

https://dspace.org/

Sparse array representations and some selected array operations on GPUs

Wang, Hairong

2014-09-01
http://hdl.handle.net/10539/15329

Downloaded from a DSpace repository. https://dspace.org/



Sparse Array Representations And Some Selected Array Operations On GPUs

Hairong Wang

School of Computer Science

University of the Witwatersrand

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science.

Supervisor: Professor Ekow Otoo

July 2014, Johannesburg

Declaration

I, Hairong Wang, hereby declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other university.

Signature of candidate:_____

_____28th_____ day of _____July_____ 20 _____14_____ in _____Johannesburg_____.

Abstract

A multi-dimensional data model provides a good conceptual view of the data in data warehousing and On-Line Analytical Processing (OLAP). A typical representation of such a data model is as a multi-dimensional array which is well suited when the array is dense. If the array is sparse, i.e., has a few number of non-zero elements relative to the product of the cardinalities of the dimensions, using a multi-dimensional array to represent the data set requires extremely large memory space while the actual data elements occupy a relatively small fraction of the space. Existing storage schemes for *Multi-Dimensional Sparse Arrays (MDSAs)* of higher dimensions k ($k > 2$), focus on optimizing the storage utilization, and offer little flexibility in data access efficiency. Most efficient storage schemes for sparse arrays are limited to matrices that are arrays in 2 dimensions. In this dissertation, we introduce four storage schemes for MDSAs that handle the sparsity of the array with two primary goals; reducing the storage overhead and maintaining efficient data element access. These schemes, including a well known method referred to as the *Bit Encoded Sparse Storage (BESS)*, were evaluated and compared on four basic array operations, namely construction of a scheme, large scale random element access, sub-array retrieval and multi-dimensional aggregation. The four storage schemes being proposed, together with the evaluation results are: i.) The extended compressed row storage (xCRS) which extends CRS method for sparse matrix storage to sparse arrays of higher dimensions and achieves the best data element access efficiency among the methods compared; ii.) The bit encoded xCRS (BxCRS) which optimizes the storage utilization of xCRS by applying data compression methods with run length encoding, while maintaining its data access efficiency; iii.) A hybrid approach (Hybrid) that provides the best control of the balance between the storage utilization and data manipulation efficiency by combining xCRS and BESS. iv.) The PATRICIA trie compressed storage (PTCS) which uses *PATRICIA* trie to store the valid non-zero array elements. PTCS supports efficient data access, and has a unique property of supporting *update* operations conveniently. v.) BESS performs the best for the multi-dimensional aggregation, closely followed by the other schemes.

We also addressed the problem of accelerating some selected array operations using General Purpose Computing on Graphics Processing Unit (GPGPU). The experimental results showed different levels of speed up, ranging from 2 to over 20 times, on large scale random element access and sub-array retrieval. In particular, we utilized GPUs on the computation of the *cube* operator, a special case of multi-dimensional aggregation, using BESS. This resulted in a 5 to 8 times of speed up compared with our CPU only implementation. The main contributions of this dissertation include the developments, implementations and evaluations of four efficient schemes to store multi-dimensional sparse arrays, as well as utilizing massive parallelism of GPUs for some data warehousing operations.

Acknowledgements

First of all, I would like to thank my research supervisor, Professor Ekow Otoo, not only for his excellent guidance during the course of my research work, but also for the ideas he suggested, and his patient revision of my dissertation. My special thanks also go to Professor Michael Sears for his help and support.

I am grateful to the National Research Foundation (NRF) South Africa for their financial support for my study.

Finally, I would like to thank my husband, Sheng, for his love and support; and my three lovely daughters, Nandi, Christine and Marilyn, for being the joy of my life.

Contents

Declaration i

Abstract ii

Acknowledgements iii

List of Figures **vii**

List of Tables **viii**

List of Algorithms **viii**

1 Introduction **1**

1.1 Problem Motivation 1

1.2 Problem Statement 2

1.3 Overview of Some Known Approaches 4

1.4 Overview of Our Solution 5

1.5 Main Contributions 7

1.6 Organization of the Dissertation 8

2 Background and Related Work **9**

2.1 Storage Schemes for Multi-Dimensional Sparse Arrays 9

2.1.1 Index-Value Pair 10

2.1.2 Offset-Value Pair 11

2.1.3 Bit Encoded Sparse Storage 12

2.1.4 Compressed Row or Column Storage 13

2.2 Data Warehousing and OLAP 14

2.3 Multi-Dimensional Aggregation 16

2.3.1 The CUBE 17

2.3.1.1 Search Lattice 19

2.3.1.2 Algorithms for Computing the CUBE 20

2.4 General Purpose Computing Using GPUs 22

2.4.1 GPU Architecture 23

2.4.2 CUDA Programming Model . . . 24
2.5 Application of GPUs to Data Warehousing . . . 26

3 Multi-Dimensional Sparse Array Representations 28
3.1 Methodology . . . 28
3.2 Extended Compressed Row or Column Storage . . . 29
3.2.1 XCRS and Its Construction . . . 30
3.2.2 Random Element Access and Sub-Array Retrieval in xCRS . . . 31
3.2.3 Space Utilization of xCRS . . . 33
3.3 Bit Encoded Extended Compressed Row Storage . . . 33
3.3.1 Word-Aligned Hybrid Code . . . 33
3.3.2 BxCRS and Its Construction . . . 34
3.3.3 Random Element Access and Sub-Array Retrieval in BxCRS . . . 36
3.4 Hybrid Approach . . . 38
3.4.1 Hybrid and Its Construction . . . 38
3.4.2 Random Element Access and Sub-Array Retrieval in Hybrid . . . 39
3.4.3 Some Properties of Hybrid . . . 40
3.4.3.1 The Storage Overhead . . . 40
3.4.3.2 The Time Complexities of Random Element Access and Sub-Array Retrieval in Hybrid . . . 40
3.5 PATRICIA Trie Compressed Storage . . . 41
3.5.1 PATRICIA . . . 41
3.5.2 PTCS and Its Key . . . 42
3.5.3 PTCS Construction . . . 43
3.5.4 Random Element Access and Sub-Array Retrieval in PTCS . . . 46
3.5.5 Some Properties of PTCS . . . 47

4 Multi-Dimensional Aggregations of Sparse Array Elements 48
4.1 Aggregation Using PTCS and BESS . . . 49
4.2 Aggregation Using xCRS and BxCRS . . . 50
4.3 Aggregation Using Hybrid . . . 51
4.4 Comparative Analysis of Computing Aggregations Using Various Schemes . . . 52
4.5 Computing the Cube . . . 53
4.5.1 Paths in the Search Lattice . . . 54
4.5.2 Computing the CUBE Using BESS . . . 56
4.5.3 Computing the CUBE Using PTCS . . . 58

5 Selected Array Operations on GPUs **59**

5.1 Overview . . . 59

5.2 Large Scale Random Element Access . . . 61

5.3 Sub-Array Retrieval . . . 62

5.4 Computing The Cube . . . 65

5.4.1 Resetting the Attribute Order . . . 65

5.4.2 Sorting . . . 66

6 Experimental Setup **68**

6.1 Experimental Environment . . . 68

6.2 Experimental Data . . . 68

7 Performance Evaluation **71**

7.1 Storage Utilization of Various Storage Schemes . . . 71

7.2 Experimental Results and Comparative Analyses . . . 72

7.2.1 Results on CPU Only Processing . . . 72

7.2.1.1 Performance of Storage Scheme Construction . . . 72

7.2.1.2 Performance of Large Scale Random Element Access . . . 73

7.2.1.3 Performance of Sub-Array Retrieval . . . 75

7.2.1.4 Performance of Aggregation . . . 76

7.2.1.5 Performance of Computing the CUBE Using PTCS and BESS . . . 77

7.2.2 Results on CPU+GPU Co-Processing . . . 78

7.2.2.1 Performance of Large Scale Random Element Access . . . 78

7.2.2.2 Performance of Sub-Array Retrieval . . . 79

7.2.2.3 Performance of Computing the CUBE Using BESS . . . 80

8 Conclusion **87**

8.1 Main Objectives . . . 87

8.2 Main Contributions . . . 88

8.3 Future Work . . . 89

Appendix **90**

A Additional Algorithms **90**

A.1 The Sub-Array Retrieval Algorithm in XCRS . . . 90

A.2 The Algorithm to Search the Array *compwrd* . . . 91

A.3 The Sub-Array Retrieval Algorithm in PTCS . . . 92

Bibliography **93**

List of Figures

2.1 An Example of an MDSA 11
2.2 The Aggregation on a 3-Dimensional Array 17
2.3 The Aggregation on the Group-By of a Single Dimension 18
2.4 A Search Lattice with 4 Attributes 19
2.5 A Modern NVIDIA GPU Architecture 24
2.6 CUDA Memory Hierarchy 26

3.1 The Process of Bitmap Compressing 35
3.2 An Example of a PATRICIA Trie 42
3.3 A PTCS Key Structure 43
3.4 The Insertion of a Key Into a PATRICIA Trie 44

