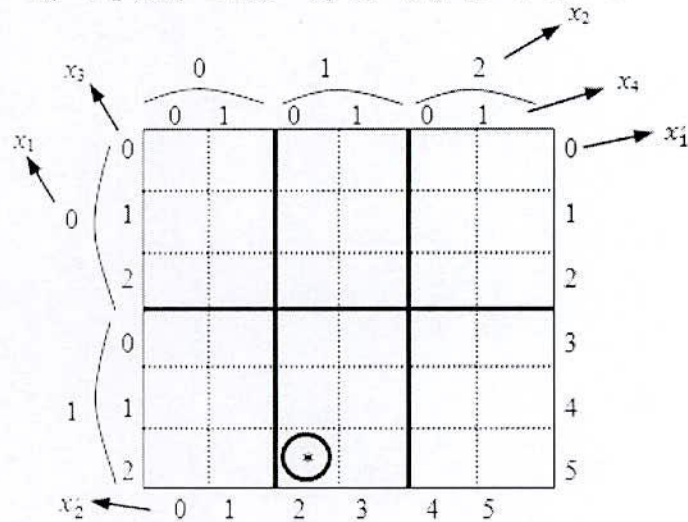


Figure 3.2 G2A representation of TMA(4)

3.2.3 2-Dimensional Representation of TMA(6)

Consider a six dimensional TMA, $A[l_1][l_2][l_3][l_4][l_5][l_6]$ of length $[l_1, l_2, l_3, l_4, l_5, l_6] \approx [2, 2, 2, 3, 3, 2]$. It can be represented as a two-dimensional array, $A'[l_1'][l_2']$ having length $[l_1', l_2'] \approx [12, 12]$ as

$$l_1' = l_1 \times l_3 \times l_5 \text{ and } l_2' = l_2 \times l_4 \times l_6$$

and the G2A elements index tuple $\langle x_1', x_2' \rangle$ can be derived from TMA tuple $\langle x_1, x_2, x_3, x_4, x_5, x_6 \rangle$ by

$$x_1' = x_1 \times l_3 \times l_5 + x_3 \times l_5 + x_5 \text{ and } x_2' = x_2 \times l_4 \times l_6 + x_4 \times l_6 + x_6$$

Thus a TMA index tuple $\langle 1, 1, 1, 0, 0, 1 \rangle$ is equivalent to G2A tuple $\langle 9, 7 \rangle$. In reverse, it is also possible to reconstruct TMA from its corresponding G2A by backward mapping as

$$x_5 = x_1' \% l_5, x_3 = x_1' \% (l_3 \times l_5), x_1 = x_1' \% (l_1 \times l_3 \times l_5)$$

$$\text{and } x_6 = x_2' \% l_6, x_4 = x_2' \% (l_4 \times l_6), x_2 = x_2' \% (l_2 \times l_4 \times l_6)$$

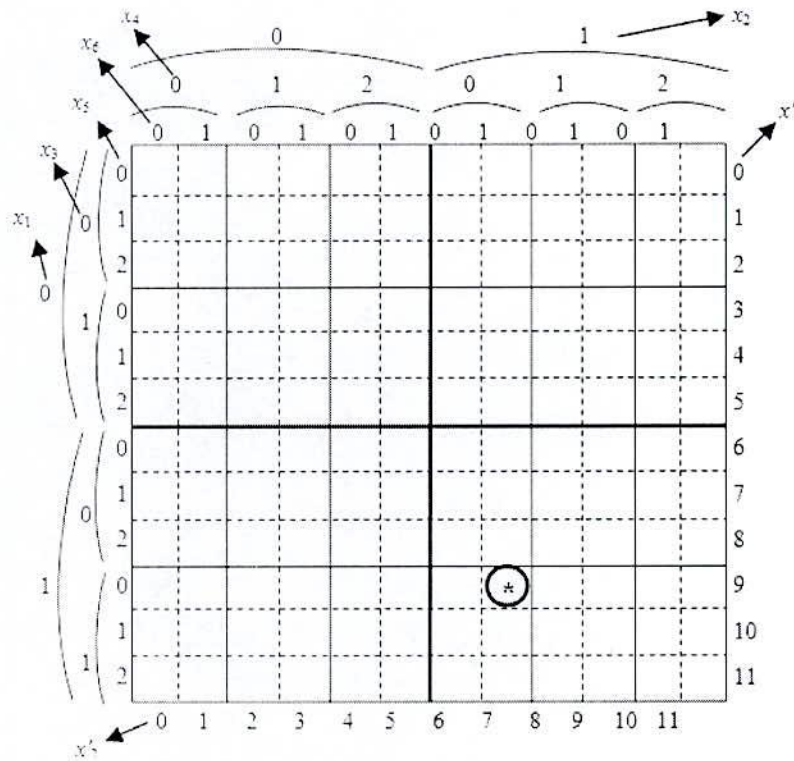


Figure 3.3 G2A representation of TMA(6)

3.2.4 2-Dimensional Representation of TMA(n)

Similarly it is possible to represent any dimensional array ($n > 2$) with a single 2-dimensional array. The length of generalized 2-dimensional array, $A'[l_1][l_2]$ from n -dimensional TMA, $A[l_1][l_2] \dots [l_n]$ length can have two cases as below

Case 1: if n is even

$$l_1' = l_1 \times l_3 \times l_5 \times \dots \times l_{n-3} \times l_{n-1} \text{ and } l_2' = l_2 \times l_4 \times l_6 \times \dots \times l_{n-2} \times l_n$$

Case 2: if n is odd

$$l_1' = l_1 \times l_3 \times l_5 \times \dots \times l_{n-2} \times l_n \text{ and } l_2' = l_2 \times l_4 \times l_6 \times \dots \times l_{n-3} \times l_{n-1}$$

Above two cases can be generalized as

$$l_i' = \prod_{j=0}^{\frac{n-i}{2}} l_{i+2j} \quad [i = 1, 2 \text{ and } j = 0, 1, 2, 3, \dots, n]$$

To generate a G2A from n -dimensional TMA it is necessary to generalize the TMA index tuple $\langle x_1, x_2, \dots, x_n \rangle$ into G2A's index tuple $\langle x_1', x_2' \rangle$. Again two cases may occur based on the value of n .

Case 1: if n is even

$$x_1' = x_1 l_3 l_5 \dots l_{n-3} l_{n-1} + x_3 l_5 l_7 \dots l_{n-3} l_{n-1} + \dots + x_{n-3} l_{n-1} + x_{n-1}$$

$$\text{and } x_2' = x_2 l_4 l_6 \dots l_{n-2} l_n + x_4 l_6 l_8 \dots l_{n-2} l_n + \dots + x_{n-2} l_n + x_n$$

Case 2: if n is odd

$$x_1' = x_1 l_3 l_5 \dots l_{n-2} l_n + x_3 l_5 l_7 \dots l_{n-2} l_n + \dots + x_{n-2} l_n + x_n \text{ and}$$

$$x_2' = x_2 l_4 l_6 \dots l_{n-3} l_{n-1} + x_4 l_6 l_8 \dots l_{n-3} l_{n-1} + \dots + x_{n-3} l_{n-1} + x_{n-1}$$

Above two cases and equations may be generalized to generate G2A tuples from TMA tuples as

$$x'_i = \sum_{j=0}^{\frac{n-i}{2}} \left(x_{i+2j} \prod_{k=j+1}^{\frac{n-i}{2}} l_{i+2k} \right)$$

$[i = 1, 2 \text{ and } j, k = 0, 1, 2, 3, \dots, n]$

The algorithm to implement above notation is shown in Algorithm 3.1 and Algorithm 3.2.

Algorithm 3.1: finding G2A indexes from TMA indexes:

G2A-forward_mapping($x_1', x_2', x_1 - x_n, l_1 - l_n$)

1. Initialize $x_1' := 0, x_2' := 0$
2. Repeat $i := 1$ to n
3. Repeat $j := i + 2$ to n
4. $x_i := x_i \times l_j$
5. $j := j + 2$
6. $x'_{2-i\%2} := x'_{2-i\%2} + x_i$

For backward mapping, if a G2A tuple $\langle x_1', x_2' \rangle$ is known then its equivalent TMA tuple $\langle x_1, x_2, x_3, \dots, x_n \rangle$ can be derived. To do this, two derivations can be considered based on the value of n .

Case 1: if n is even

$$x_n = x_2' \% l_n$$

$$x_i = (\dots (x_2' / l_n) \dots / l_{i+2}) \% l_i \quad [i = 4, 6, \dots, n-2]$$

$$x_2 = ((\dots ((x_2' / l_n) / l_{n-2}) \dots) / l_6) / l_4$$

$$x_{n-1} = x_1' \% l_{n-1}$$

$$x_j = (\dots (x_1' / l_{n-1}) \dots / l_{j+2}) \% l_j \quad [i = 3, 5, \dots, n-3]$$

$$x_1 = ((\dots ((x_1' / l_{n-1}) / l_{n-3}) \dots) / l_5) / l_3$$

Case 2: if n is odd

$$x_n = x_1' \% l_n$$

$$x_i = (\dots (x_1' / l_n) \dots / l_{i+2}) \% l_i \quad [i = 3, 5, \dots, n-2]$$

$$x_1 = ((\dots ((x_1' / l_n) / l_{n-2}) \dots) / l_5) / l_5$$

$$x_{n-1} = x_2' \% l_{n-1}$$

$$x_j = (\dots (x_2' / l_{n-1}) \dots / l_{j+2}) \% l_j \quad [i = 4, 6, \dots, n-3]$$

$$x_2 = ((\dots ((x_2' / l_{n-1}) / l_{n-3}) \dots) / l_6) / l_4$$

The generalization of above mentioned backward mapping can be done as below

$$\left. \begin{array}{l} x_i = x'_{2-i \% 2} \% l_i \\ x_{2-i \% 2} = x'_{2-i \% 2} / l_i \end{array} \right\} \text{ where } i = n \text{ to } 1$$

Algorithm 3.2 shows the reverse mapping which described above.

Algorithm 3.2: finding TMA(n) indexes from G2A indexes:

G2A-backward_mapping($x_1', x_2', x_1 - x_n, l_1 - l_n$)

1. Repeat $i := n$ to 1
2. $x_i := x'_{2-i \% 2} \% l_i$
3. $x'_{2-i \% 2} := (x'_{2-i \% 2} - x_i) / l_i$

3.3 Comparison of TMA and G2A for Matrix Operations

The TMA and G2A are both higher dimension arrays with different data layouts. In G2A, the array cells are organized into chunks according to the number of dimensions. For a TMA(n) $A[l_1][l_2] \dots [l_n]$ its equivalent G2A $A'[l_1][l_2]$ has the chunk size $|l_n \times l_{n-1}|$ and there are such $|l_1 \times l_2 \times \dots \times l_{n-2}|$ chunks exists. Each chunk is a two dimensional array of size $[l_n, l_{n-1}]$. Figure 3.4 shows the data layout separated into chunks for matrix- matrix addition/subtraction and multiplication for 4 dimensional matrices.

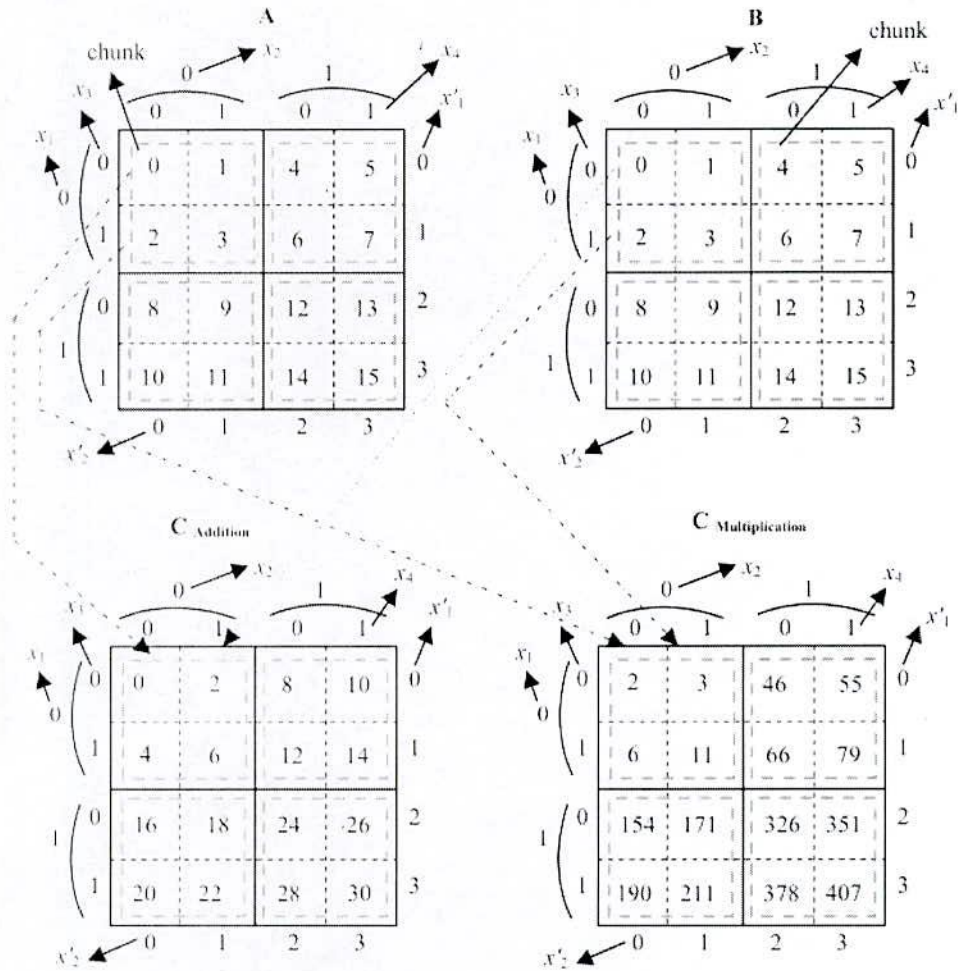


Figure 3.4 Addition and Multiplication of matricized TMA(4) of size $2 \times 2 \times 2 \times 2$

The G2A is organized as chunks according to d_4 and d_3 (i.e. x_4 and x_3). The chunk size is $|l_3 \times l_4|$ i.e. 2×2 and there are such $2 \times 2 = 4$ chunks exists. Fig. 3.4 shows the data layout for matrix-matrix addition and multiplication for $C_{\text{addition}} = A + B$ and $C_{\text{multiplication}} = A \times B$ where A and B are two matrices. The addition/subtraction and multiplication are performed by fixing rows for varying columns as done for a 2 dimensional matrix. This increases the cache hit rate of the processor because of the data locality [15][16]. In the following at first the algorithms for three dimensional matrixes are presented and then extend it to higher dimensional matrix representation.