4.1 Aggregating the MDSA Represented in PTCS or BESS 50

5.1 An Example of P-Ary Search 62

7.1 Storage Ratios for the 8-Dimensional Sparse Arrays 72
7.2 The Construction Time of Various Storage Schemes 73
7.3 The Average Random Element Access Time of Various Storage Schemes 74
7.4 The Structures of the Sub-Arrays to be Retrieved 75
7.5 The Average Sub-Array Retrieval Time of Various Storage Schemes ($k = 2$ and $k = 3$) 77
7.6 The Average Sub-Array Retrieval Time of Various Storage Schemes ($k = 4$ and $k = 8$) 77
7.7 The Multi-Dimensional Aggregation Time of Various Storage Schemes ($k = 2$ and $k = 3$) . . 81
7.8 The Multi-Dimensional Aggregation Time of Various Storage Schemes ($k = 4$ and $k = 8$) . . 81
7.9 The Multi-Dimensional Aggregation Time of Various Storage Schemes ($k = 4$ and $k = 8$) . . 81
7.10 The Time for Computing the Cube Using PTCS and BESS ($k = 2$ and $k = 3$) 82
7.11 The Time for Computing the Cube Using PTCS and BESS ($k = 4$ and $k = 8$) 82
7.12 The Average CPU+GPU Random Element Access Time Using BESS 83
7.13 The Average CPU+GPU Random Element Access Time Using xCRS 83
7.14 The Average CPU+GPU Random Element Access Time Using Hybrid 83
7.15 The Average CPU+GPU Sub-Array Retrieval Time Using BESS 84

7.16 The Average CPU+GPU Sub-Array Retrieval Time Using xCRS 84
7.17 The Average CPU+GPU Sub-Array Retrieval Time Using Hybrid 84
7.18 The CPU+GPU Time for Computing the Cube Using BESS ($k = 4$) 85
7.19 The CPU+GPU Time for Computing the Cube Using BESS ($k = 8$) 85
7.20 The Sort and Aggregation Time for Computing the Cube Using BESS ($k = 4$) 86
7.21 The Sort and Aggregation Time for Computing the Cube Using BESS ($k = 8$) 86

List of Tables

2.1 The Parameters and Notations 10
2.2 An Example of Index-Value Pair Representation 11
2.3 An Example of Offset-Value Pair Representation 12
2.4 An Example of BESS Representation 13
2.5 An Example of CRS 14

3.1 An Example of XCRS 31
3.2 An Example of BxCRS 36
3.3 An Example of Hybrid 39
3.4 The PTCS Key-Value Pair 43

4.1 The Costs of Computing Multiple Aggregations 53

6.1 The Specifications of a Set of Experimental Data 70

List of Algorithms

1 XCRS Construction 31

2 XCRS Random Element Access 32

3 Compressing the Bitmap 35

4 BxCRS Random Element Access 37

5 PTCS Construction 45

6 PTCS Random Element Access 46

7 Computing the Paths in a Search Lattice 56

8 Computing the *Cube* Using BESS 57

9 GPU P-Ary Search Algorithm 63

10 GPU Sub-Array Retrieval Using BESS 64

11 Generating the Dimensional Data of an MDSA 70

12 XCRS Sub-Array Retrieval 90

13 Searching the Array $compwrd$ 91

14 PTCS Sub-Array Retrieval 92

Chapter 1

Introduction

A multi-dimensional data model provides a good conceptual view of the data in data warehousing. This system supports On-Line Analytical Processing (OLAP) and facilitates complex analyses and visualizations of data. It enables finding certain patterns or trends in the data. Such valuable sources of information are essential in business management and decision making. Data warehouses usually contain historical, summarized and consolidated data, perhaps from several operational databases, over very long periods of time. They tend to be much larger than an individual operational database. The workloads are mostly complex queries that often need to access large amount of data and perform operations, such as scans and aggregations. As a result, the data in a data warehousing system is usually modeled as a multi-dimensional arrays [5]. For example, a data warehouse may represent business data (such as the sales data of a car manufacturer), medical data, scientific data and other real-world data. These data sets are characterized by several dimensions of interest and one or more measured values. Allied to the multi-dimensional data model is the multi-dimensional data cube, where the dimensions form the axes of the cube in multi-dimensional OLAP (MOLAP). It is typical in data warehousing and MOLAP to have the data sets be large and sparse, and it is not unusual in MOLAP data to find that 20% or fewer of the data elements are non-zero [42]. This degree of *sparsity* of data tends to be much higher with higher dimensions. We define the sparsity of a multi-dimensional array later in this text. Storing and manipulating large multi-dimensional data sets are also very common in many scientific and statistical databases, as well as scientific and engineering applications. The challenge in these applications is to find an efficient storage scheme to store very large data sets that have large number of dimensions.

1.1 Problem Motivation

A typical representation of a multi-dimensional data model is as a multi-dimensional array. It organizes the data in multi-dimensional space, where each dimension represents one of the attributes of the data. The size of the multi-dimensional space is determined by the cross product of the cardinalities of each dimension. Such an approach is reasonably efficient when the array is dense, and has the advantage of excellent data access

efficiency. However, if the multi-dimensional data is sparse, using multi-dimensional array to represent such data sets requires extremely large data space while the actual data elements occupy only a small fraction of this space. Even though the entire storage space cannot be defined in memory, the actual data of non-zero values can be accommodated entirely in memory. Besides extremely large storage overhead, the efficiency in accessing data is easily traded off by visiting large amounts of invalid array elements. A number of storage schemes exist for sparse matrices, i.e., 2-dimensional sparse arrays, but not necessarily for higher dimensional arrays. Sparse matrices and sparse arrays result from data structures used for representing the information in various applications. Examples are: i.) use of adjacency matrix representation for very large graphs in numerous graph applications; ii.) multi-dimensional representation of relations for relational database and consequently in data warehousing.

The data in data warehousing and MOLAP is characterized by large volume, high dimensionality, and sparsity. Let ρ denote the *occupancy ratio* of a k-dimensional array $A[D_{k-1}]\ldots[D_0]$, where D_j, $0 \leq j \leq k-1$, is the cardinality or bound of the j^{th} dimension. Suppose the number of non-zero elements is $\mathcal{N}_{nz}$, then $\rho = \mathcal{N}_{nz}/\prod_{j=0}^{k-1} D_j$ and the sparsity , denoted by σ, is defined by $\sigma = 1-\rho$. Multi-dimensional array is often a desirable data structure to be used to represent the data in data warehousing and MOLAP. It is essential to handle the sparsity in the array structure so that we can achieve overall performance enhancement. Efficient storage schemes of sparse matrices have been developed to gain significant performance improvements in scientific computing and engineering applications.

1.2 Problem Statement

The question is whether a storage scheme can be implemented that stores only those occupied cells of array elements (or the non-zero values) in the corresponding multi-dimensional array structure with efficient element access performance. The problem we address concerns developing and implementing a storage scheme for multi-dimensional sparse arrays (MDSAs) that uses space proportional to the number of non-zero elements without compromising on the element access efficiency. Formally, we have the following: given a k-dimensional array $A[D_{k-1}]\ldots\ [D_1]\ [D_0]$ of $\mathcal{N}_{nz}$ non-zero elements where $\mathcal{N}_{nz} \ll \prod_{j=0}^{k-1} D_j$, define an efficient representation of the array A using space $O(\mathcal{N}_{nz})$ that has element access time of $O(k)$ at best and $O(\log \mathcal{N}_{nz})$ in the worst case. To elaborate on the problem statement, consider the non-zero elements of the k-dimensional sparse array, $A[D_{k-1}]\ldots\ [D_0]$ mapped onto a p-dimensional array $F[M_{p-1}]\ldots\ [M_0]$ where $p < k$. The size of F is $\prod_{j=0}^{p-1} M_j = s \geq c\mathcal{N}_{nz}$, for some small constant c. The array F is assumed to be linearized into a sequence of consecutive memory locations $\mathcal{L}[s] = \mathcal{L}\langle 0\rangle, \mathcal{L}\langle 1\rangle, \ldots, \mathcal{L}\langle s-1\rangle$. The problem then becomes finding a data structure to represent $\mathcal{L}$ and a function $f()$ that maps an element $A\langle n_{k-1}, n_{k-2}, \ldots, n_0\rangle$

onto a location $\mathcal{L}\langle q\rangle$, with $A\langle 0,0,\ldots,0\rangle$ assigned to $\mathcal{L}\langle 0\rangle$ such that $\mathcal{L}$ stores only or mostly the non-zero elements. We desire an efficient compute function $f()$ such that $f(\langle n_{k-1}, n_{k-2}, \ldots, n_0\rangle) \rightarrow q$ and its inverse function $f^{-1}(q) \rightarrow \langle n_{k-1}, n_{k-2}, \ldots, n_0\rangle$.

Our primary goal in this research is to develop efficient storage schemes for multi-dimensional sparse arrays that address the problem stated above, i.e., reducing the storage overhead and maintaining the data access efficiency for query operations. Less storage overhead results in reduced memory and disk space, and better utilization of memory bandwidth. Efficient data access ensures less runtime penalties and faster matrix and array computations. It is also crucial to perform the typical array operations and data analytics on the resulting data structure of any storage scheme without restoring the original sparse array. A number of problems precipitate from the original principle research problem. These include:

Time to create the representative structure of $\mathcal{L}$: The general presentation of the non-zero elements of a sparse matrix or array is in the Matrix Market format (MM-format) [3, 4], which is also referred to as the *Coordinate* format. It is an ASCII file format with indexes that are either 0-based or 1-based. Each record gives the coordinates and the non-zero value. The first record represents the values of the bounds of the dimensions and the number of records of non-zero values in the file. Given an MDSA in the MM-format, the question is how fast can the data structure for $\mathcal{L}$ be created? What is the occupancy ratio achievable by $\mathcal{L}$?

Complexity of accessing a random element $A\langle n_{k-1}, \ldots, n_0\rangle$: Given the array index $\langle n_{k-1}, \ldots, n_0\rangle$, what is the complexity of computing $q = f(\langle n_{k-1}, \ldots, n_0\rangle)$, or given the value of q, what is the complexity of computing $\langle n_{k-1}, \ldots, n_0\rangle = f^{-1}(q)$?

Sub-array retrieval: Given a k-dimensional array $A[D_{k-1}]\ldots[D_0]$ and rectilinear boundary indexes $L = \langle l_{k-1}, \ldots, l_0\rangle$ and $H = \langle h_{k-1}, \ldots, h_0\rangle$ where $l_j \leq h_j, 0 \leq j \leq k-1$, what is the time to return all the non-zero elements in the sub-array defined by L and H?

Multi-dimensional aggregation: Given an aggregate function $f()$ and a subset of k dimensions, how fast can the aggregation be computed for the representative data structure of $\mathcal{L}$? This is also related to the problem of sub-array retrieval where in that case, an aggregate function is applied to the elements retrieved.

Nearest neighbor retrieval and top-k: Given a non-zero value $\mathcal{L}\langle q\rangle$ and a distance metric, determine the nearest non-zero element $\mathcal{L}\langle q'\rangle$. Related to this is the top-k that subsumes the nearest neighbor query. In this case we are given a non-zero value $\mathcal{L}\langle q\rangle$ and a measure function $g()$, we desire the first k values $\mathcal{L}\langle q'_1\rangle, \mathcal{L}\langle q'_2\rangle, \ldots, \mathcal{L}\langle q'_k\rangle$, closest to $\mathcal{L}\langle q\rangle$ that satisfy $g(\mathcal{L}\langle q\rangle) \otimes g(\mathcal{L}\langle q'_j\rangle), 1 \leq j \leq k$, where $\otimes \in \{<, \leq, =, >, \geq\}$.

Multi-dimensional data model not only captures the structure of the underlying data well, it is also amenable to parallelism. Parallelism plays a significant role in processing the massive amount of data in data warehousing and MOLAP. As the secondary goal of this research, we explore parallelizing selected array operations on GPUs using some of the storage schemes concerned in this research. The selected array operations include large scale random array element access, sub-array retrieval, as well as multi-dimensional aggregation on a special case of the *cube* computation.

1.3 Overview of Some Known Approaches

A number of methods have been used to handle sparse multi-dimensional arrays in the literature. In the case of 2-dimensional sparse arrays or sparse matrices the Compressed Row (or Column) Storage (CRS/CCS) is a well known scheme. This is discussed in detail in Section 2.1.4. The *Offset-Value* pair [51] is the most often used method to optimize the storage utilization of multi-dimensional sparse arrays with higher dimensionality (> 2). It stores only a pair of values; an offset value l and the data element v, for each non-zero element in the sparse array, given a scan order of the array and thereby incurring a low storage overhead. Some typical scan orders are the *row-major* or *column-major* order. Given a k-dimensional array $A[D_{k-1}] \dots [D_0]$, where an element $A\langle n_{k-1}, \dots n_0 \rangle$, is referenced by the indexes $n_{k-1}, n_{k-2}, \dots, n_1, n_0$, a scan order is termed *row-major* if the scan of the elements has the lowest index n_0, varying the fastest. It is termed *column-major* order if the high order index n_{k-1} varies the fastest. The computation of offsets from the array indexes, or vice versa, requires certain number of algebraic operations such as multiplication, addition, division, or subtraction. These computations become expensive when they are performed on a large number of dimensions.

Bit Encoded Sparse Storage (BESS) [12] was designed to overcome the computational cost of offset-value pair by encoding the array indexes into binary bits, concatenating these bits, and interpreting the concatenated bit string as an integer (see Section 2.1.3 for more details). As a result, the algebraic operations are replaced by more efficient bit concatenation operation. This method has the same storage efficiency as the offset-value pair. The random array element access time, in both cases, is achieved in time $O(\log \mathcal{N}_{nz})$, since a binary search algorithm has to be used for a random element access.

Sparse matrices, a class of multi-dimensional array with $k = 2$, arise in a wide range of compute-extensive scientific and engineering applications. Many different storage schemes for sparse matrix have been designed to take advantage of the structure of the matrices or the specificity of the problem from which they arise. Consequently, these storage schemes are often application or structure specific, and limited to 2 dimensions. On the other hand, there are relatively much less methods to efficiently represent multi-dimensional sparse

arrays of more than 2 dimensions. The k-dimensional data models applied in data warehousing, OLAP systems and multi-dimensional databases, typically have $k > 2$.

1.4 Overview of Our Solution

We developed four storage schemes for MDSAs, and implemented algorithms for constructing these schemes. Our approaches, in developing new storage schemes, include extending sparse matrix storage schemes to higher dimensions, combining different storage schemes, and applying suitable data compression, when necessary, to the resulting representation of MDSAs using some storage formats. We also further explored a trie based approach, PATRICIA Trie Compressed Storage, an improved version from our previous work [48]. These storage schemes are:

Extended Compressed Row Storage (xCRS): Compressed Row Storage, the widely used sparse matrix storage scheme, is extended to represent sparse arrays of any dimensions by simply mapping the multi-dimensional sparse array to a set of one dimensional arrays.

Bit Encoded Extended Compressed Row Storage (BxCRS): To address the shortcoming of xCRS on very sparse multi-dimensional arrays, we applied data compression using Run Length Encoding to optimize the storage utilization of xCRS, while maintaining its data access efficiency.

Hybrid Approach (Hybrid): The xCRS and BESS are combined to give a better control of the balance between storage utilization and data manipulation efficiency.

PATRICIA Trie Compressed Storage (PTCS): A PATRICIA trie that stores a key-value pair for each non-zero element is constructed for a given MDSA.

To evaluate the storage schemes, we designed and implemented algorithms for *large scale random array element access*, *sub-array retrieval*, and *multi-dimensional aggregation*, for each of the storage schemes outlined in the previous paragraph, as well as a known scheme, BESS, for comparative purpose. As a special case of multi-dimensional aggregation, we also implemented algorithms to compute the *cube* operator using two storage schemes, namely PTCS and BESS, respectively.

We explored accelerating some of the operations for the sparse arrays using GPU as a co-processor. These operations include large scale random array element access or searching, and sub-array retrieval. Finally, computing the *cube* operator using BESS was accelerated using GPU as a co-processor. Data warehousing and OLAP analyze large volumes of data and are highly compute- and data-intensive. In addition, the multi-core

and many-core architectures are the current predominant technologies and will remain as trends in the future. Therefore, parallelism should be exploited whenever possible in these applications. We chose GPU as the parallel platform for this purpose. Traditionally designed for gaming applications, GPUs have relatively more computing power and high memory bandwidth compared with their contemporary CPUs. While the performance of graphics hardware is rapidly increasing, they become more programmable due to the flexible architectural designs with every major generations of GPUs. High level GPU programming models and languages, or programming models for heterogeneous systems, such as Compute Unified Device Architecture (CUDA) from NVIDIA [32] and OpenCL [23], have been emerging to meet the demands of utilizing GPUs or other types of processors for general purpose computing. Furthermore, compared with other parallel platforms, heterogeneous systems often have potential to achieve excellent $performance/cost$ ratio. All these features of GPUs make them an attractive platform to be utilized in many applications. Recent research work has shown that GPUs can be used to accelerate data warehousing applications [49, 46]. GPUs have been successfully applied to accelerate individual database operations, such as sort [13] and join [19], relational query processing [18], and some of the data mining operations [11].

As part of the work of our second goal, the issue of computing the multi-dimensional aggregation on a special case of the *cube* operator was examined. The *cube* operator is the multi-dimensional generalization of *group-by* [44] operator. It computes group-bys corresponding to all possible combinations of a list of attributes. Many queries over data warehouses and MOLAP require summary data and as such use aggregate operations. Further, data analyses in these applications are often interactive. Hence, response time is a crucial factor in the performance. Materializing some of the summary data, or pre-computing partial or total *cube*, is a key technique to answer common queries efficiently in data warehousing and MOLAP. In computing the *cube*, we mainly combined the following two approaches; one was to represent the MDSA using a space efficient storage scheme, such as BESS; the other was to utilize GPUs in computing some part of the cube operation.

Taking the time constraint into consideration, we restricted ourselves to consider only the case where the data can be fit into main memory, leaving the case where the data can not fit into main memory for future work. The methods to represent MDSAs we propose ensure either more data fitting into main memory or efficient access to the data. Accommodating more data in main memory results in less partitioning cost on disk data, hence less I/O cost. Graefe pointed out a number of benefits of data compression in database system [14]. These benefits include reduced disk space, improved I/O performance, more data being fit into main memory, etc. The author also pointed out that most query processing can be carried out on the compressed data. Some of these benefits can be realized to a certain degree by applying a suitable storage scheme to an MDSA. Organizing MDSAs using an efficient storage scheme could lead to two fold benefits. Firstly, we may achieve an optimal storage utilization, efficiency in data manipulation, or a desired balance between them, by carefully choosing a storage scheme to represent the MDSA. Secondly, we may further apply one of the data compression techniques, such

as in the case of BxCRS, to the data represented in some storage schemes to improve the storage utilizations while maintaining their data access efficiencies.

1.5 Main Contributions

The main contributions of this dissertation include the following. Firstly, we designed, implemented and evaluated four of the storage schemes for MDSAs, namely xCRS, BxCRS, Hybrid and PTCS. These methods contribute towards a wider range of methods of organizing MDSAs in data warehousing, MOLAP, and other relevant applications in different fields. They provide different features such as optimal storage utilization, efficient data access, or some balance between the two. Among these methods, we investigated various basic yet novel ideas, such as using trie based data structure to store MDSAs, combining different storage schemes to gain better control of the balance between space and computational efficiencies, and applying data compression within a storage scheme. Secondly, by utilizing GPUs as a co-processor to accelerate some selected array operations, we demonstrated how GPUs can be applied to accelerate some basic operations in data warehousing and MOLAP, and consequenly improve the performance in these applications. Thirdly, we implemented computing the *cube* using BESS as the storage scheme, and utilizing GPUs as the co-processor. The benefits of this approach are as follows.