3.3.1 Matrix-Matrix Addition/Subtraction Algorithms[2]

Let two three dimensional TMA A and B of size $[l_1, l_2, l_3]$. The resultant matrix, $C = A \pm B$ is down by fixing first dimensions for pointing the row and column (second dimension as

row and third dimension as column). Algorithm 3.3, 3.4 and 3.5 show the matrix-matrix addition/subtraction for TMA(3), TMA(n) and G2A respectively.

Algorithm 3.3:

matrix-matrix addition_TMA3

begin

for $x_1 = 0$ to (l_1-1) do

for $x_2 = 0$ to (l_2-1) do

for $x_3 = 0$ to (l_3-1) do

$$C[x_1][x_2][x_3] = A[x_1][x_2][x_3] + B[x_1][x_2][x_3];$$

End.

Algorithm 3.4:

matrix-matrix_addition_TMA_n

begin

for $x_1 = 0$ to (l_1-1) do

for $x_2 = 0$ to (l_2-1) do

.....

for $x_n = 0$ to (l_n-1) do

$$C[x_1][x_2] \dots [x_n] = A[x_1][x_2] \dots [x_n] + B[x_1][x_2] \dots [x_n];$$

End.

Algorithm 3.5:

matrix-matrix_addition_G2A

begin

for $x'_1 = 0$ to (l'_1-1) do

for $x'_2 = 0$ to (l'_2-1) do

$$C'[x'_1][x'_2] = A'[x'_1][x'_2] + B'[x'_1][x'_2];$$

End.

3.3.2 Matrix-Matrix Multiplication Algorithms

Let two three dimensional TMA **A** and **B** of size $[l_1, l_2, l_3]$. The resultant matrix, $C = A \times B$ is done by fixing first dimensions for pointing each 2-dimensional grid of row and column (second dimension as row and third dimension as column) where length of row and column are same *i.e.* $l_{n-1} = l_n$. Algorithm 3.6, 3.7 and 3.8 show the matrix-matrix multiplication for TMA(3), TMA(n) and G2A respectively.

Algorithm 3.6:

matrix-matrix multiplication_TMA_3

begin

 for $x_1 = 0$ to (l_1-1) do for $x_2 = 0$ to (l_2-1) do for $x_3 = 0$ to (l_3-1) do for $i = 0$ to (l_3-1) do

$$C[x_1][x_2][x_3] = C[x_1][x_2][x_3] + A[x_1][x_2][i] \times B[x_1][i][x_3];$$

End.

Algorithm 3.7:

matrix-matrix_multiplication_TMA_n

begin

 for $x_1 = 0$ to (l_1-1) do for $x_2 = 0$ to (l_2-1) do

.....

 for $x_n = 0$ to (l_n-1) do for $i = 0$ to (l_n-1) do

$$C[x_1][x_2] \dots [x_n] = C[x_1][x_2] \dots [x_n] + A[x_1][x_2] \dots [x_{n-1}][i] \times B[x_1][x_2] \dots [i][x_n];$$

End.

Algorithm 3.8:

matrix-matrix_multiplication_G2A_naive

begin

 for $x_1' = 0$ to $(l_1'-1)$ do

begin

$$u = x_1' - x_1' \% l_1'$$

 for $x_2' = 0$ to $(l_2'-1)$ do

begin

$$v = x_2' - x_2' \% l_2'$$

 for $i = 0$ to $(l-1)$ do

$$C[x_1'][x_2'] = C[x_1'][x_2'] + A[x_1'][v+i] \times B[u+i][x_2'];$$

end

end

End.

However, the performance of the naive algorithm (Algorithm 3.8) for $C=A' \times B'$ will be low because of the modulus operation for the calculation of u and v . The algorithm is revised for avoiding modulo operation as shown in Algorithm 3.9.

Algorithm 3.9:

matrix-matrix_multiplication_G2A

begin

$u = 0$

for $x_1' = 0$ to $(l_1' - 1)$ do

begin

$p = x_1' - u$

if $p = l$ then

$u = u + l$;

$v = 0$

for $x_2' = 0$ to $(l_2' - 1)$ do

begin

$q = x_2' - q$

if $q = l$ then

$v = v + l$

for $i = 0$ to $(l - 1)$ do

$C[x_1'][x_2'] = C[x_1'][x_2'] + A'[x_1'][v+i] \times B'[u+i][x_2']$

end

end

end.

The correctness of algorithm 3.9 is shown in Table 3.1 by showing the operation and values of different variables where A' and B' are two G2A of four-dimensional TMA having length 2 for each of four dimensions as shown in Figure 3.4.

A'		B'	
$x_1=0$	$x_2=0$	$x_1=0$	$x_2=0$
$x_3=0$	$x_4=0$	$x_3=0$	$x_4=0$
1	1	1	1
0	0	0	0
1	1	1	1
$x_2'=0$	$x_1'=0$	$x_2'=0$	$x_1'=0$
1	1	1	1
2	2	2	2
3	3	3	3

Table 3.1 Multiplication of matricized TMA(4) of size $2 \times 2 \times 2 \times 2$ as of Algorithm 3.9

x'_1	x'_2	u	v	$C'[x'_1][x'_2] = C'[x'_1][x'_2] + A'[x'_1][v+i] \times B'[u+i][x'_2]$
0	0	0	0	$0 \times 0 + 1 \times 2$
0	1	0	0	$0 \times 1 + 1 \times 3$
0	2	0	2	$4 \times 4 + 5 \times 6$
0	3	0	2	$4 \times 5 + 5 \times 7$
1	0	0	0	$2 \times 0 + 3 \times 2$
1	1	0	0	$2 \times 1 + 3 \times 3$
1	2	0	2	$6 \times 4 + 7 \times 6$
1	3	0	2	$6 \times 5 + 7 \times 7$
2	0	2	0	$8 \times 8 + 9 \times 10$
2	1	2	0	$8 \times 9 + 9 \times 11$
2	2	2	2	$12 \times 12 + 13 \times 14$
2	3	2	2	$12 \times 14 + 13 \times 15$
3	0	2	0	$10 \times 8 + 11 \times 10$
3	1	2	0	$10 \times 9 + 11 \times 11$
3	2	2	2	$14 \times 12 + 15 \times 14$
3	3	2	2	$14 \times 13 + 15 \times 15$

3.4 Theoretical Analysis

There are two aspects for performance improvement on matrix operation of proposed G2A (shown in Section 3.3) namely 1) Cost of index computation and 2) Cost of cache line accessed. The cost of index computation comprise of the cost of total number of addition and multiplication operations. Another aspect, the cost of cache line accessed is analyzed with the algorithm called *LoopCost(l)* proposed by Carr et al. [15][16]. The parameters are grouped as shown in Table 4.2. Some of these parameters are provided as input, while others are derived from the input parameters. All lengths or sizes are in bytes.

Table 3.2 Parameters for theoretical analysis

Parameter	Description
n	Number of dimension for both TMA & G2A
l_i	Length of dimension i ($2 \leq i \leq n$) for TMA. Consider $l_1 = l_2 = l_3 = \dots = l_n = l$ for all i . Hence size of the array or total array elements becomes l^n .
l'_1	Length of row for G2A; i.e. $l'_1 = l^k$ where $k = \lfloor n/2 \rfloor$
l'_2	Length of column for G2A; i.e. $l'_2 = l^k$
r	Size of cache line
α	Cost of a multiplication operation
β	Cost of an Addition/Subtraction operation
η	Improvement; $\eta = \left(1 - \frac{\text{Cost of G2A}}{\text{Cost of TMA}}\right) \times 100\%$

3.4.1 Cost of Index Computation

According to our algorithms described at section 3.3, the total cost of index computation for matrix-matrix addition and multiplication for different number of dimensions can be presented as

$$\text{Total Cost} = \text{Cost of Index Calculation} + \text{Cost of Array operation}$$

The analysis is performed on three TMA **A**, **B** and **C** as well as three G2A **A'**, **B'** and **C'**. The index computation is generalized with n number of dimensions though 3, 4 and 6 dimensions.

Matrix-Matrix Addition/Subtraction:

For matrix-matrix addition/subtraction operation, each element is accessed only once. An n -Dimensional TMA of length l for each dimension needs to compute l^n number of index for each of three matrixes as well as l^n number of addition/subtraction operations. Consider $C=A \pm B$ and $C'=A' \pm B'$ for three dimensional data having l^3 elements for each. The index computation function for TMA and G2A are f and f' respectively as below

$$f(x_1, x_2, x_3) = x_1 \times l \times l + x_2 \times l + x_3 \quad \text{and} \quad f'(x'_1, x'_2) = x'_1 \times l'_2 + x'_2$$

The function f requires two additions and three multiplications while f' requires one addition and one multiplication to compute each index. The G2A also requires another one multiplication to calculate $l'_1 = l \times l$. Thus the index computation costs $3(3\alpha + 2\beta) l^3$ and $3(\alpha + \beta) l^3 + \alpha$ for TMA and G2A respectively. But the total cost is the sum of index computation cost and cost of array operation. So,

$$\text{Total cost for operation on TMA} = 3(3\alpha + 2\beta) l^3 + l^3 \beta = (9\alpha + 7\beta) l^3 \text{ and}$$

$$\text{Total cost for operation of G2A} = 3(\alpha + \beta) l^3 + \alpha + l^3 \beta = \alpha + (3\alpha + 4\beta) l^3$$

Now, the improvement, η over schemes for three dimensional arrays can be calculated as follows.

$$\eta = \left(1 - \frac{\alpha + (3\alpha + 4\beta) l^3}{(9\alpha + 7\beta) l^3} \right) \times 100$$

As we know that the cost of multiplication is very high than that of addition ($\alpha \gg \beta$). If we ignore the α with respect to β then

$$\eta = \left(\frac{2}{3} - \frac{1}{9l^3} \right) \times 100$$

If the above analysis is considered for four dimensional array with below TMA index computation function

$$f(x_1, x_2, x_3, x_4) = x_1 \times l \times l \times l + x_2 \times l \times l + x_3 \times l + x_4$$

There are l^4 elements and each element requires three addition and six multiplication operations. The index computation function for any dimensional array is same as shown for three dimensional data. For four dimensional data equivalent G2A needs another multiplication operation to compute $l_2 = l \times l$. So, the index computation cost for total three array of TMA and G2A are $3(6\alpha + 3\beta) l^4$ and $3(\alpha + \beta) l^4 + 2\alpha$ respectively. The total cost will be the above cost plus $l^4 \beta$ for each case as shown below

$$\text{Total cost for operation on TMA} = 3(6\alpha + 3\beta) l^4 + l^4 \beta = (18\alpha + 10\beta) l^4 \text{ and}$$

$$\text{Total cost for operation of G2A} = 3(\alpha + \beta) l^4 + 2\alpha + l^4 \beta = 2\alpha + (3\alpha + 4\beta) l^4$$

So, the improvement rate over scheme for addition operation on four dimensional data,

$$\begin{aligned} \eta &= \left(1 - \frac{2\alpha + (3\alpha + 4\beta)l^4}{(18\alpha + 10\beta)l^4}\right) \times 100 \\ &= \left(\frac{5}{6} - \frac{1}{9l^4}\right) \times 100 \quad [\text{for } \alpha \gg \beta] \end{aligned}$$

Similarly, below TMA index computation function for six dimensional arrays have five additions and fifteen multiplications operation for each of total l^6 elements.