- Using BESS as the storage scheme leads to more data being accommodated in memory. Further, since BESS represents the dimensional data (or the indexes of the array elements) in a very compact form, it not only simplifies some of the necessary operations, such as sorting when computing the *cube*, but also optimizes the PCI-Express bus bandwidth between the CPU and GPU implicitly, by transferring more information in the same amount of data.
- We were able to achieve a speed-up of 5 to 8 times compared with our single-core CPU implementation of computing the *cube* using BESS or PTCS. Hence, the effectiveness of utilizing both CPU and GPUs on the problem of computing the *cube* was demonstrated.

Due to the limited time for this research, the problems of nearest neighbor and top-k retrievals are not addressed in this dissertation. They are left for future work, where we hope to design and implement the algorithms in these regards for the data represented by the selected storage schemes.

1.6 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 gives the backgrounds and related works in MDSA storage schemes, data warehousing and OLAP, multi-dimensional aggregation and general purpose computing on GPUs. We introduce the designs of new storage schemes in Chapter 3, as well as the relevant algorithms for creating the storage structures, implementing the basic array operations, and give some analytical properties of each scheme. We discuss computing the multi-dimensional aggregations and the *cube* operator using various storage schemes in Chapter 4. Our approaches to utilize GPUs on the selected array operations are presented in Chapter 5. The experimental setup and discussion of data sources for reproducibility are given in Chapter 6. The experimental results and comparative analyses of the results obtained are presented in Chapter 7. We finally conclude the work of this dissertation in Chapter 8, where we also discuss the direction for future work.

Chapter 2

Background and Related Work

In this chapter, we discuss related work on multi-dimensional sparse array representation and their application to data warehousing and OLAP. We discuss earlier works on computing multi-dimensional aggregates and the *cube* operator. We also briefly present the GPU architecture, the CUDA programming model and their use as general purpose computing engines in related fields.

2.1 Storage Schemes for Multi-Dimensional Sparse Arrays

There exist a number of storage schemes for multi-dimensional sparse arrays, mostly specialized for sparse matrices. Many different schemes for sparse matrices have been designed to take advantage of the structure of the matrices or the specificity of the problem from which they arise. For example, the special sparse structures that are often exploited in designing storage schemes for sparse matrix include: a diagonal matrix where the matrix consists of a few diagonals of non-zero elements; a block matrix where the non-zero elements are square dense blocks; symmetric matrix; asymmetric matrix etc. The purpose of each of these schemes is to gain efficiency both in memory utilization and matrix computations. A survey of such a collection of sparse matrix storage formats can be found in the work of Barret et al. [2] and Saad [39].

In the following text, we give an overview of some of the known storage formats that are relevant to our work. These formats are general in the sense that they do not make any assumption about the sparsity or the structural shape of the underlying arrays. Table 2.1 gives the notations and descriptions of the parameters used in the analyses of the various storage schemes in this dissertation.

Parameter and Notation	Description
C_i	The number of bytes in an integer
C_{il}	The number of bytes in the longest integer
C_f	The number of bytes in a float or real number
d_j	The name of a dimension j
D_j	The cardinality or bound of a dimension j or d_j
$\mathcal{S}$	The size of a storage space in bytes
W	The bit length of a computer word
$\mathcal{N}_{nz}$	The number of non-zero elements in an array
$\mathcal{N}_j$	$\mathcal{N}_j = \prod_{r=k-j}^{k-1} D_r$, $1 \leq j \leq k$
ρ	The occupancy ratio, $\rho = \mathcal{N}_{nz}/\mathcal{N}_k$
σ	The sparsity of an MDSA, $\sigma = 1 - \rho$
l, v	The offset and value pair for an array element $A\langle n_{k-1} \ldots n_0 \rangle$
BEI	Bit Encoded Index

Table 2.1: The parameters and notations

2.1.1 Index-Value Pair

The index-value pair format is the simplest format for representing an MDSA. It is also termed Coordinate (COO) format and is commonly used for sparse matrices. This is the general format for storing the non-zero values of an MDSA in an ASCII file and is also referred to as the Matrix-Market format [3]. To represent a k-dimensional sparse array, the index-value pair uses $k + 1$ one-dimensional arrays or vectors; one vector for storing all the non-zero values, and the other k vectors for storing the indexes of the corresponding non-zero values. The length of each of these vectors is $\mathcal{N}_{nz}$, i.e., the number of non-zero elements. Hence, the total storage space required for the index-value representation is $(kC_i + C_f)\mathcal{N}_{nz}$. Table 2.2 shows the index-value pair representation of an example MDSA in Figure 2.1. In Table 2.2, the vector val stores the non-zero values, and the vectors $ind1$, $ind2$, $ind3$ store the corresponding indexes, for each non-zero value, on dimension I, J, K respectively. The example of the 3-dimensional sparse array of Figure 2.1 is used throughout the dissertation.

Both the storage and the data access efficiencies of the index-value pair format become worse when the number of dimensions, k, increases. However, two variations to this format, namely, Offset-Value Pair and Bit Encoded Sparse Storage, are commonly applied for MDSAs of higher dimensions, where the k one-dimensional arrays of indexes are reduced to only one.

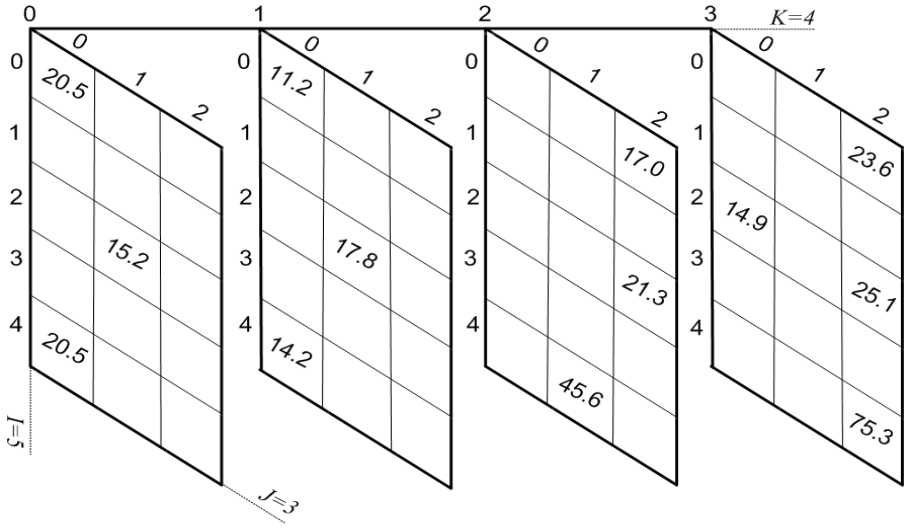


Figure 2.1: An example 3-dimensional ($5 \times 3 \times 4$) sparse array.

$ind1$	$ind2$	$ind3$	val
0	0	0	20.5
0	0	1	11.2
0	2	2	17.0
0	2	3	23.6
2	0	3	14.9
2	1	0	15.2
2	1	1	17.8
2	2	2	21.3
2	2	3	25.1
.....			

Table 2.2: The index-value pair representation of the MDSA in Figure 2.1.

2.1.2 Offset-Value Pair

Given a k-dimensional sparse array $A[D_{k-1}] \dots [D_0]$, the offset-value pair format represents each non-zero element in A using two values, an offset and the non-zero value. The offset l is a displacement in a given linearly addressed space starting from 0. We may obtain a linear address space for any sparse array by traversing the array elements in a certain scan order, such as *row-* or *column-major* order. Other scan orders are the *Z-order*, the *Peano-Hilbert order* and the *Grey-Code* order [40]. For example in a *row-major* offset-value representation, the k-dimensional index $\langle n_{k-1}, \dots, n_0 \rangle$ of any array element is used to compute its offset l as

$$l = \sum_{i=0}^{k-1} n_i \prod_{j=0}^{i-1} D_j$$

Offset-value pair storage uses two one-dimensional arrays; one to store the actual non-zero values and the other to store the corresponding offsets. The lengths of both of these two arrays are $\mathcal{N}_{nz}$. Thus, the storage space requirement using offset-value pair is only $(C_{il}+C_f)\mathcal{N}_{nz}$. Table 2.3 shows two offset-value pair representations of the 3-dimensional array of Figure 2.1 for two different linear addressing. This method is often used to

represent multi-dimensional sparse arrays because of its storage efficiency [51], which is linear in the number of non-zero elements. Random array element access in offset-value pair has a complexity of $O(\log \mathcal{N}_{nz})$ using *binary search*. One may argue if the use of interpolation search, with average search time complexity of $O(\log \log \mathcal{N}_{nz})$ [37], would not be more appropriate. In practice, this is worse than the use of binary search, since the interpolation search involves multiplication and division of floating point numbers while the binary search involves only one integer division by 2 and is more efficient.

offset_1	*val*
0	20.5
1	11.2
10	17.0
11	23.6
27	14.9
28	15.2
.....	

A

offset_2	*val*
0	20.5
4	20.5
7	15.2
15	11.2
19	14.2
22	17.8
.....	

B

Table 2.3: The offset-value pair representations of the MDSA in Figure 2.1. Table A is in the row-major ($I - J - K$) order, and Table B is in the column-major ($K - J - I$) order.

The major shortcoming of offset-value pair is that it is not computationally efficient. To compute an offset, we need to do $k - 1$ multiplications and summations, and to recover the array index, we need to do the same amount of divisions and subtractions. Another issue of offset-value pair is that it has the potential to overflow. The value $\prod_{i=0}^{k-1} D_i$ tends to be very large when the dimensionality increases, or the individual dimensions have large cardinalities. This could result in a large integer value that overflows the 32-bit or even a 64-bit integer representation. Our approach, in this dissertation, is to partition the very large linear address space, if it occurs, into relatively smaller ones.

2.1.3 Bit Encoded Sparse Storage

Bit Encoded Sparse Storage (BESS) consists of two one-dimensional arrays; one stores the non-zero values, the other stores the *bit encoded indexes* of the corresponding non-zero values [12]. This is very much like the Offset-Value pair approach, except that the 'Offset' in BESS is computed by concatenating the bit encoded representations of the indexes. Let $\langle n_{k-1}, \ldots, n_1, n_0 \rangle$, denote the index of a non-zero array element and let $\beta_j, 0 \leq j < k$, denote the compact binary bit representation of the index value n_j. A Bit Encoded Index (BEI) of such an array element is the integer representation of the concatenation of the β_j's,

$\beta = \beta_{k-1}||\beta_{k-2}||\ldots||\beta_0$. The BEI is then interpreted as a simple integer position code. This can be perceived as the equivalent to the offset index in the preceding discussion of the *Offset-Value* representation. We term the generated BEI also as a *key*. An example of using BESS to represent the MDSA of Figure 2.1 is shown in Table 2.4. The column β is the concatenated bit sequences of the 3 index values for each non-zero element. We use $\lceil \log D_i \rceil (0 \leq i \leq 2)$ bits for each dimension. These are concatenated in the order of $I-J-K$. The storage requirement of BESS is $(C_{il} + C_f)\mathcal{N}_{nz}$. A random array element is accessed in time $O(\log \mathcal{N}_{nz})$, since like the offset-value pair format, a binary search is used to determine whether a non-zero value exists or not given a k-dimensional index $\langle n_{k-1}, \ldots, n_1, n_0 \rangle$.

β	BEI	val
00000000	0	20.5
00000001	1	11.2
00001010	10	17.0
00001011	11	23.6
00100011	35	14.9
00100100	36	15.2
.....		

Table 2.4: The BESS representation of the MDSA in Figure 2.1.

Offset-value pair and BESS are both very efficient in storage space utilization. However, the computational efficiencies of these two storage schemes are limited to using binary search or sequential scan of the non-zero values. The difference between them lies in the computation of the offset in the former and the BEI in the latter. While computing the offset of an array element involves certain number of multiplications and additions, constructing the BEI only takes a number of bit operations, which are considered to be more efficient than the arithmetic operations. A well studied comparison between the offset-value pair and BESS was presented in the work of Goil et al. [12] and the result showed that BESS is much more efficient than the the use of the offset-value pair. As a result, we compare our new methods being introduced in this dissertation, with BESS, but the basic idea of offset-value pair is applied in designing the methods of xCRS and the Hybrid storage scheme.

2.1.4 Compressed Row or Column Storage

Compressed row or column storage (CRS or CCS) is widely used to represent sparse matrices. It uses three one-dimensional arrays, denoted by val, $cind$ and $rptr$ respectively here, to represent a sparse matrix. The array val stores the non-zero values, $cind$ stores the column indexes of each non-zero element and $rptr$ stores

the pointers (position index of the values) of the starting position of each row in the array $cind$. Table 2.5 illustrates an example of CRS representation of a 6×6 matrix A. In CRS, the array elements are traversed in row-major order. CCS is similar to CRS except that the array elements are traversed in column-major order. The random array element access time in CRS is $O(\log D_0)$. However, the actual sizes of the rows vary and can be far less than the cardinality D_0 due to the sparsity of the matrix. Retrieval of a row is much more efficient than of a column in CRS, and vice versa in CCS.

$$A = \begin{pmatrix} 10.2 & 0 & 0 & 0 & 2.34 & 0 \\ 13.7 & 23.5 & 0 & 0 & 0 & 11.6 \\ 0 & 17.3 & 0 & 35.4 & 0 & 0 \\ 53.2 & 0 & 28.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 89.4 & 0 & 13.9 \\ 0 & 44.5 & 0 & 0 & 20.9 & 0 \end{pmatrix}$$

offset	0	1	2	3	4	5	6	...	10	11	12	13
val	10.2	2.34	13.7	23.5	11.6	17.3	35.4	...	13.9	44.5	20.9	13
cind	0	4	0	1	5	1	3	...	5	1	4	13
rptr	0	2	5	7	9	11	13					

Table 2.5: The CRS of matrix A

2.2 Data Warehousing and OLAP

The multi-dimensional data model, which we focus on in this research, is intended to represent primarily the data in data warehousing and OLAP. Data warehousing and OLAP are the core technology in decision support systems. Data warehousing refers to a collection of decision support technologies. It aim at enabling the knowledge workers to make better and faster decisions [5]. Data warehousing has been increasingly deployed in many industries, such as manufacturing, retail, financial services, etc. A data warehouse is an integrated repository that stores information which may originate from multiple, possibly heterogeneous operational or legacy data sources [41]. A data warehouse may contain original raw data or preprocessed data. OLAP, which enables analysts to work with data warehouses interactively, supports dynamic synthesis, analysis, and consolidation of large volumes of data [30].

The data in a warehouse and OLAP system is typically modeled as multi-dimensional data, in the sense that the data can be modeled as dimension attributes and measure attributes [44]. A measure attribute measures some numeric value, such as sales, budget, etc., and dimension attributes define the dimensions on which the

measures are viewed, such as time, location, etc. Thus, the multi-dimensional data views the measures as values in the multi-dimensional space. Further, a dimension can be defined by a set of attributes, forming a hierarchy. For simplicity, we did not take the hierarchy of a dimension into consideration in this research.

The construction, operation, and maintenance of a data warehouse and OLAP server involve many techniques and research issues. An overview of these topics is presented in the work of Chaudhuri et al. [5]. In the evaluations of the new and existing storage schemes, we selected four operations, namely the construction of the storage structure, random element access, sub-array retrieval and multi-dimensional aggregation, as the main performance criteria for comparisons. The random element access and sub-array retrieval are the most basic and yet fundamental operations to constitute the efficient queries over data warehouse, together with the slicing and dicing operations in OLAP. In the materialized views of data warehousing, roll-up and drill-down operations in OLAP use aggregations.

If the data in data warehouses and OLAP is to be represented in an MDSA storage format, the first basic array operation we need to consider is random element access. Data warehouses usually contain large volumes of data that are still sparse in the multi-dimensional space. It is essential to be able to efficiently access the data in order to answer the queries efficiently. Random element access, on one hand, means retrieving the value associated with a given array index. In array structure representation, this could be done conveniently by finding the displacement in a linear address space according to the index. However, when the array is represented in a certain storage scheme, to access an array element using its array index requires using computations according to the schemes. On the other hand, we may need to recover the array indexes of the elements given a position index in a storage scheme. The efficiency of this inverse mapping is actually a critical factor in the performance of computations, such as sub-array retrieval, aggregation, and nearest neighbor queries.

Unlike in operational databases, the operations in data warehousing and OLAP may not often access individual data; instead they often need to access the data in a 'sub-region'. For example, in Figure 2.1, we may only be interested in the data on the dimension I with index value 1 or 2. In this case, we need to extract the data on this specific dimension value efficiently. We refer to this operation as sub-array retrieval. More precisely, given a k-dimensional array $A[D_{k-1}]\dots[D_0]$ and rectilinear boundary indexes $L = \langle l_{k-1}, \dots, l_0\rangle$ and $H = \langle h_{k-1}, \dots, h_0\rangle$ where $l_j \leq h_j, 0 \leq j \leq k-1$, we are required to retrieve all the non-zero elements in the sub-array defined by L and H. Sub-array retrieval constitutes the partial and exact match queries over data warehouses.

To analyze the multi-dimensional data in a data warehouse or MOLAP, an analyst may want to see a summary data in a number of selected dimensions. The summary data is often presented in an m-dimensional array ($1 \leq m \leq k$, k is the dimensionality). Especially, in OLAP, the summary data is often called *cross-tab* or

pivot-table when $m = 2$, and *data cube* when $m > 2$. For example, in Figure 2.2, the aggregation result on the dimensions (J, K) is a *cross-tab*, the aggregation may also be carried on (I, K) or (I, J). These operations, called *pivoting*, are to get different summarized data by aggregating on some of the specific dimensions only, ignoring all the other dimensions, or assuming the special *all* [44] value for them. On the other hand, sometimes we assign fixed values, instead of the *all* value, on some dimensions first, then carry out aggregations on the remaining dimensions. We refer to these operations as *slicing* (specify a fixed value on one dimension only), and *dicing* (specify fixed values on more than one dimension). Besides these operations, OLAP servers also support two other operations, namely *roll-up* and *drill-down*. *Roll-up* refers to the aggregation operations that move from aggregating on more detailed group-by attributes to less detailed ones. *Drill-down*, the inverse of *roll-up*, refers to the aggregation operations that move from less detailed group-by attributes to more detailed ones. The typical OLAP operations we discussed above mainly involve aggregations on varying dimensions or combinations of dimensions, as well as accessing the data frequently on varying parts.

Refreshing or updating a data warehouse raises a number of issues in itself, although we do not consider them in this research. However, we should point out that among the storage schemes we considered, PATRICIA trie compressed storage (PTCS) has the unique advantage of efficiently supporting some update operations, such as insertion and deletion. This is due to the fact that PTCS employs PATRICIA trie structure to store the valid elements in an MDSA. The other storage schemes are based on array structure, where the insertion or deletion of a single element may cause the whole structure to be rebuilt.