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = x_1 \times l \times l \times l \times l \times l + x_2 \times l \times l \times l \times l + x_3 \times l \times l \times l + x_4 \times l \times l + x_5 \times l + x_6$$

Equivalent G2A needs four multiplications to determine the lengths $l_1 = l \times l \times l$ and $l_2 = l \times l \times l$. So, the total cost for matrix-matrix addition for six dimensional data would be as below

$$\text{Total cost for operation on TMA} = 3(15\alpha + 5\beta) l^6 + l^6 \beta = (45\alpha + 16\beta) l^6 \text{ and}$$

$$\text{Total cost for operation of G2A} = 3(\alpha + \beta) l^6 + 4\alpha + l^6 \beta = 4\alpha + (3\alpha + 4\beta) l^6$$

So, the improvement rate for six dimensional data operation,

$$\eta = \left(1 - \frac{4\alpha + (3\alpha + 4\beta)l^6}{(45\alpha + 16\beta)l^6}\right) \times 100$$

$$= \left(\frac{14}{15} - \frac{4}{45l^6} \right) \times 100 \quad [\text{for } \alpha \gg \beta]$$

Now the generalization of above analysis for n -dimensional array with below TMA index computation function which has $(n-1)$ addition and $n(n-1)/2$ multiplication.

$$f(x_1, x_2, \dots, x_{n-1}, x_n) = x_1 \times l^{n-1} + x_2 \times l^{n-2} + \dots + x_{n-1} \times l + x_n$$

G2A index computation function is same for any number of dimensions as shown before except calculation of l^1 and l^2 which require $(n-2)$ multiplication. The total cost for n -dimensional data computation for both cases are below

$$\text{Total cost of TMA} = 3l^n \left(\frac{n(n-1)}{2} \alpha + (n-1)\beta \right) + \beta l^n \quad \text{and}$$

$$\text{Total cost of G2A} = 3(\alpha+\beta) l^n + (n-2)\alpha + l^n \beta = (n-2)\alpha + (3\alpha+4\beta) l^n$$

Hence, the improvement for addition operation considering $\alpha \gg \beta$,

$$\eta = \left(1 - \frac{2}{n(n-1)} - \frac{2(n-2)}{3n(n-1)l^n} \right) \times 100 \dots \dots \dots (3.1)$$

Matrix-Matrix Multiplication:

Matrix-matrix multiplication described in section 3.3.2 requires multiple accesses for same element. The index computation function for TMA and G2A are same as described for matrix-matrix addition for different number of dimensions. The multiplication operation for three dimensional array having length l of each dimension needs to access l^4 elements. The array operation needs another one addition and one multiplication of l^4 elements. So the total cost to compute $C = A \times B$ for three dimensional TMA are the sum of index computation cost and cost of array operation as

$$\text{Total cost of TMA} = 3(3\alpha+2\beta) l^4 + (\alpha+\beta)l^4 = (10\alpha+7\beta) l^4 \quad \text{and}$$

$$\text{Total cost of G2A} = 3(\alpha+\beta) l^4 + \alpha + (\alpha+\beta) l^4 = \alpha + 4(\alpha+\beta) l^4$$

Thus the improvement, η over schemes can be calculated as below

$$\begin{aligned} \eta &= \left(1 - \frac{\alpha + 4(\alpha + \beta)l^4}{(10\alpha + 7\beta)l^4} \right) \times 100 \\ &= \left(\frac{3}{5} - \frac{1}{10l^4} \right) \times 100 \quad [\alpha \gg \beta] \end{aligned}$$

Similarly, the total cost of TMA(4) and equivalent G2A, can be calculate as follows

$$\text{Total cost of TMA} = 3(6\alpha+3\beta) l^5 + (\alpha+\beta)l^5 = (19\alpha+10\beta) l^5 \quad \text{and}$$

$$\text{Total cost of G2A} = 3(\alpha+\beta)l^5 + 2\alpha + (\alpha+\beta)l^5 = 2\alpha + 4(\alpha+\beta)l^5$$

If we neglect the cost of addition with comparing the cost of multiplication then the improvement for four dimensional array computation,

$$\eta = \left(\frac{15}{19} - \frac{2}{19l^5} \right) \times 100$$

Again the consideration of six dimensional arrays for matrix multiplication then total cost of TMA and equivalent G2A would be as below

$$\text{Total cost of TMA} = 3(15\alpha+5\beta)l^7 + (\alpha+\beta)l^7 = (46\alpha+16\beta)l^7 \text{ and}$$

$$\text{Total cost of G2A} = 3(\alpha+\beta)l^7 + 4\alpha + (\alpha+\beta)l^7 = 4\alpha + 4(\alpha+\beta)l^7$$

Thus the improvement rate for six dimensional data considering $\alpha \gg \beta$ is

$$\eta = \left(\frac{21}{23} - \frac{2}{23l^7} \right) \times 100$$

Now the generalization of above analysis for n-dimensional data needs to access l^{n+1} element for each array. Thus the total cost for both TMA and G2A are as below

$$\text{Total cost of TMA} = 3 \left(\frac{n(n-1)}{2} \alpha + (n-1)\beta \right) l^{n+1} + (\alpha + \beta) l^{n+1} \text{ and}$$

$$\text{Total cost of G2A} = 3(\alpha+\beta)l^{n+1} + (n-2)\alpha + (\alpha+\beta)l^{n+1} = (n-2)\alpha + 4(\alpha+\beta)l^{n+1}$$

Hence the generalized improvement rate by ignoring cost of addition for matrix-matrix multiplication of n-dimensional data can be represent as below

$$\eta = \left(1 - \frac{8}{3n(n-1)+2} - \frac{2(n-2)}{\{3n(n-1)+2\}l^{n+1}} \right) \times 100 \dots \dots \dots (3.2)$$

Remarks 3.1: For large values of n and l the improvement rate η will increase. Hence the overall improvement will increase for higher dimensional array of large size for matrix-matrix addition or multiplication operation.

3.4.2 Cache Effect Analysis

It is well known that the compilers allocate memory sequentially. For a n -dimensional array, it is possible to access into factorial of n possible loop order. Among them, $(n-1)!$ number of loop order access memory randomly. We all know that random access of memory increase the cache miss rate. But this thesis desires lowest cache miss rate to

ensure higher efficiency and thus considering sequential access of memory. Hence, the loop order $\langle l_1, l_2, \dots, l_{n-1}, l_n \rangle$ is considered i.e. outer loop is managed by l_1 then l_2 and finally l_n .

Matrix-Matrix Addition/Subtraction:

There are 3 (three) different loop orders are possible for a TMA(3). We assume the loop order $\langle l_1, l_2, l_3 \rangle$ to ensure the most sequential access of the memory. As the Cache is partitioned into lines and, during data transfer, a whole line is read or written. If the cache line size is r then the number of cache line access using $LoopCost(l)[2][3]$ for matrix-matrix addition/ subtraction, $l^2 \left\lceil \frac{l}{r} \right\rceil$ (see algorithm 3.3.1) is same for both TMA(3) and its equivalent G2A i.e. no improve for G2A over TMA(3) for matrix-matrix addition/subtraction. For TMA(4), 4! different loop orders are possible. We assume the loop order $\langle l_1, l_2, l_3, l_4 \rangle$ to ensure the most sequential access of the memory. The number of cache line access using $LoopCost(l)$ for matrix-matrix addition/ subtraction for TMA(4) and equivalent G2A are $l^3 \left\lceil \frac{l}{r} \right\rceil$ & $l^2 \left\lceil \frac{l^2}{r} \right\rceil$ respectively (See algorithm 3.3.2). So, the improve rate for G2A over TMA(4) is

$$\eta = \left\{ 1 - \frac{\left\lceil \frac{l^2}{r} \right\rceil \times l^2}{\left\lceil \frac{l}{r} \right\rceil \times l^3} \right\} \times 100$$

For TMA(n), n! different loop orders are possible. We assume the loop order $\langle l_1, l_2, l_3, l_4, \dots, l_{n-1}, l_n \rangle$ to ensure most sequential access of the memory. The number of cache line access for matrix-matrix addition/ subtraction for TMA(n) and equivalent G2A are $l^{n-1} \left\lceil \frac{l}{r} \right\rceil$ & $l^{\lfloor n/2 \rfloor} \left\lceil \frac{l^{\lfloor n/2 \rfloor}}{r} \right\rceil$ respectively. So, the improvement for G2A over TMA(n) is

$$\eta = \left\{ 1 - \frac{\left\lceil \frac{l^{\lfloor n/2 \rfloor}}{r} \right\rceil \times l^{\lfloor n/2 \rfloor}}{\left\lceil \frac{l}{r} \right\rceil \times l^{n-1}} \right\} \times 100 \dots \dots \dots (3.3)$$

When l is divisible by r , the improvement is 0, that is, the number of cache line accessed for the G2A is the same as that of the TMA. When l is not divisible by r , the improvement is positive. If l is much larger than r , then $\eta \approx 0$.

Matrix-Matrix Multiplication:

If we consider most sequential access of memory i.e. x_3 is the innermost indices of TMA(3) then the *LoopCost*(l) of $A[x_1][x_2][x_3]$, $B[x_1][x_2][x_3]$ and $C[x_1][x_2][x_3]$ are $1 \times l^3$, $[l/r] \times l^3$ and $[l/r] \times l^3$ respectively. So the total loop cost of TMA(3), $2[l/r] \times l^3 + l^3$ is same for equivalent G2A i.e. no improve for G2A over TMA(3). The number of cache line accessed for $A[x_1][x_2][x_3][x_4]$, $B[x_1][x_2][x_3][x_4]$ and $C[x_1][x_2][x_3][x_4]$ of TMA(4) are $1 \times l^4$, $[l/r] \times l^4$ and $[l/r] \times l^4$ respectively. $1 \times l^3$, $[l^2/r] \times l^3$ and $[l^2/r] \times l^3$ are the number of cache line accessed for A', B' and C' respectively of equivalent G2A. The improvement for G2A over TMA(4) is

$$\eta = 1 - \frac{\left(\left(2 \left\lfloor \frac{l^2}{r} \right\rfloor + 1 \right) \times l^3 \right)}{\left(\left(2 \left\lfloor \frac{l}{r} \right\rfloor + 1 \right) \times l^4 \right)}$$

When l is divisible by r , the improvement is 0, and when l is not divisible by r , the improvement is

$$\eta = 1 - \frac{\left(\left\lfloor \frac{l^2}{r} \right\rfloor + 1 \right)}{\left(\left\lfloor \frac{l}{r} \right\rfloor + 1 \right) l}$$

Similarly, The number of cache line accessed for A, B and C of TMA(n) are $1 \times l^n$, $[l/r] \times l^n$ and $[l/r] \times l^n$ respectively. Equivalent G2A has cache line accessed for A', B' and C' are $1 \times l^{n/2+1}$, $\left\lfloor \frac{l^{n/2}}{r} \right\rfloor \times l^{n/2+1}$ and $\left\lfloor \frac{l^{n/2}}{r} \right\rfloor \times l^{n/2+1}$ respectively. So, the improvement for G2A over TMA(n) is

$$\eta = 1 - \frac{\left(2 \left\lfloor \frac{l^{n/2}}{r} \right\rfloor + 1 \right) \times l^{n/2+1}}{\left(2 \left\lfloor \frac{l}{r} \right\rfloor + 1 \right) \times l^n \times l^4}$$

Once again, if l is divisible by r then the improved rate is 0, that is, the number of cache line accessed for the G2A is the same as that of the TMA. When l is not divisible by r , the improvement is

$$\eta = 1 - \frac{\left(2 \left\lfloor \frac{l^{n/2}}{r} \right\rfloor + 1 \right)}{\left(2 \left\lfloor \frac{l}{r} \right\rfloor + 1 \right) \times l^{(n-2)/2}} \dots \dots \dots (3.4)$$

Remarks 3.2: If l is divisible by r then there is no improvement for G2A based algorithms over TMA based algorithms i.e, the number of cache line accessed for the G2A is the same as that of the TMA. When l is not divisible by r , the improvement is positive for G2A based algorithms.

3.5 Conclusion

This chapter explains proposed G2A scheme as the realization of the scheme based on three, four and six dimensional data. Finally the generalization scheme for n -dimensional data is described. The derivation of equations to calculate the parameter for generating G2A from its equivalent TMA are called forward mapping. Similarly, returning to its original TMA from G2A namely backward mapping is generalized. For both of mapping algorithms are provided. The performance improvement due to G2A structure is shown by matrix-matrix addition and multiplication operation. Algorithms for 3, 4 and 6 dimensional array are shown for both operations. Finally the generalized algorithms to support n dimensional data are analyzed. Lower cost of index computation and higher data locality show the improvement performance though details analysis. The matrix-matrix addition and multiplication operation described in this chapter works well for dense array. But when there are huge of empty cells in a multidimensional array then the schemes needs some revision. In this regards, generalized array storage scheme for sparse multidimensional data based on G2A is described in chapter IV.

CHAPTER IV

Generalized Compressed Row/Column Storage

4.1 Introduction

Multidimensional arrays are good storage for dense data but it shows bad performance for sparse datasets which wastes huge memory because of the empty cells in the array. Hence it is very hard to use in actual implementation. The sparsity problem becomes serious when the number of dimensions increases. This is because the number of all possible combinations of dimension values exponentially increases, whereas the number of actual data values would not increase at such a rate. Efficient storage schemes are required to store such sparse data for multidimensional array[2][5][35][40][42]. That's why most data scientists suggest efficient storage scheme on higher dimensional sparse array where the operations on stored data can be performed without returning to the original structure. But existing storage schemes are inefficient and clunky when the number of dimension becomes higher specially greater than four. The traditional CRS/CCS[5][27][34] or n -way tensor decomposition scheme requires n number of auxiliary one dimensional array/vector to store the indexes. The length of each vector is the number of non-zero element that exists in entire array. For example, to store a 4-dimensional array in CRS/CCS or 4-way tensor decomposition requires 4 vectors. When the number of dimension increases, those schemes become unusable due to low data sparsity and high space and time complexity. Hence the schemes are not effective enough for storing higher dimensional sparse array. This chapter proposes a storage scheme namely Generalized Compressed Row/Column Storage (GCRS/GCCS) based on the G2A representation described in chapter III.

4.2 Realization of GCRS/GCCS Scheme

GCRS/GCCS is a storage scheme to store n -dimensional sparse array. It is independent of number of array dimensions and requires only three vectors namely RO, CO and VL. VL is one-dimensional floating point array, RO and CO are two one-dimensional integer array.

The length of RO and VL are same and equal to the total number of non-zero value. Array VL stores the values of nonzero array elements. Array RO stores information of nonzero array elements of each G2A row (columns for CCS).