2.3 Multi-Dimensional Aggregation

Aggregation is a very important statistical concept to summarize information about large amounts of data [14]. The idea is to represent a set of items by a single value or to classify items into groups and determine one value per group. Aggregation is usually supported in the form of *aggregate functions*, which determine a set of values from an input set of values, such as a relation. Aggregate functions can be classified into three categories: *distributive, algebraic* and *holistic* [16, 1]. An aggregate function $f()$ is distributive if it allows the input data set to be partitioned into disjoint sets that can be aggregated independently and the results combined to obtain the final one. Aggregate functions such as determining the maximum, minimum, summation, count of values, are all distributive. Algebraic aggregate functions, such as computing the average value, can be expressed in terms of other distributive functions. Holistic aggregate functions, such as finding the median or standard deviation, are those that cannot be computed in parts and combined. Aggregation is an important operation in data warehousing and MOLAP applications. Therefore in data analytics and decision support systems. Data warehousing and MOLAP applications typically aggregate data across many dimensions, when

processing queries or looking for anomalies, patterns and trends.

The group of attributes constituting the dimensions along which the aggregates are computed are called the *group-by* attributes or dimensions. Since we do not consider the hierarchy of a single dimension in this research, we use the term dimension and attribute interchangeably. The Figure 2.2 shows the aggregations on the group-bys of a single dimension (K) and two dimensions (I, J) for the MDSA in Figure 2.1. The aggregate function in this example is *summation*. For example, to compute the aggregation on the group-by of K, we group the array elements according to their index value on dimension K. Those elements with the same index value on k are grouped into the same group. The aggregate function is then carried on to each group. Figure 2.3 shows this process. Given a k-dimensional sparse array, aggregating on a group of $m(1 \leq m \leq k)$ dimensions results in an m-dimensional array. Algorithms for aggregate functions require grouping of the input data, and then one output item is computed per group. The typical grouping methods are based on either *sorting* or *hashing*.

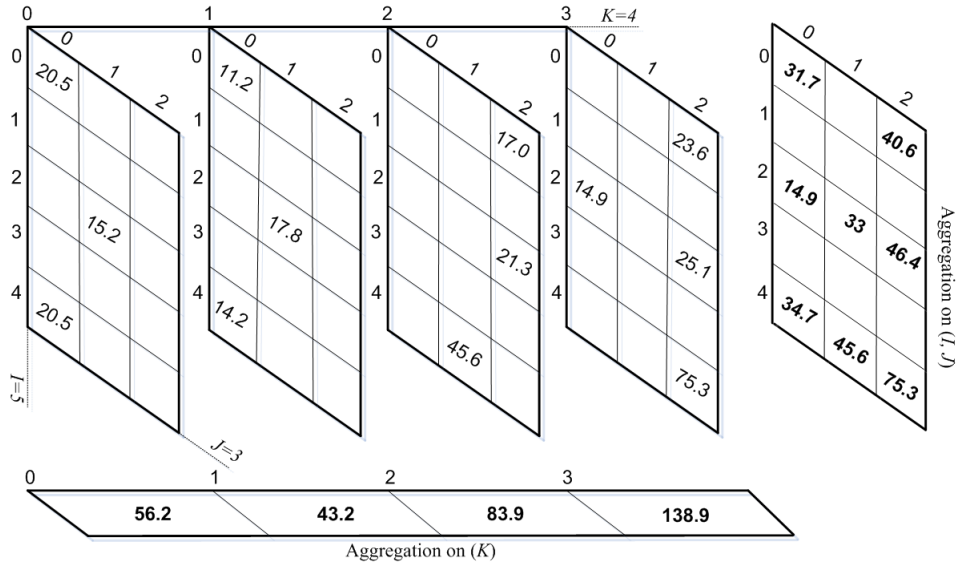


Figure 2.2: The aggregations with the aggregate function as summation, on a single dimension and multiple dimensions

2.3.1 The CUBE

It is often necessary to aggregate data efficiently in data warehousing applications. To make interactive analysis possible in these applications, various levels of aggregations are usually precomputed. To conveniently support multiple aggregations, Gray et al. proposed the "*cube*" operator [16], which computes aggregations over all possible combinations of a set of dimensions. By changing the granularity on the group of attributes or dimensions, different aggregation information can be obtained. For example, to analyze computer sales, one might put special focus on the effects of brand and type of the computers, ignoring all the other attributes. In such a

K	J	I	val	sum(val)
0	0	0	20.5	
0	0	4	20.5	56.2
0	1	2	15.2	
1	0	0	11.2	
1	0	4	14.2	43.2
1	1	2	17.8	
2	1	4	45.6	
2	2	0	17.0	83.9
2	2	2	21.3	
3	0	2	14.9	
3	2	0	23.6	138.9
3	2	2	25.1	
3	2	4	75.3	

Figure 2.3: The aggregation process for the group-by of a single dimension K. The aggregate function is summation.

case, we may analyze the total sale of computers by brand and type respectively, or by some combinations of both. Given a data set with k attributes, the number of all possible combinations of the k attributes is $\sum_{i=0}^{k} \binom{k}{i}$, which is 2^k. This number is exponential with respect to the number of dimensions. Thus, computing the *cube* presents challenges on both speed and space.

A number of efficient methods of computing multi-dimensional aggregation and the *cube* have been developed [1, 38, 51, 17]. While many of these methods have been developed in the context of relational database systems, most of them are applicable to the database systems with different data models. In the following text, some of the basic rules in computing aggregation and the *cube* are outlined, we also present some of the state of the art methods in computing the *cube*.

As Graefe points out in [14], two types of algorithms for aggregation, based on sorting and hashing are often used in standard database aggregation problems. The data items are sorted on their grouping attributes in a sort-based aggregation algorithm. This allows us to compute multiple aggregations in one scan of the data. Furthermore, the output data is also in sorted order and can further be exploited to compute other aggregations. By hashing on the grouping attributes, items of the same group can be found and aggregated in hash-based algorithms.

Some basic techniques for computing the *cube* are outlined in the work of Gray et al. [16]. The objectives of these techniques are to:

- Minimize the data movement.
- Use sorting or hashing to organize the data.
- Map the non-integer types of attribute values to integers starts from zero.
- Use parallelism.

2.3.1.1 Search Lattice

Underlying all the *cube* construction methods for relational data sets is the lattice representation of 2^k combinations of k attributes and their parent-child relationships. Such a lattice for 4 attributes (dimensions) is shown in Figure 2.4. Each node, called a *cuboid*, in the lattice represents some combination of the attributes and the *all*

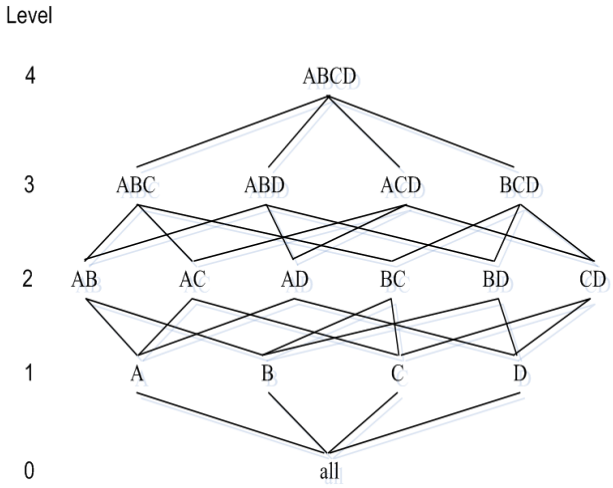


Figure 2.4: A search lattice with 4 attributes.

node represents the empty combination. The edges represent the parent-child relationships between the nodes. There is an edge connecting two nodes if one (child) of them can be computed from the result of the another (parent). For example, the edge between ABC and AB indicates the aggregation on AB can be computed from the result of the aggregation on ABC. Figure 2.4 shows that at each level, a child node may have more than one parent node from which it can be computed. It is necessary to find the parent that minimizes the cost of computing the child node. The algorithm presented in the work of Agarwal et al. [1] found the best way of computing each level i [1] from level $i+1$ by reducing the problem to a weighted bipartite matching problem. Once each child node is assigned to a parent, the search lattice can be partitioned into a number of paths each containing a series of cuboids with parent-child relationships determined. For example, in Figure 2.4, one such

[1]Typically, the *all* node in a search lattice is at level 0, and level i includes all the cuboids with the combinations (group-bys) of i attributes.

path is $ABCD \rightarrow ABC \rightarrow AB \rightarrow A$. To compute all the group-bys on this path, the input data need to be sorted only once according to the attribute order at the root node (node $ABCD$ in this case), and read only once as well. In order to optimize the *cube* computation, it is often desirable to determine a certain number of paths in the search lattice so that the cost of sorting, memory space requirement and disk reads, of computing the *cube* can be minimized. Ross et al. in [38] argued that there must exist at least $\binom{k}{\lceil k/2 \rceil}$ paths in the search lattice. They also showed that this was an upper bound on the number of paths required to cover all the nodes in the search lattice. The algorithm to find $\binom{k}{\lceil k/2 \rceil}$ paths proposed in [38] is further discussed in Chapter 4.

Based on the different approaches to utilize parent-child relationships among the nodes in the search lattice, two methods, top-down and bottom-up, are often used to compute the *cube*. Top-down approach computes less detailed group-bys from more detailed ones, so that one sort operation at the finest level granularities may be shared among the less detailed ones. Bottom-up approach computes more detailed group-bys from less detailed ones so that the partial sort on the less detailed granularities may be used to generate more detailed sorts.

2.3.1.2 Algorithms for Computing the CUBE

Fast algorithms to compute the *cube* operator were introduced in [1] by optimizing sort-based and hash-based grouping methods with several optimizations. These optimization techniques include:

1. Combining common operations across multiple aggregates;
2. Caching the results of a group-by from which other group-bys are computed;
3. Computing as many group-bys as possible while the input data is in memory and sorted in a certain attribute order;
4. Sharing sorting cost in sort-based methods or partitioning cost in hash-based methods across multiple group-bys.