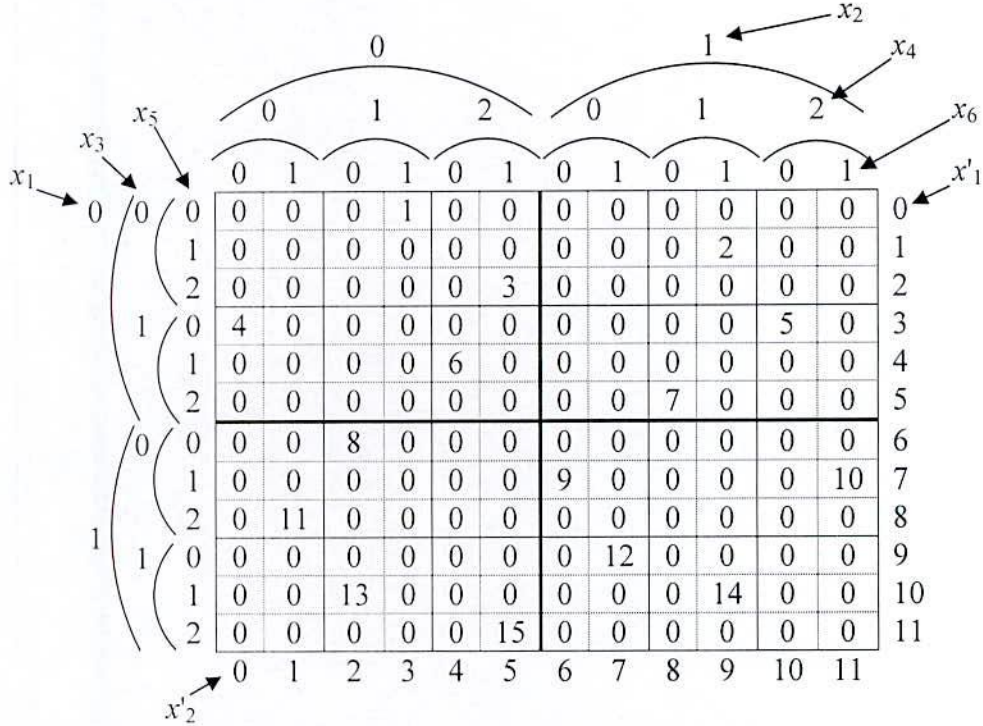


Figure 4.1 G2A representation of TMA(6) having size $2 \times 2 \times 2 \times 3 \times 3 \times 2$

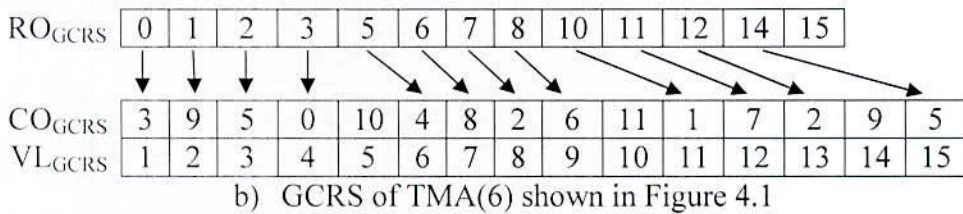
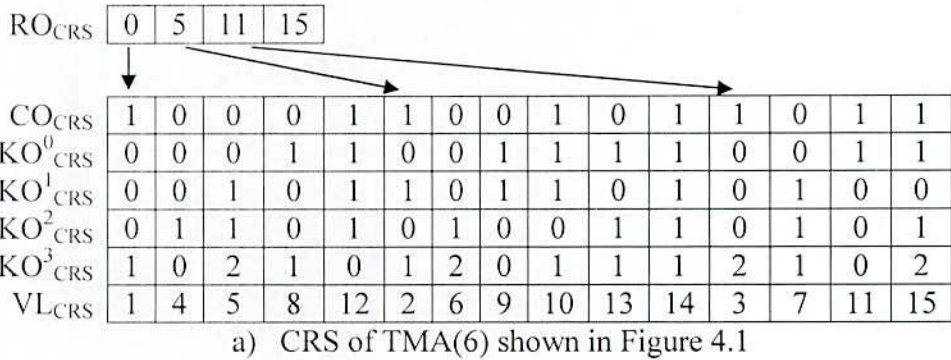


Figure 4.2 CRS and GCRS of TMA(6) having size $2 \times 2 \times 2 \times 3 \times 3 \times 2$

If the number of G2A rows is l'_1 for a n -dimensional TMA then RO contains l'_1+1 elements. RO[0] contains 0, RO[1] contains the summation of the number non zero elements in row 0. In general, RO[i] contains the number of nonzero elements in $(i-1)^{\text{th}}$ row of the array plus the contents of RO[i-1]. The number of non zero array elements in the i^{th} row of G2A can be obtained by subtracting the value of RO[i] from RO[i+1]. Array CO stores the G2A column (row for CCS) indices of nonzero array element of each row (columns for CCS). Figure 4.1 illustrate a TMA(6) of size $2 \times 2 \times 2 \times 3 \times 3 \times 2$ and its equivalent G2A representation. The CRS of this TMA(6) is shown in Figure 4.2-a) which requires to store total of 79(seventy nine) indexes and 15(fifteen) value itself. The GCRS of same TMA(6) is shown in Figure 4.2-b) which requires to store total of only 28(twenty eight) indexes.

4.2.1 Generating of GCRS/GCCS File

The generation of GCRS/GCCS can be done from G2A which is a row-column representation of TMA(n) with only three one-dimensional array RO, CO and VL. But we cannot neglect the cost of matricization of higher dimensional array. So an alternate solution can be the generation of GCRS/GCCS from TMA(n) directly. In this case G2A is logical/abstraction and there is no need to convert the TMA(n) into G2A. To do this, it needs to transform the loop order. The loop order is such that odd dimensions are accessed first then access the even dimensions i.e. loop order $\langle l_1, l_3, l_5, \dots, l_2, l_4, l_6, \dots \rangle$. So, the generation of GCRS/GCCS can be done in two ways:

1. First, convert the TMA(n) into G2A. Then store the converted G2A according existing CRS/CCS scheme described in chapter II.
2. Generate GCRS/GCCS directly from TMA(n) though loop transformation.

The rest of the storage scheme is described based on loop transformation technique. Consider a three-dimensional array of Figure 4.3 having length $[l_1, l_2, l_3] \approx [2, 3, 3]$. In G2A, odd dimensions contribute for row-direction and even dimensions contribute for column direction. Thus, the length of RO for GCRS, $l = 1 + l_1 \times l_3 = 1 + 2 \times 3 = 7$ and for GCCS, $l = 1 + l_2 = 1 + 3 = 4$. First element of RO is always zero and all elements are initialized with 0 at the beginning. Table 4.1 shows the generation of GCRS where loop order to read TMA(3) is $\langle l_1, l_3, l_2 \rangle$. If a non-zero element is found then its corresponding G2A index $\langle x'_1, x'_2 \rangle$ is calculated which is described in Chapter 3.

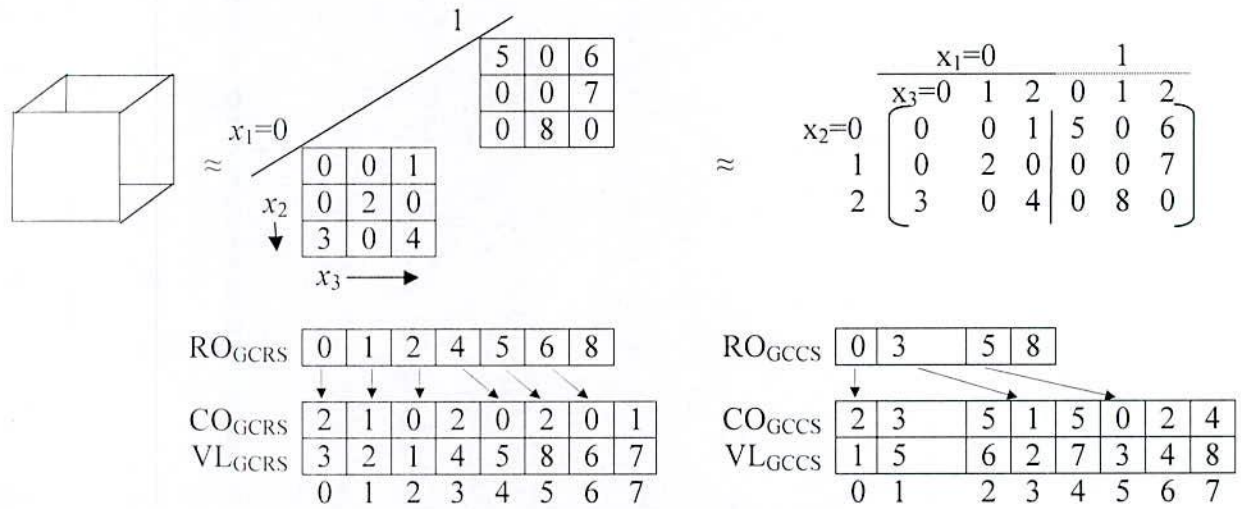
Figure 4.3 GCRS/GCCS from TMA(3) of size $2 \times 3 \times 3$

Table 4.1 GENERATION OF GCRS FROM TMA(3)

Non-zero Value	TMA Index	G2A Index	Generation of GCRS (loop order $\langle l_1, l_3, l_2 \rangle$)		
	$\langle x_1, x_2, x_3 \rangle$	$\langle x'_1, x'_2 \rangle$	VL	CO	RO
3	0, 2, 0	0, 2	Insert 3	Insert 2	[0] ++
2	0, 1, 1	1, 1	Insert 2	Insert 1	[1] ++
1	0, 0, 2	2, 0	Insert 1	Insert 0	[2] ++
4	0, 2, 2	2, 2	Insert 4	Insert 2	[2] ++
5	1, 0, 0	3, 0	Insert 5	Insert 0	[3] ++
8	1, 2, 1	4, 2	Insert 8	Insert 2	[4] ++
6	1, 0, 2	5, 0	Insert 6	Insert 0	[5] ++
7	1, 1, 2	5, 1	Insert 7	Insert 1	[5] ++

Then insert the non-zero value into VL_{GCRS} , x'_2 into CO_{GCRS} and the value of $RO_{GCRS}[x'_1]$ is incremented by 1. Here, insert means adding the value at the end of the array. Finally, the cumulated some of RO_{GCRS} values are calculated (i.e. $RO[i+1] = R[i] + RO[i+1]$ where $1 \leq i \leq l$). To store this TMA according CRS/CCS scheme, it require to store total 28 elements (Figure 2.8) while GCCS scheme needs only 20 elements to store as of Figure 4.1. Similarly, Table 4.2 shows the generation of GCCS where loop order to read TMA(3) is $\langle l_2, l_1, l_3 \rangle$. For a non-zero element, corresponding G2A index $\langle x'_1, x'_2 \rangle$ is calculated. Then push the non-zero value into VL_{GCCS} , x'_1 into CO_{GCCS} and the value of $RO_{GCCS}[x'_2]$ is incremented by 1.

Table 4.2 GENERATION OF GCCS FROM TMA(3)

Non-zero Value	TMA Index	G2A Index	Generation of GCRS		
	$\langle x_1, x_2, x_3 \rangle$	$\langle x'_1, x'_2 \rangle$	VL	CO	RO
1	0, 0, 2	2, 0	Insert 1	Insert 2	[0] ++
5	1, 0, 0	3, 0	Insert 5	Insert 3	[0] ++
6	1, 0, 2	5, 0	Insert 6	Insert 5	[0] ++
2	0, 1, 1	1, 1	Insert 2	Insert 1	[1] ++
7	1, 1, 2	5, 1	Insert 7	Insert 5	[1] ++
3	0, 2, 0	0, 2	Insert 3	Insert 0	[2] ++
4	0, 2, 2	2, 2	Insert 4	Insert 2	[2] ++
8	1, 2, 1	4, 2	Insert 8	Insert 4	[2] ++

If array dimension is n is even for GCRS scheme, then the length, l of RO can be found as follows:

$$l = 1 + \prod_{i=1}^{\frac{n+1}{2}} l_{2i-1} \dots \dots \dots (4.1)$$

The number of non-zero element(s) in i 'th Row/Column can be found by $RO[i1]-RO[i-1]$. Its not necessary to convert the whole n -dimensional array into two-dimensional array, but G2A's index computation function returns two-dimensional equivalent row and column indices. For each non-zero element, its TMA equivalent G2A row and column index is calculated. Then push the non-zero value into VL, push the column/row index into CO and increment the RO pointed by row/column index. It is noted that the GCRS/GCCS affects by odd/even number of dimensions which may be described as

1. If n is odd and scheme is GCRS then CO stores row index and increment RO value pointed by column index
2. If n is odd and scheme is GCCS then CO store column index and increment RO value pointed by row index
3. If n is even and scheme is GCRS then CO stores column index and increment RO value pointed by row index
4. If n is even and scheme is GCCS then CO store row index and increment RO value pointed by column index