The algorithm *PipeSort* [1], first generates a set of paths using a local optimizing technique based on weighted bipartite matching at each level of the search lattice. These paths are then evaluated in turn. During the evaluation of a path, the data is sorted in the order indicated by the attribute order of the root node in the path. After the data is sorted, all the group-bys in the path are computed in pipelined fashion during a single scan of the data. The number of sorts that *PipeSort* has to perform, is at least $\binom{k}{\lceil k/2 \rceil}$, which is exponential in k, the number of dimensions. Clearly, when the number of dimensions increases, the cost of sorting will dominate the

complexity of the *cube* computation. Thus parallelism of the sort operation, which we will show in Chapter 5, is a viable means to improve the efficiency of computing the *cube*.

The algorithm *Memory-Cube*, introduced in [38], is similar to *PipeSort* in that it also applies the pipelined evaluation of the cuboids in a path. However, *Memory-Cube* tries to generate an optimal set of paths (i.e., the number of sorts) in the search lattice while *PipeSort* does not guarantee this. Further, *Memory-Cube* utilizes considerably sharing of the cost of sorts among different paths.

An array-based algorithm, *Array-Cubing* algorithm, to compute the *cube* for MOLAP systems introduced by Zhao et al. [51] performs well because the array representation allows direct access to the data elements. Most importantly, re-sorting the non-zero values on different sets of aggregate attributes can be done by simply visiting those array elements in the right order. It was also shown that given appropriate data compression techniques, the *Array-Cubing* algorithm could be significantly faster than the relational algorithms, and it might also be used for relational OLAP. Array-Cubing algorithm organizes large arrays in chunks, and the array chunks in turn are organized using offset-value pair storage scheme. The algorithm computes a minimum memory spanning tree (MMST) for a given dimension order, based on the choice of the parent node that minimizes the memory required for computing a child node at each level. Generally, if the required memory for each MMST node is allocated, the computation of the nodes can be done concurrently. However, this is not always the case for large sparse data sets with high dimensions.

Harinarayan and others [17] investigated the issue of how to determine the set of aggregates, on different combinations of dimensions, to be computed in the case where the whole *cube* is too expensive to be materialized. A greedy algorithm was proposed to work on a lattice framework by determining an optimal set of aggregates to be computed, while the remaining ones can be computed from the materialized ones with minimum cost if such a need arises at query processing time. The algorithm, termed the *Greedy Algorithm*, was designed on the basis of a linear cost model. In the linear cost model, the cost of computing a child node, in the search lattice, is the size of the parent node. The *Greedy Algorithm* determines a set of predetermined number of nodes to materialize by comparing the benefits of one new parent node over the other already selected ones in the set. A new node is added to the set only if the benefit of that node is greater than the benefits of those that are compared with. The total benefit of a new node v is the sum over all the child nodes w of v of the benefit using v to evaluate w. Suppose we denote the cost of a node v as $C(v)$, which is the space cost associated with that node [17], the benefit of v over a competitor node u is either $C(u) - C(v)$ if $C(v) < C(u)$, or 0 otherwise.

While there are various efficient sequential algorithms to compute multi-dimensional aggregations, the following works showed the effectiveness of parallelizing the same computation.

- A general methodology for the efficient parallelization of the existing data *cube* construction algorithms was described by Dehne et al. in [7], which supports the transfer of optimized sequential data *cube* code to a parallel setting.

- MCMD-CUBE, a new parallel data *cube* construction method for multi-core processors with multi-disks introduced by Dehne et al. in [8], could achieve close to linear speed-up on the *cube* construction. Based on PipeSort, MCMD-CUBE parallelizes the computation of each of the paths PipeSort generates as follows. First apply a parallel external memory sort, MCMD-SORT, on the root cuboid of a path. Then aggregate the sorted cuboid in parallel. After carefully dividing the workload among the processors, MCMD-SORT utilizes STL sort of the STL library and a merging algorithm based on deterministic sampling method to get the entire data sorted. The aggregation on the sorted data is parallelized by first partitioning the data into smaller segments. Then each segment is aggregated in parallel, followed by computing the aggregates across the segment boundaries in parallel as well. Finally the child cuboids following the root cuboid in the path, are computed in parallel level by level using their parent cuboid.

2.4 General Purpose Computing Using GPUs

Computer graphics chips, known as Graphics Processing Units or GPUs, are primarily designed to process interactive 3D graphics efficiently. The use of computer graphics hardware for non-graphics or general purpose computation, collectively known as General Purpose Computing on GPU (GPGPU), has been an active research area for many years. GPUs have been successfully applied as co-processors to CPU to accelerate a broad range of applications, such as seismic database, molecular dynamics, and MRI processing [34]. The state of GPGPU can be best put as the statement that appeared in [36]: "The rapid increase in the performance of the graphics hardware, coupled with recent improvements in its programmability, have made graphics hardware a compelling platform for computationally demanding tasks in a wide variety of application domains."

Three prominent features of GPU, which have been harnessed for general purpose computing, are computational power, high memory bandwidth and low energy consumption. For example, NVIDIA's Tesla K40 offers a capacity of 2880 cores, 12GB device memory with 288GB/sec bandwidth, and peak single precision floating point performance at 4.29 Teraflops. Along with the rapid developments in hardware, GPUs have become more flexible and programmable with every major generation of the products. Besides graphics APIs, such as OpenGL and DirectX, high level GPU programming models have been emerged to meet the demands of general purpose computing. For example, NVIDIA's CUDA [32] allows programmers to use C/C++, Python, or Fortran language extensions to program GPUs. The promising hardware features and programmability of GPUs make it possible to realize supercomputing with a PC. However, not all problems can be mapped effi-

ciently onto GPUs. It is hard to define such a boundary so as to tell what kind of applications should run better on GPUs or CPUs. Vuduc et al. present some discussions on the limits of GPUs on general purpose computing in their work [47, 27].

2.4.1 GPU Architecture

Driven by the ever-increasing demand for real time, high-resolution 3D graphics rendering, GPUs have been designed to provide a large number of simple, highly parallel, multi-threaded cores with very high memory bandwidths. The following context in this section is primarily based on NVIDIA GPU. The Nvidia GPU architecture is built around a scalable array of multi-threaded *Streaming Multiprocessors (SMs)*. Figure 2.5 shows a modern NVIDIA GPU architecture. Each SM has a number of *streaming processors (SPs)* that share the control logic and instruction cache. In Figure 2.5, 2 SMs form a building block, and each SM has 8 SPs. Note that these numbers differ with each new generation of GPUs. The GPU device memory (global memory in Figure 2.5) is typically in the amount of several gigabytes for the low end product line, and several tens to hundreds of gigabytes for the high end. Compared with the system memory (DRAMs) on the CPU motherboard, the GPU device memory has higher bandwidth and higher access latency. However, it is often possible to hide the slow access to global memory by utilizing the high memory bandwidth in coalescing accesses to memory among multiple threads. Moreover, each SM has a on-chip local memory, although small in size, which is shared among the SPs and has very low access latency.

A GPU supports thousands of concurrent threads due to the fact that each SP itself is designed to execute hundreds of threads concurrently. Compared with CPU threads, GPU threads are extremely light-weighted and the thread management, such as creation, scheduling, and synchronization, is done by hardware in SMs with very little cost. The number of GPU threads could surpass the one of CPU's easily by thousands. To manage such a large amount of threads, GPU employs a unique architecture called *Single Instruction Multiple-Threads (SIMT)*. In SIMT, a multiprocessor creates, manages, schedules, and executes threads in groups. For example, the SMs on a NVIDIA GPU typically organize the threads in groups of 32 threads called warps [32]. A thread group executes one common instruction at a time. If the execution path diverges, such as in a conditional statement, within a group, then each path will be executed serially. When all the paths are executed, the threads in the same group converge back to the same execution path. The SIMT architecture performs best while executing non-divergent data parallel codes.

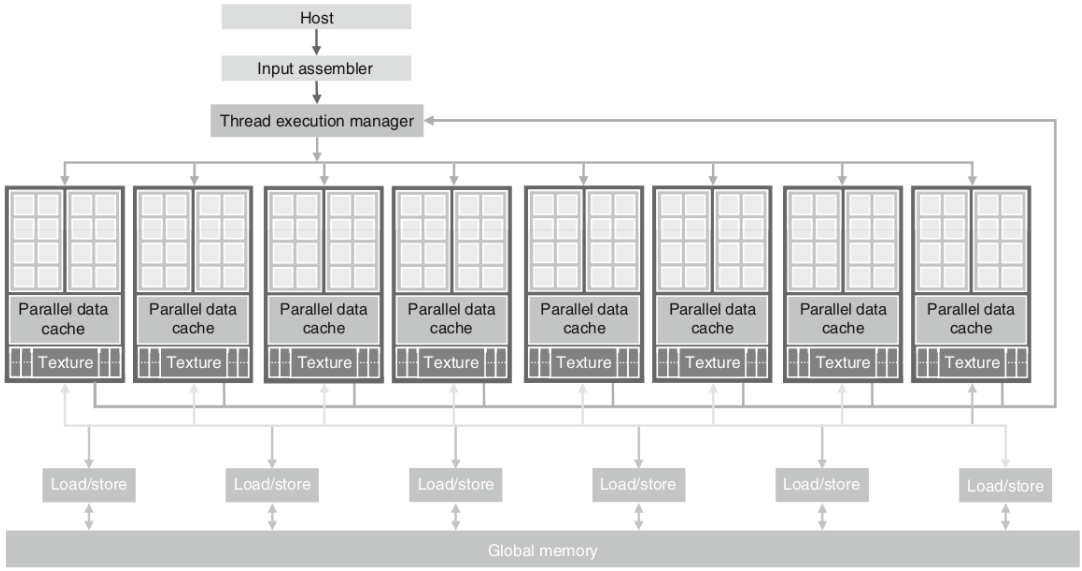


Figure 2.5: A modern NVIDIA GPU architecture [24].