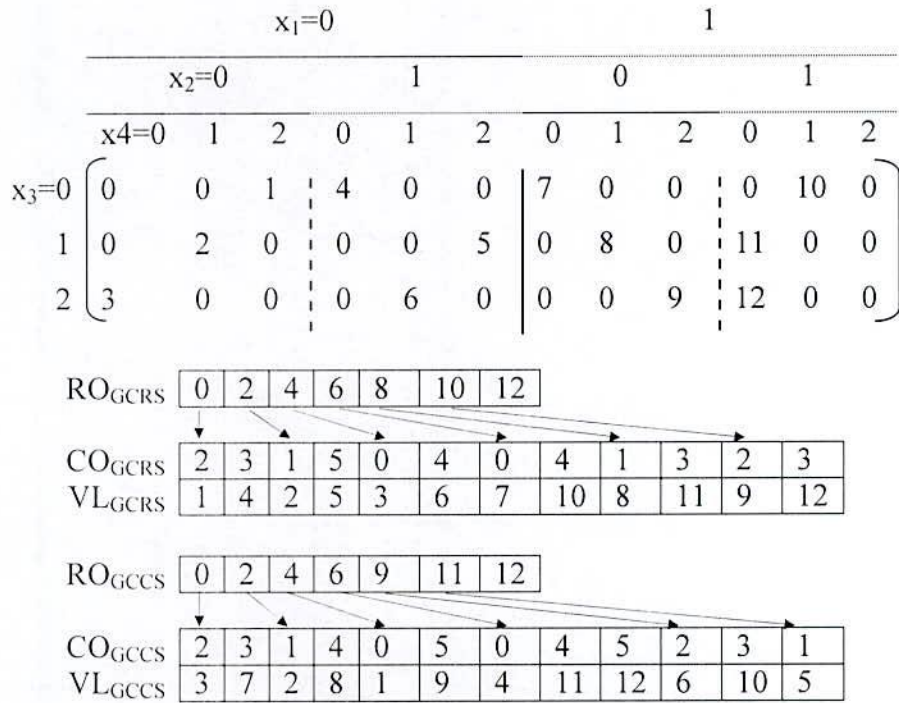


Figure 4.4 GCRS/GCCS of TMA(4)

Table 4.3 GENERATION OF GCRS FROM TMA(4)

Non-zero Value	TMA Index	G2A Index	Generation of GCRS		
	$\langle x_1, x_2, x_3, x_4 \rangle$	$\langle x'_1, x'_2 \rangle$	VL	CO	RO
1	0, 0, 0, 2	0, 2	Insert 1	Insert 2	[0] ++
4	0, 1, 0, 0	0, 3	Insert 4	Insert 3	[0] ++
2	0, 0, 1, 1	1, 1	Insert 2	Insert 1	[1] ++
5	0, 1, 1, 2	1, 5	Insert 5	Insert 5	[1] ++
3	0, 0, 2, 0	2, 0	Insert 3	Insert 0	[2] ++
6	0, 1, 2, 1	2, 4	Insert 6	Insert 4	[2] ++
7	1, 0, 0, 0	3, 0	Insert 7	Insert 0	[3] ++
10	1, 1, 0, 1	3, 4	Insert 10	Insert 4	[3] ++
8	1, 0, 1, 1	4, 1	Insert 8	Insert 1	[4] ++
11	1, 1, 1, 0	4, 3	Insert 11	Insert 3	[4] ++
9	1, 0, 2, 2	5, 2	Insert 9	Insert 2	[5] ++
12	1, 1, 2, 0	5, 3	Insert 12	Insert 3	[5] ++

Figure 4.4 show a four-dimensional array having length $[l_1, l_2, l_3, l_4] \approx [2, 2, 3, 3]$. In GCRS scheme, the length of RO, $l = 1 + l_1 \times l_3 = 1 + 2 \times 3 = 7$ and for GCRS, $l = 1 + l_2 \times l_4 = 7$. The generation of GCRS/GCCS is shown in Table 4.3 where RO is finally recalculated as earlier. It needs to store only $(1+6) + 2 \times 12 = 31$ elements in GCRS/GCCS

which is 52 in CRS/CCS scheme. From the above discussion, it is clearly visible that the storage in GCRS/GCCS is improved than CRS/CCS. The improvement will be superior with the increase of number of dimensions.

4.3 Operations on GCRS/GCCS

Matrix-matrix addition/subtraction and multiplication is the basic operation for most of the computing techniques and it is shown that the efficiency GCRS/GCCS scheme with matrix-matrix addition/subtraction and multiplication operation on stored data. It is well known that the operation on two similar arrays is possible. G2A is a two-dimensional representation of higher dimensional array. So, two-dimensional array/matrix operation is possible on G2A. Similarly, GCRS/GCCS is the storage scheme for higher-dimensional sparse array and matrix operations are also possible on stored data. To do this, it is not necessary to be same length CO and VL for both GCRS but same length of RO is essential. For the simplification of algorithms and representation, let us consider equal length for each dimensions in both matrixes (i.e. length of each dimension in array A is same with respective dimension of array B).

4.3.1 Matrix-Matrix Addition

Let two n dimensional array, \mathbf{A} and \mathbf{B} having same length $[l_1, l_2, \dots, l_{n-1}, l_n]$. If \mathbf{A} and \mathbf{B} are stored according GCRS or GCCS, then six vectors $RO_A, CO_A, VL_A, RO_B, CO_B$ and VL_B would be generated. The length of vector RO_A and RO_B, l_{RO} would be same. The number of non-zero elements in array \mathbf{A} is l_A and equal to the length of vectors CO_A and VL_A . The length of vector CO_B and VL_B is l_B . It is not necessary to be $l_A = l_B$ and $CO_A = CO_B$. If the resultant GCRS/GCCS is \mathbf{C} then it would be, $\mathbf{C} = \mathbf{A} + \mathbf{B}$. The algorithm to calculate the addition can be done as below algorithm.

Algorithm: Addition of two n -dimensional arrays stored as GCRS/GCCS scheme

GCRS/GCCS-addition ($l_{RO}, RO_A, CO_A, VL_A, RO_B, CO_B, VL_B$)

1. Initialize $k := 0, RO[0] := 0$
2. Repeat $i = 0$ to $(l_{RO}-2)$
3. $RL_A = RO_A[i+1] - RO_A[i], RL_B = RO_B[i+1] - RO_B[i]$
4. $RP_A = RO_A[i], RP_B = RO_B[i],$
5. Repeat $j = 0$ to $(l_{RO}-2)$

6. $SUM := 0$
7. *if* ($CO_A[RP_A] = CO_B[RP_B]$ and $CO_A[RP_A] = j$ and $RL_A > 0$ and $RL_B > 0$)
8. $SUM = VL_A[RP_A] + VL_B[RP_B], RP_A++, RP_B++, RL_A--, RL_B--$
9. *else if* ($CO_A[RP_A] = j$ and $RL_A > 0$)
10. $SUM = VL_A[RP_A], RP_A++, RL_A--$
11. *else if* ($CO_B[RP_B] = j$ and $RL_B > 0$)
12. $SUM := VL_B[RP_B], RP_B++, RL_B--$
13. *if* ($SUM \neq 0$)
14. $RO[i+1]++, CO[k] = j, VL[k] = SUM, k++$
15. $RO[i+1] = RO[i+1] + RO[i]$

Figure 4.5 illustrates the addition of two GCRS where associated pointer and length are shown

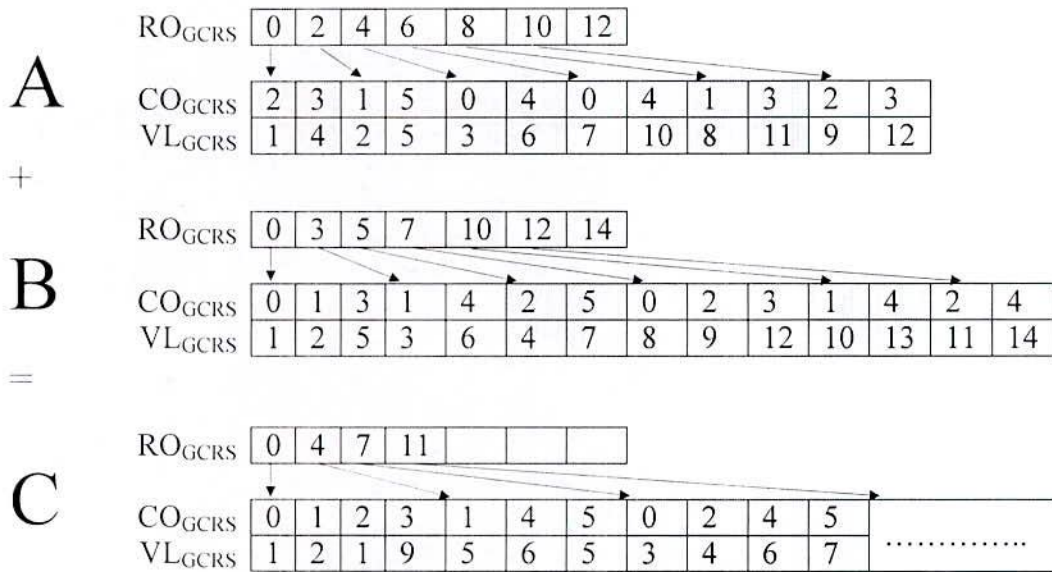


Figure 4.5 Addition of Two GCRS/GCCS

The GCRS/GCCS-addition algorithm picks the elements row/collumn-wise from array **A** and **B**. The number of element to be picked for *i*'th row addition is calculated as:

$$\mathbf{A}: RL_A = RO_A[i+1] - RO_A[i] \text{ and } \mathbf{B}: RL_B = RO_B[i+1] - RO_B[i]$$

The value of $RO_A[i]$ and $RO_B[i]$ points the *i*'th row first element of **A** and **B** respectively. After pointing and fixing the number of total elements in row-wise (column-wise in GCCS), there are three conditions to be checked for defining result for *j*'th column as below

- i. is the column value same as pointed element of CO_A AND is pointed CO_A and CO_B same
 - addition is done for VL_A and VL_B
 - both pointers are incremented
- ii. is the column value same as pointed element of CO_A
 - add new non-zero element of VL_A in result
 - pointer of A is incremented
- iii. is the column value same as pointed element of CO_B
 - add new non-zero element of VL_B in result
 - pointer of B is incremented

4.3.2 Matrix-Matrix Multiplication

For the multiplication of two matrixes, it should be similar (i.e. the number of dimensions and the length of each dimensions should be same). Let two n dimensional array, \mathbf{A} and \mathbf{B} having same length $[l_1, l_2, \dots, l_{n-1}, l_n]$. We know the elements of \mathbf{A} and \mathbf{B} are accessed row-wise and column-wise respectively. So, it needs to generate GCRS of array \mathbf{A} and GCCS of array \mathbf{B} for multiplication for improving the data locality. If the resultant multidimensional array is \mathbf{C} then

$$C = A \times B \Rightarrow C_{GCRS} = A_{GCRS} \times B_{GCRS} \text{ or } C_{GCCS} = A_{GCCS} \times B_{GCCS}$$

The only condition for multiplication is the same length of vector RO_A and RO_B . The algorithm to calculate the multiplication is shown below.

Algorithm: Multiplication of two n-dimensional arrays stored as GCRS/GCCS scheme

GCRS/GCCS-multi ($l, l_{RO}, RO_A, CO_A, VL_A, RO_B, CO_B, VL_B$)

1. Initialize $k := 0, RO[0] := 0$
2. Repeat $i = 0$ to $(l_{RO}-2)$
3. $RL_A = RO_A[i+1] - RO_A[i], RP_A = RO_A[i]$
4. $m = i - i \% l$
5. Repeat $j = 0$ to $(l_{RO}-2)$
6. $MUL := 0, n = j - j \% l$

7. $CL_B = RO_B[i+1] - RO_B[i], CP_B = RO_B[i]$
8. Repeat $p = 0$ to $(RL_A - 1)$
9. $if(CO_A[RP_A + p] \geq n \text{ AND } CO_A[RP_A + p] < n + l)$
10. Repeat $q = 0$ to $(CL_B - 1)$
11. $if(CO_B[CP_B + q] \geq m \text{ AND } CO_B[CP_B + q] < m + l)$
12. $if(CO_A[RP_A + p] = CO_B[CP_B + q] \text{ OR } CO_A[RP_A + p] \% l = CO_B[CP_B + q] \% l)$
13. $MUL += VL_A[RP_A + p] * VL_B[CP_B + q]$
14. $if(MUL \neq 0)$
15. $RO[i+1]++, CO[k] = j, VL[k] = MUL, k++$
16. $RO[i+1] = RO[i+1] + RO[i]$

GCRS/GCCS-multi algorithm takes GCRS of **A** and GCCS of **B** as input as shown in Figure 4.6.

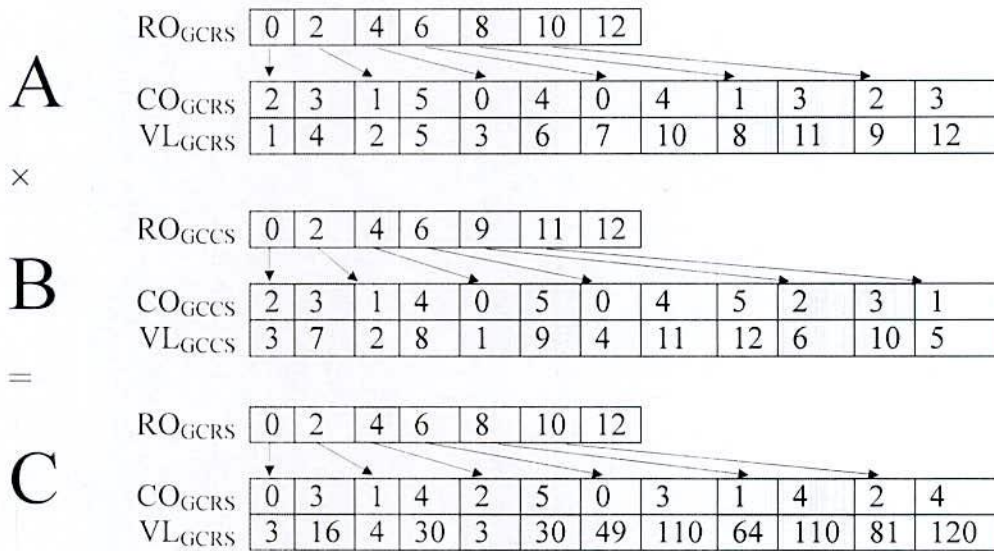


Figure 4.6 Multiplication of Two GCRS/GCCS

To determine result for element (i, j) , i 'th row from **A** and j 'th column from **B** is chosen. A partition for specific number of element for a specific chunk is generated with variable m and n when scanning the elements. After defining the partition, checking is done for equivalent CO_A and CO_B . If the result is non-zero then it is used for constructing resultant GCRS.

4.4 Theoretical Analysis

The cost model for the compression scheme is developed in this section. Theoretical evaluation is compared with the experimental implementation of Chapter V. Some definition is important to proceed for detail of analysis.

Density of array, ρ determines the degree of data sparsity of an array. It is the ratio of number of non-empty elements with total number of array elements. The maximum value of data density is one. We can right data density as below

$$\rho = \frac{\text{Total number of non-zero elements}}{\text{Total number of elements}}$$

The compression ratio, η is the ratio of the total memory required after compress an array and the actual size of the array is defined as compression ratio. It can represented as follows

$$\eta = \frac{\text{Memory required to store after compression}}{\text{Total size of uncompressed array}}$$

Range of usability of a compression scheme is defined as the maximum range of data density up to which the compression ratio is less than 1.

Improvement over scheme, t : It is the ratio of memory required to store after compression in CRS scheme and GCRS scheme. It is desired to be greater than 1. Formally it can be expressed as

$$t = \frac{\text{Compression Ratio for CRS/CCS scheme}}{\text{Compression Ratio for GCRS/GCCS scheme}} = \frac{\eta_{\text{CRS/CCS}}}{\eta_{\text{GCRS/GCCS}}}$$

This section shows the space requirement and hence the compression ratio for GCRS/GCCS scheme based on G2A. The cost model for the range of usability and improvement over schemes for CRS and GCRS is shown in section 4.2 and 4.3. Some parameters are provided as input and some are derived which is grouped as shown in table 4.3. All the lengths and size are in bytes.

Table 4.4 DIFFERENT PARAMETERS FOR THEORITICAL EVALUATION OF GCRS/GCCS

Parameter	Description
n	Number of dimensions
l_i	Length of dimension i
l	Length of each dimension (for simplification $l_1=l_2=l_3=\dots=l$)
l^n	Total number of elements

ρ	Data density or sparse ratio of array
ρl^n	Total number of Non-zero elements
l_1'	Length of G2A Row where $l_1' = l^{(n+1)/2}$
l_2'	Length of G2A Column where $l_2' = l^{n/2}$
γ	Size of elements for storing index
δ	Size of array element
S_{CRS}	Memory required to store in CRS/CCS scheme
S_{GCRS}	Memory required to store in GCRS/GCCS scheme

4.4.1 Space Requirement of CRS/CCS and GCRS/GCCS

Consider a TMA of n dimension having sparse ratio ρ and the length of each dimension is l . The storage requirement is the sum of storage of indexes and the storage of value itself.

Compression Ratio:

Cost of storing indexes in CRS/CCS scheme is the sum of one vector RO having length $l+1$, one vector CO having length ρl^n and $n-2$ number of KO vector having length ρl^n for each. The non-zero values needs storage of $\rho l^n \delta$. So, total storage required for CRS/CCS, $S_{CRS/CCS}$ can be shown as below:

$$\begin{aligned}
 S_{CRS/CCS} &= \text{Space for RO} + \text{Space for CO} + \text{Space for VL} + \text{Space for } (n-2) \text{ number of KO} \\
 &= (l+1)\gamma + \rho l^n \gamma + \rho l^n \delta + (n-2)\rho l^n \gamma \\
 &= \{ l+1 + \rho l^n + (n-2)\rho l^n \} \gamma + \rho l^n \delta \\
 &= \{ l+1 + (n-1)\rho l^n \} \gamma + \rho l^n \delta \dots\dots\dots (4.2)
 \end{aligned}$$

So, the compression ratio for CRS/CCS,

$$\begin{aligned}
 \eta &= \frac{\text{Memory required to store after compression}}{\text{Total size of uncompressed array}} \\
 &= \frac{\{ l+1+(n-1)\rho l^n \} \gamma + \rho l^n \delta}{\delta l^n} \\
 &= \frac{l+1}{l^n} + (n-1)\rho + \rho \quad [\text{if } \gamma = \delta] \\
 &= \frac{l+1}{l^n} + n\rho \\
 &= n\rho \quad [\text{for large value of } l \text{ and } n, \frac{l+1}{l^n} \approx 0]
 \end{aligned}$$

In GCRS/GCCS scheme, there is no need to store any KO index but increased amount of index at RO vector which is equivalent to $1+l^{(n+1)/2}$. Thus the total storage required for GCRS/GCCS, $S_{GCRS/GCCS}$ can be shown as below:

$$\begin{aligned}
 S_{GCRS/GCCS} &= \text{Space for RO} + \text{Space for CO} + \text{Space for VL} \\
 &= \{1+l^{(n+1)/2}\} \gamma + \rho l^n \gamma + \rho l^n \delta \\
 &= \{1+l^{(n+1)/2} + \rho l^n\} \gamma + \rho l^n \delta \dots\dots\dots (4.3)
 \end{aligned}$$

So, the compression ratio,

$$\begin{aligned}
 \eta &= \frac{\text{Memory required to store after compression}}{\text{Total size of uncompressed array}} \\
 &= \frac{\{1+l^{\frac{n+1}{2}} + \rho l^n\} \gamma + \rho l^n \delta}{\delta l^n} \\
 &= \frac{1+l^{\frac{n+1}{2}}}{l^n} + 2\rho \quad [\text{if } \gamma = \delta] \\
 &= 2\rho \quad [\text{for large value of } l \text{ and } n, \frac{1+l^{\frac{n+1}{2}}}{l^n} \approx 0]
 \end{aligned}$$

So, the storage improvement in GCRS/GCCS is linear and independent of data sparsity. The improvement will be four times for eight-dimensional array and nineteen times for thirty-eight dimensional array storage. Now, the ratio of improvement over schemes can be defined as the ratio of compression. The improvement over scheme, t can be as follows

$$\begin{aligned}
 t &= \text{Compression Ratio of CRS/CCS} \div \text{Compression Ratio of GCRS/GCCS} \\
 &= n\rho \div 2\rho \\
 &= n/2 \dots\dots\dots (4.4)
 \end{aligned}$$

Range of Usability:

The range of usability of a compression scheme is defined as the maximum range of sparse ratio up to which the size of the compressed array is less than that of the size of the original array. So, the range of usability for CRS/CCS scheme can be defined as follows:

$$\begin{aligned}
 &\{l+1 + (n-1) \rho l^n\} \gamma + \rho l^n \delta \leq l^n \delta \\
 \Rightarrow &\rho \leq \{l^n \delta - (l+1) \gamma\} / (n l^n \gamma - l^n \gamma + l^n \delta) \\
 \Rightarrow &\rho \leq \delta / \{(n-1) \gamma + \delta\} \quad [\text{Let } l^n \delta - (l+1) \gamma \approx l^n \delta] \\
 \Rightarrow &\rho \leq 1/n \quad [\text{if } \gamma=\delta; \text{ for worst case}]
 \end{aligned}$$

It shows that the range of usability of CRS/CCS is inversely proportional with the number of data dimensions (i.e. range of usability decrease with the increase of data dimensions). Similarly, the range of usability for GCRS/GCCS scheme can be defined as follows:

$$\begin{aligned}
 & (\rho l^n + l^{(n+1)/2} + 1) \gamma + \rho l^n \delta \leq l^n \delta \\
 \Rightarrow & \rho \leq \{ l^n \delta - (l^{(n+1)/2} + 1) \gamma \} / l^n (\gamma + \delta) \\
 \Rightarrow & \rho \leq \delta / (\gamma + \delta) \quad [l^n \delta - (l^{n/2} + 1) \gamma \approx l^n \delta] \\
 \Rightarrow & \rho \leq 1/2 \quad [\text{if } \gamma = \delta, \text{ for worst case}]
 \end{aligned}$$

Hence range of usability of GCRS/GCCS is minimum 50% and independent of number of data dimensions. So, the scheme can also be usable for a wide range of dense data.

4.5 Conclusion

This chapter described GCRS/GCCS scheme to store an n -dimensional sparse array. Total memory required to store an array depends on data density and independent of number of data dimensions. The operations on stored data are also shown by matrix operation. Algorithms ensure the matrix-matrix addition and multiplication operation on stored data. The experimental result complying with the theoretical analysis shown in this chapter is described in Chapter V.

CHAPTER V

Experimental Results

5.1 Introduction

This chapter simulates the algorithms for matrix-matrix addition and multiplication operation for both TMA and G2A as described in chapter III. This chapter also shows the experimental results to store sparse array of different dimensions in CRS/CCS and GCRS/GCCS schemes of chapter IV.

5.2 Experimental Setup

The construction of prototype system for G2A operations, GCRS/GCCS conversion and GCRS/GCCS operations are done on a machine having the following specification.

Table 5.1 EXPERIMENTAL SETUP

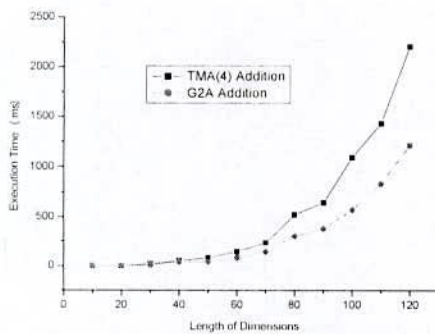
Parameter	Specification
Processor	Intel(R) Xeon(R) E5620
No. of Processor	8
Clock Speed	2.4 GHz
Cache Memory	1406 MB
RAM	32 GB
HDD	2.0 TB
Operating System	Linux (Debian 8.2)
Compiler	GCC
Compiler Optimization	None

5.3 Experimental Results for G2A Operations

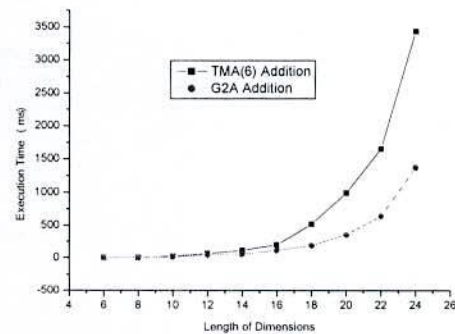
The array size is set from 10 to 120, 6 to 24 and 4 to 11 for each length of dimension and number of dimension 4, 6 and 8 respectively in both matrix-matrix addition/subtraction and matrix-matrix multiplication. The experimental result considered at traditional row major order looping for both addition and multiplication. Figure 5.1 shows the execution

time in milliseconds of algorithms for the matrix-matrix addition on the TMA and G2A based algorithms(see chapter III). Experimental results show that execution time is less for G2A based algorithms than TMA based algorithms. This is because the lower index computation cost and higher data locality. The improvement due to index computation for matrix-matrix addition of TMA(n) and G2A as shown in equation 3.1 which shows that if n and l increase then the improvement η will increase which supports remarks 3.1.

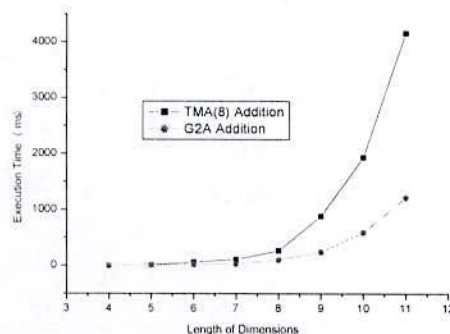
The improvement due to data locality of TMA(n) and G2A for matrix-matrix addition algorithm as derived in equation 3.3. Hence TMA based algorithm has higher cache miss rate than that of G2A based algorithms. The cache misses has direct influence to the performance because the processor needs to wait for the next data to be fetched from the next cache level or from the main memory. Even a single cache miss can degrade the performance as processor speed outperforms the memory speed. On the other hand G2A based algorithm improves the data locality that minimizes the cache miss rates. Caches take advantage of data locality in programs.



(a) Addition TMA(4) and G2A

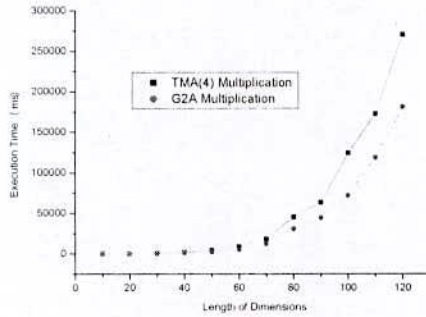


(b) Addition TMA(6) and G2A

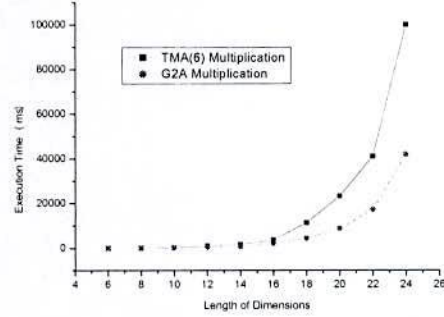


(c) Addition TMA(8) and G2A

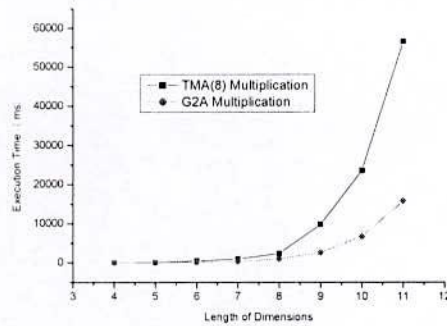
Figure 5.1 Experimental results for matrix-matrix addition



(a) Multiplication TMA(4) and G2A



(b) Multiplication TMA(6) and G2A



(c) Multiplication TMA(8) and G2A

Figure 5.2 Experimental performance for matrix-matrix multiplication

Figure 5.2 shows the improved performance for matrix-matrix multiplication of TMA and equivalent G2A. As of our analysis the improvement is for lower index computation cost shown in equation 3.2 and higher data locality shown in equation 3.4.

5.4 Experimental Results for generation of CRS/CCS and GCRS/GCCS

The input is taken from secondary memory and the output is also written into secondary memory. The number of dimensions is set 4, 8, 16 and 32. The length of each dimension was set such that the input file is less than or equal to 1 TB. The sparse ratio is set 0.01 to 0.15 and $\alpha=\beta=4$ for all cases.

5.4.1 Time Requirement

Figure 5.3 show the test results for time requirement for constructing of CRS and GCRS scheme for $n= 4, 8, 16$ and 32. This section only shows the experimental result for CRS and GCRS (row major order). It is clear that storing time increases for both CRS and GCRS when number of non-zero elements increase.

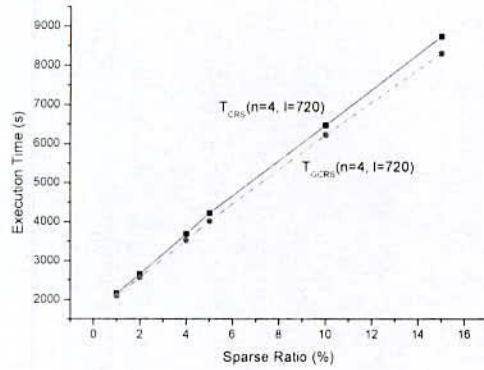
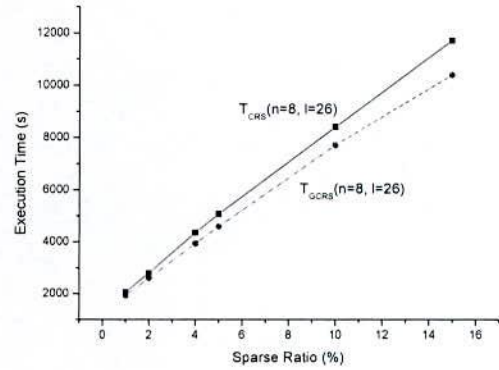
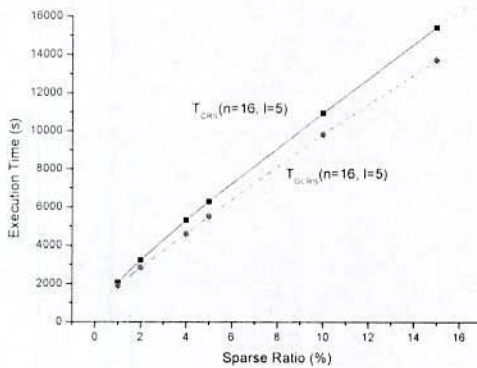
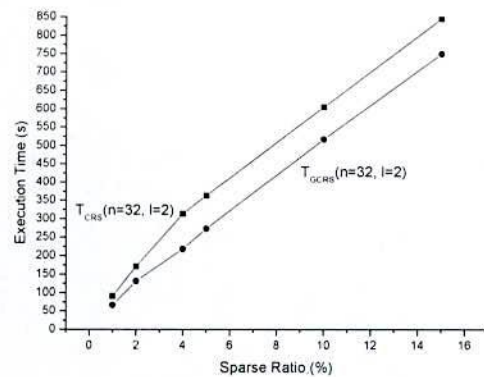
a) CRS and GCRS for $n=4$ and $l=720$ b) CRS and GCRS for $n=8$ and $l=26$ c) CRS and GCRS for $n=16$ and $l=5$ d) CRS and GCRS for $n=32$ and $l=2$

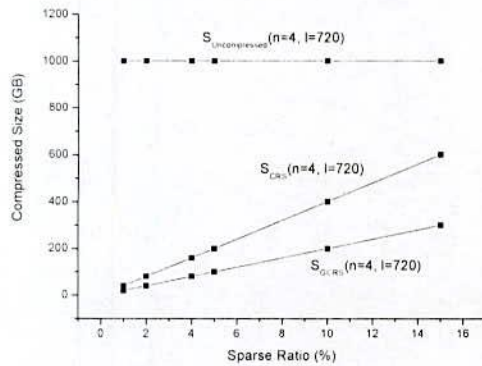
Figure 5.3 Time requirement for creating CRS and GCRS of different number of dimensions

Most of the required time is the scanning time or conversion time and index computation time is negligible to scanning time. Index computation cost increases with increase number of dimension (equation 2.4). That is why it shows very slow improvement of time requirement with increased dimensions. It is well known that, for TMA(n), it can be possible $n!$ loop order for scanning array and ordering of loops highly affect the performance. But in this experiment, for both case, our loop order was such that most sequential memory access is ensured i.e. (x_1, x_2, \dots, x_n) order which ensure most outmost loop is x_1 and most innermost loop is x_n .

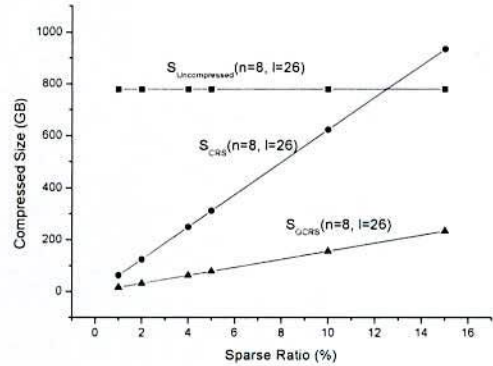
5.4.2 Space Requirement

Figure 5.4 shows the total size of stored arrays for 4-D, 8-D, 16-D and 32-D arrays (left to right). Compressed storage increases with the increase in sparse ratio for each specified

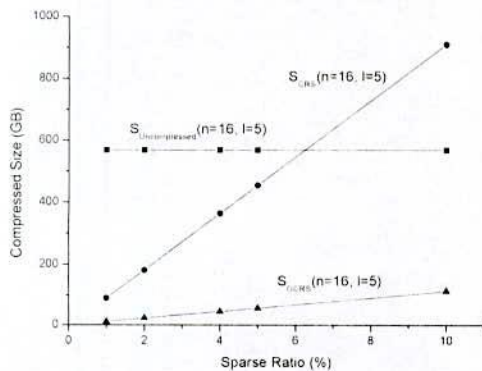
length and dimension shown in Figure 5.4- a), b), c) and d). From experimental result it is clear that the GCRS/GCCS outperform over CRS/CCS which comply with the equation 4.2 and equation 4.3 analyzed in section 4.4.1



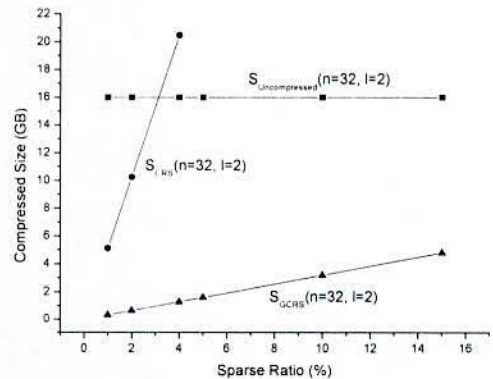
a) CRS and GCRS for $n=4$ and $l=720$



b) CRS and GCRS for $n=8$ and $l=26$



c) CRS and GCRS for $n=16$ and $l=5$



d) CRS and GCRS for $n=32$ and $l=2$

Figure 5.4 Space requirement for CRS and GCRS of different number of dimensions

It is clearly visible that the requirement of total storage in CRS increases drastically with the increase in number of dimensions. Sparse ratio of CRS shown in Figure 5.5- a), b), c) and d) are 25.0%, 12.5%, 6.25% and 3.13% respectively (decrease with increase in data-dimensions). Storage improvement in GCRS over CRS is 2, 4, 8 and 16 times in Figure 5.4- a), b), c) and d) respectively. It is also observed that the usable sparse ratio 50% is same for any dimensional array i.e independent of number of array dimension.

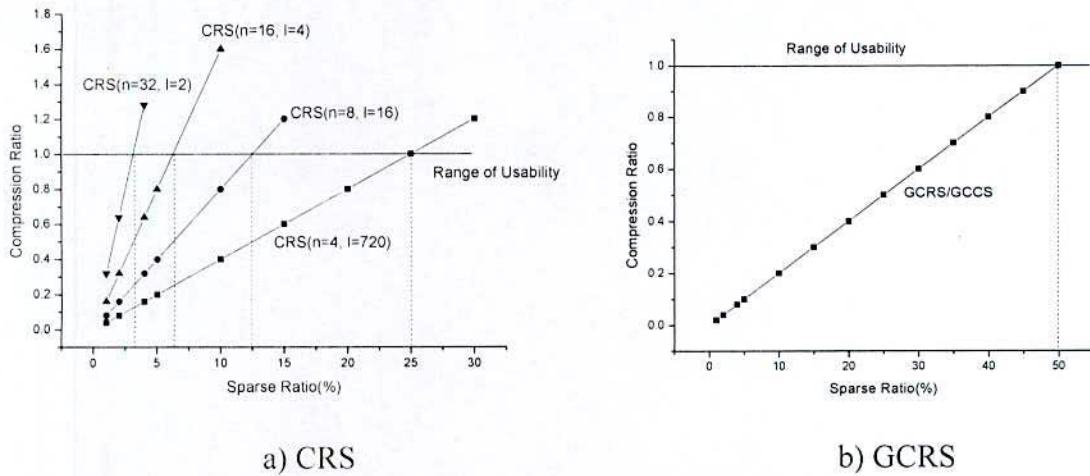


Figure 5.5 Compression Ratio for CRS & GCRS

Advantages of applying GCRS scheme over CRS scheme are the improvement of about 2, 4, 8 and 16 for 4-D, 8-D, 16-D and 32-D arrays respectively. Finally, Figure 5.6 shows the improvement over schemes and which is a straight line increases sharply with increase of the number of dimensions. It is clear that space complexity is very much prominent for GCRS than CRS when number of dimensions is very high.

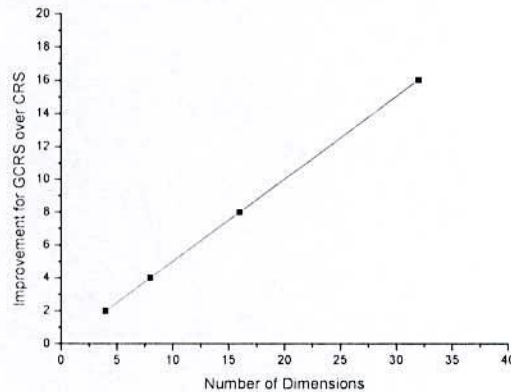
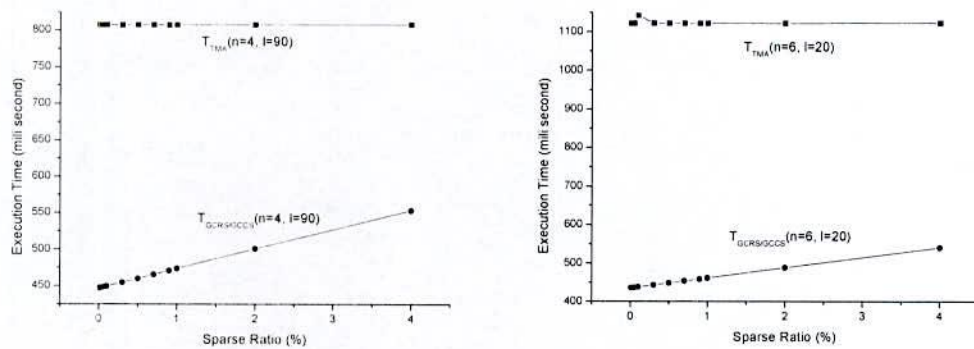


Figure 5.6 Improvement of GCRS over CRS

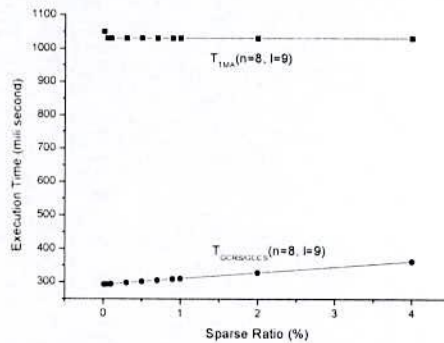
Experimental result of Figure 5.5 comprises that when number of dimensions increase for TMA then CRS/CSS schemes become unusable soon. But in case of our proposed GCRS/GCCS scheme, there is no relation between degree of data sparsity or range of usability and number of array dimensions. Hence it verified the theoretical analysis in Section 4.4.

5.5 Experimental Result for GCRS/GCCS Operations

The sparse ratio is set 0.01 to 4 percent for 4, 6 and 8 dimensional array operations in both matrix-matrix addition and matrix-matrix multiplication. Figure 5.7 shows the execution time in milliseconds of algorithms for the matrix-matrix addition on the sparse TMA and GCRS/GCCS. Experimental results show that execution time is less for GCRS/GCCS until data density is less than or equal to 2%. It seems that matrix-matrix addition algorithm for GCRS/GCCS is superior to TMA for wider range of data density. But matrix-matrix multiplication algorithm is superior for highly sparse data.

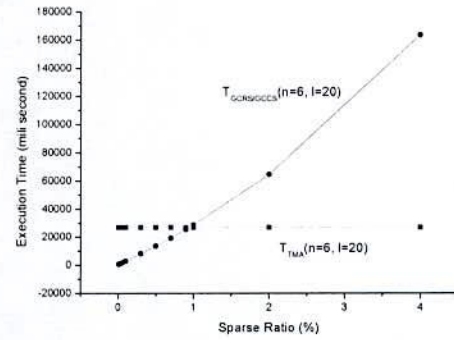
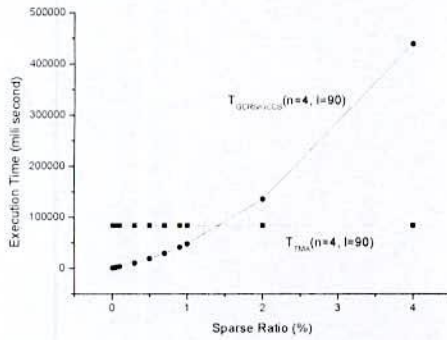


(a) Addition TMA(4) and CRS/GCCS (b) Addition TMA(6) and GCRS/GCCS



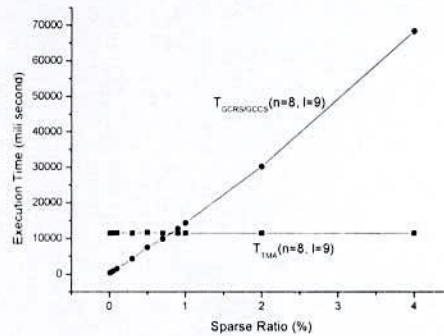
(c) Addition TMA(8) and GCRS/GCCS

Figure 5.7 Experimental results for GCRS/GCCS addition



(a) Multiplication TMA(4) and GCRS/GCCS

b) Multiplication TMA(6) and GCRS/GCCS



c) Multiplication TMA(8) and GCRS/GCCS

Figure 5.8 Experimental performance for GCRS/GCCS multiplication

5.6 Discussion

This chapter shows the experimental result is described in chapter 3 and chapter 4. The experimental results comply with the theoretical analysis described in individual cases. For matrix-matrix addition of G2A and TMA, the experimental results are shown only for equal length of each dimension. It is also experimented by varying length of each dimension from 2 to 120 and every case, G2A show better performance. The performance improvement in GCRS/GCCS scheme over CRS/CCS scheme to store an n -dimensional sparse array is a bit low than TMA and G2A operation. This is because the sparse storage scheme is implemented in secondary memory and accessing the secondary storage takes most of the time.

CHAPTER VI

Conclusions

6.1 Concluding Remarks

Most of the scientific and engineering computing requires operation on flooded amount of data having very high number of dimensions. This thesis represented a higher dimensional array implementation as row-column view or matricization. The main idea of the row-column view or G2A is fitting the odd dimensions allow row direction and even dimensions along column direction. The performance of matricized representation was shown and analyzed with matrix-matrix addition, subtraction and multiplication operation. But when most of the elements of multidimensional array are empty or null then above representation needs special treatment. Our proposed generalized row/column storage scheme for compressing higher dimensional sparse array was described. The algorithms for matrix operation on GCRS/GCCS data were elaborated with interactive figure. It is shown that the GCRS/GCCS scheme is independent with data dimensions and range of usability for it is higher than that of CRS/CCS scheme which comply with the theoretical analysis with simulated results. The performance improvement of GCRS/GCCS is directly proportional with degree of data sparsity while CRS/CCS performance inversely proportional with number of data dimensions. For worst case, GCRS/GCCS worked well for at least 50% dense data. Therefore, the scheme can be applied to the implementation of higher dimensional array computation, storage and analysis applications.

6.2 Future Scope

The future direction of this research may be summarized as bellow

- The parallel implementation of G2A scheme would be possible as G2A generates a set of 2-D blocks and each 2-D block is independent of each other to perform the operation on G2A. For the same independency of 2-D blocks it may be possible to found the parallel algorithms to store like GCRS/GCCS scheme.

- The G2A is a static structure. It is possible to make the G2A as a dynamic one i.e extension and reduction of the length of G2A can be made dynamic. This is an important property of big data technology.
- The parallel algorithms can be found for operation of matrix-matrix addition, subtraction and multiplication on compressed data according GCRS/GCCS scheme.
- The scheme can be applied to implement the compressed form of MOLAP server and schemes are based on G2A.

REFERENCES

- [1] Tamara G. Kolda, Brett W. Bader, "Tensor Decompositions and Applications", *SIAM Review* 51(3), pp. 455-500, 2009
- [2] Chun-Yuan Lin, Jen-Shiuh Liu, and Yeh-Ching Chung "Efficient Representation Scheme for Multidimensional Array Operations", *IEEE Transactions on Computers*, 51(3), pp.327-345, 2002
- [3] S. Sarawagi, M. Stonebraker. "Efficient organization of large multidimensional arrays" In *Proc. of 10th International Conference on Data Engineering (ICDE)*, pp. 328-386, Houston, Texas, 1994
- [4] Yihong Zhao, Prasad Deshpande, Jeffrey F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates", In *Proc. of SIGMOD Conference*, pp.159-170, 1997
- [5] Sheikh Mohammad Masudul Ahsan, "An efficient implementation scheme for multidimensional index array operations and its evaluation", A Masters' Thesis submitted to Department of Computer Science and Engineering, Khulna University of Engineering & Technology, Khulna, Bangladesh, Thesis No. CSER-M-1201, January 2012
- [6] Mackale Joyner, Zoran Budimlić, Vivek Sarkar, Rui Zhang "Array Optimizations for Parallel Implementations of High Productivity Languages" In *Proc. of PPOPP*, pp. 1-8, 2008
- [7] Emad Soroush, Magdalena Balazinska "ArrayStore: A Storage Manager for Complex Parallel Array Processing" In *Proc. of ACM SIGMOD International Conference on Management of data*, pp. 253-264, 2011
- [8] Francisco Heron de Carvalho Junior, Cenez Araújo Rezende, Jefferson de Carvalho Silva, Francisco José Lins Magalhães, and Renato Caminha Junior "On the Performance of Multidimensional Array Representations in Programming Languages Based on Virtual Execution Machines" In *Proc. of SBLP-2013, LNCS(8129)*, pp. 31-45, 2013
- [9] Jun Yan, Ning Liu, Shuicheng Yan, Qiang Yang, Weiguo Fan, Wei Wei, Zheng Chen "Trace-Oriented Feature Analysis for Large-Scale Text Data Dimension Reduction", *IEEE Trans. Knowl. Data Eng.* 23(7), pp.1103-1117, 2011
- [10] E Acar, B Yener, "Unsupervised multiway data analysis: A literature survey", *IEEE Transactions on Knowledge and Data Engineering* 21 (1), pp. 6-20, 2009

- [11] G. Beylkin, M. J. Mohlenkamp, "Algorithms for numerical analysis in high dimensions", *SIAM J. Sci. Comput.*, 26 (2005), pp. 2133–2159, 2005
- [12] P. M. Kroonenberg, "Applied Multiway Data Analysis", Wiley, New York, 2008
- [13] Jimeng Sun, Dacheng Tao, Spiros Papadimitriou, Philip S. Yu, Christos "Faloutsos: Incremental tensor analysis: Theory and applications" *ACM Transactions on Knowledge Discovery from Data*, 2(3), 2008
- [14] B. Christian, M. Urs, "Multidimensional Index Structures in Relational Databases", *Intelligent Information Systems*, 15, pp. 51–70, 2000
- [15] Steve Carr, Kathryn S. McKinley, Chau-Wen Tseng "Compiler optimizations for improving data locality", In *Proc. of the sixth international conference on Architectural support for programming languages and operating systems*, p.252-262, 1994
- [16] Kathryn S. McKinley, Steve Carr , Chau-Wen Tseng , "Improving data locality with loop transformations", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4), p.424-453, 1996
- [17] Holger Arndt, Markus Bundschuh; Andreas Naegele, "Towards a Next-Generation Matrix Library for Java", 33rd Annual IEEE International Computer Software and Applications Conference pp: 460–467, 2009
- [18] K. M. Azharul Hasan, Masayuki Kuroda, Naoki Azuma, Tatsuo Tsuji, and Ken Higuchi. "An extendible array based implementation of relational tables for multi dimensional databases." In *Data Warehousing and Knowledge Discovery*, LNCS, Springer Berlin Heidelberg, pp. 233-242, 2005
- [19] Ekow J. Otoo, T. H. Merrett. "A storage scheme for extendible arrays." *Computing* 31, no. 1 (1983): 1-9
- [20] Arie Shoshani "OLAP and statistical databases" *Proceeding of the 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of databases systems*, pp. 185-196,1997
- [21] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, Jeffrey F. Naughton, "Caching Multidimensional Queries Using Chunks" In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 259-270, 1998
- [22] Michael Steinbach, Levent Ertöz, Vipin Kumar, "The Challenges of Clustering High Dimensional Data" *New Directions in Statistical Physics*, pp. 273-309, Springer Berlin Heidelberg, 2004

- [23] Ekow J. Otoo, Doron Rotem, and Sridhar Seshadri "Optimal Chunking of Large Multidimensional Arrays for Data Warehousing" In Proc. of DOLAP, pp. 25-32, 2007
- [24] Michael Stonebraker, Paul Brown, Alex Poliakov, Suchi Raman "The Architecture of SciDB" In Proc. of the 23rd international conference on Scientific and statistical database management, pp. 1-16, 2011
- [25] Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, P. Sadayappan "Automatic Selection of Sparse Matrix Representation on GPUs", ICS'15, June 8–11, 2015
- [26] Tamara G. Kolda, Brett W. Bader, J. P. Kenny, "Higher-order web link analysis using multilinear algebra", in ICDM 2005: Proceedings of the 5th IEEE International Conference on Data Mining, IEEE Computer Society Press, pp. 242–249, 2005
- [27] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, "Templates for the Solution of Algebraic Eigenvalue Problems: *a Practical Guide*", SIAM, ISBN: 978-0-89871-471-5 (pp. 315-336), 2000
- [28] Nathan Bell, Michael Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA" NVIDIA Technical Report, 2008
- [29] The X10 Programming Language, "<http://x10-lang.org/>" retrieved at May 11, 2016
- [30] The Julia Language. "<http://julialang.org/>" retrieved at May 11, 2016
- [31] Parallel Programming and Computing Platform, "http://www.nvidia.com/object/cuda_home_new.html" retrived at May 11, 2016
- [32] Ekow J. Otoo, H Wang, G Nimako, "New Approaches to Storing and Manipulating Multi-Dimensional Sparse Arrays", Proc. of SSDBM'14, 2014
- [33] Brett W. Bader, Tamara G. Kolda "Efficient MATLAB computations with sparse and factored tensors" SIAM Journal on Scientific Computing, 2007
- [34] Chun-Yuan Lin, Yeh-Ching Chung, Jen-Shiuh Liu, "Efficient Data Storage Methods for Multidimensional Sparse Array Operations Based on the EKMR Scheme", IEEE Transactions on Computers, Vol. 52, No. 12, pp.1640-1646, 2003
- [35] M d. Rakibul Islam, K. M. Azharul Hasan, Tatsuo Tsuji, "EaCRS: An Extendible Array Based Storage Scheme for High Dimensional Data", Proc. of SoICT '11, pp. 92-99, 2011

- [36] Md. Rakibul Islam, "Compression schemes for high dimensional data based on extendible multidimensional arrays", A Masters' Thesis submitted to Department of Computer Science and Engineering, Khulna University of Engineering & Technology, Khulna, Bangladesh, Thesis No. CSER-M-1502, March 2015
- [37] Tatsuo Tsuji, Akihiro Hara and Ken Higuchi, "An Extendible Multidimensional Array System for MOLAP", Proc. of SAC'06, pp. 503-510, 2006
- [38] K. M. Azharul Hasan, Masayuki Kuroda, Naoki Azuma, Tatsuo Tsuji, Ken Higuchi (2005) "An Extendible Array Based Implementation of Relational Tables for Multi dimensional Data bases", In: Proceedings of 7th International Conference on Data Warehousing and Knowledge Discovery (DaWak'05), Copenhagen, Denmark, LNCS 3580, Springer-Verleg, pp. 233-242, 2005
- [39] K M Azharul Hasan, Tatsuo Tsuji, Ken Higuchi "An Efficient MOLAP Basic Data structure and Its Evaluation", Proc. of DASFAA, LNCS 4443, Springer-Verleg, pp. 288-299, 2011
- [40] Sk. Md. Masudul Ahsan and K.M. Azharul Hasan "An Implementation Scheme for Multidimensional Extendable Array Operations and Its Evaluation" In Proc. of ICIEIS, Part III, CCIS 253, pp. 136-150, Springer-Verleg, 2011
- [41] Geir Gundersen, Trond Steihaug, "Sparsity in higher order methods for unconstrained optimization", Optimization Methods and Software, Volume.27, Issue.2, pp.275, 2012
- [42] Zbigniew Koza, Maciej Matyka, Sebastian Szkoda, Łukasz Mirosław, "Compressed multi-row storage format for sparse matrices on graphics processing units", SIAM J. Sci. Comput. 36-2, pp. 219-239, 2014
- [43] Michael McCourt, Barry Smith, Hong Zhang, "Sparse Matrix-Matrix Products Executed Through Coloring", SIAM Journal on Matrix Analysis and App., 36:1, pp. 90-109, 2015
- [44] Chun-Yuan Lin, Huang Ting Y, Che-Lun Hung, "Efficient Strategies of Compressing Three-Dimensional Sparse Arrays based on Intel XEON and Intel XEON Phi Environments" IEEE International Conference on Computer and Information Technology, 2015
- [45] M. M. Mano, "Digital Logic and Computer Design", Prentice Hall, 2005