2.4.2 CUDA Programming Model

Along with the GPU hardware evolution from fixed function devices into increasingly flexible programmable processors, high level, general purpose GPU programming models have also emerged. NVIDIA's CUDA [32], introduced in 2006, is one such programming model. CUDA abstracts the parallel features of GPUs to high level programming language extensions, in C and C++. The key abstractions include a hierarchy of thread group, shared memories and barrier synchronizations. The main concepts behind CUDA programming model can be summarized as follows.

Kernel: *Kernel* is equivalent to a *function* in the C language. A kernel is callable from the host and executed on the device by many threads in parallel. A kernel launch requires specifying the execution configuration which defines the number of threads, multi-processors, etc., to execute the kernel.

Thread Hierarchy: Threads in CUDA can be identified using a 1-dimensional, 2-dimensional or 3-dimensional *thread index*, forming 1-dimensional, 2-dimensional or 3-dimensional blocks . This provides a natural way to invoke computation across the elements in a data structure such as 1, 2 or 3-dimensional array.

Memory Hierarchy: CUDA provides multiple memory spaces to its threads, including per-thread registers and local memory, per-block shared memory, and global memory, plus two read-only memory spaces, constant and texture memory.

Heterogeneous Programming: CUDA programming model assumes that the *kernels* are executed on a separate *device*, the GPU, that acts as a co-processor to the *host*, CPU, which runs the rest of the program. It

also assumes that the *host* and *device* maintain their own memory spaces, referred to as *host memory* and *device memory* respectively. In CUDA, the communication between the host and device is via the PCIe bus, which has a much lower bandwidth compared with the device memory bandwidth. The dynamic memory allocation on the device memory, I/O of the device are done by the host.

Barrier Synchronization: CUDA threads within the same block can be synchronized by a synchronization function being called in a kernel. The thread that executes the synchronization function will be held at the calling location until every thread within the block reaches that location. However, CUDA does not allow threads in different blocks being barrier synchronized. This allows CUDA runtime system to execute the thread blocks in any order.

As mentioned above, CUDA allows programmers to access several types of memory space, see Figure 2.6. These memory spaces differ in size and access latency. Appropriate use of these memory spaces can have significant performance implications. The registers and shared memory are on-chip memory, thus they can be accessed extremely fast. Each thread has one set of local registers, and each multiprocessor or block has its own parallel data cache or shared memory. These two types of memory space are very scarce in sizes. Local memory is usually used to hold automatic variables, which belong to individual thread, either in the registers or in the global memory. Local memory is not an actual hardware component of the multi-processor. The read-only constant memory is shared by all the threads. It is implemented as a read-only region of global memory. The variables in the constant memory (or constant variables) are usually cached for efficient access. The read-only texture memory is also implemented as a read-only region of device memory. The texture memory provides an alternative memory access path to some regions of the global memory. The global memory has the largest size and highest access latency. For more details on CUDA memory hierarchy, we refer the readers to [32, 24].

One of the limitations of CUDA is that it is hardware specific. Using CUDA makes it difficult to access other types of processing units within a single multi-platform source code. Open Computing Language (OpenCL) is an open standard for general purpose parallel programming of heterogeneous systems that include CPUs, GPUs, and other types of processors, such as digital signal processors (DSPs) [23]. OpenCL programming model can easily be mapped onto that of CUDA in their execution models and memory hierarchies [33]. However, OpenCL has a more complex platform and device management model that reflects its support for cross platform, cross vendor portability. Another important parallel programming API for heterogeneous computing system is OpenACC [35]. The OpenACC API describes a collection of compiler directives to specify loops and regions of code in C, C++, and Fortran to be offloaded from a host CPU to an attached accelerator, such as GPU. Directives, similar to "#pragma" directives in OpenMP, are simple hints provided to the compiler. The complier then attempts to generate parallel kernel code for a GPU or other types of processor. OpenACC

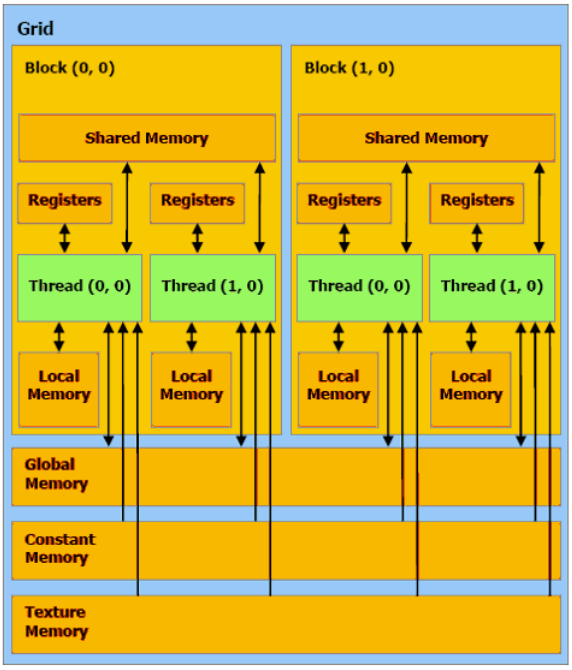


Figure 2.6: CUDA memory hierarchy [32].

allows programmers to create high level heterogeneous programs without the need to explicitly program the accelerator or launch the kernel on it.

2.5 Application of GPUs to Data Warehousing

We give an overview, in this section, of the applications of GPUs to data warehousing, database management and data mining. For a broader range of applications and GPGPU techniques, we refer the readers to the work of Owens et al. [36] and Hwu [21]. To address the data movement overheads in data warehousing applications, Wu et al. [49] proposed a set of compiler optimizations, *kernel fusion/fission*, which were inspired by *loop fusion/fission* optimizations in the scientific computing. Using kernel fusion, one new functionally equivalent kernel is created from two or more dependent kernels. Thus reduces the data traffic in PCIe bus, and creates larger body of code for compiler optimizations. Kernel fission, on the other hand, tries to hide PCIe transfer time by overlapping kernel execution and data transfer using CUDA *streams* [32]. A *stream* is a sequence of commands that execute in order. Multiple streams can be created to one GPU. Different streams may execute their commands out of order with respect to one another or concurrently.

Many works have been done on using GPUs to accelerate database applications. In particular, He et al. [18] designed and implemented an in-memory relational query co-processing system, termed GDB, using GPUs. The GDB supports a number of common relational query processing operators, namely selection, projection,

ordering, aggregation and join. These operators were implemented using a set of data-parallel primitives such as *map*, *split* and *sort*. Each operator in GDB tries to utilize effectively the computation resources based on a co-processing scheme. The co-processing scheme in this regard takes into consideration the costs of data movements between CPU and GPU, data partitioning between CPU and GPU. Moreover, He et al. [10] pointed out that the overhead of data transfer between CPU and GPU through PCIe bus might outweigh the benefit of GPU acceleration and suggested the use of data compression technique to alleviate the performance problem.

The parallel data mining system, *GPUMiner*, introduced by Fang et al. [11], demonstrated the effectiveness of utilizing GPUs to two of the data mining algorithms, *k-means clustering* and *Apriori frequent itemset mining (FIM)*. GPU is a co-processor to CPU in GPUMiner. As such, the tasks for each algorithms are carefully divided between CPU and GPU in such a way that both computing resources can be utilized optimally. Another feature in GPUMiner is that the use of "regular-shaped" data structures, e.g., bitmap, that are suitable for the GPUs. Counting is a core operation in both k-means clustering and Apriori FIM. K-means clustering counts the number of data objects associated with a cluster, and Apriori counts the number of transactions containing the same item. These associations in both cases are represented using bitmaps, i.e., 2-dimensional arrays of binary bits, in GPUMiner. This data structure suits GPUs well, especially in the case of trie-based Apriori algorithm.

Sorting is a fundamental building block for many applications in different domains, such as database systems [15], and scientific computing. This is no exception for data warehousing applications as well. Sorting is a crucial part in sort-based aggregation problem. The importance of sorting has led to the design of many sorting algorithms for different parallel platforms, including heterogeneous systems using GPUs [13, 43, 28]. As part of our algorithm in computing the *cube* operator, we utilized the GPU based radix sort from the Thrust [20], a C++ template library for CUDA.

Chapter 3

Multi-Dimensional Sparse Array Representations

3.1 Methodology

The multi-dimensional array is widely used to represent multi-dimensional data model in various kinds of applications. The storage requirement for a k-dimensional array $A[D_{k-1}] \ldots [D_1]\,[D_0]$ is $c\mathcal{N}_k$, where $\mathcal{N}_k = \prod_{i=0}^{k-1} D_i$, for some constant c that depends on the data type of the array elements. The access time for a random array element is $O(k)$ for a dense array. On the other hand, if the array is sparse, i.e., the majority of the array elements are zero or have the same value, storing the whole array will result in inefficient space usage. This is especially the case when the dimensionality and sparsity of the array both become high, while the number $\mathcal{N}_{nz}$ of valid array elements, or non-zero values, remain relatively small. In such cases only the valid array elements need to be stored in such a way that the storage overhead is minimized, while an element is still accessed efficiently.

The basic methodology we follow in designing a storage scheme for MDSAs is to find a function $f()$ that maps any given k-dimensional sparse array space to an r-dimensional array space for $1 \leq r < k$. Typically values for r are 1 or 2. Such a function $f()$ must be an invertible (or bijective) function. Given any array index, $\langle n_{k-1}, \ldots, n_1, n_0\rangle$, of a k-dimensional array element, the function $f()$ should map it to a unique array index, $\langle p_{r-1}, \ldots, p_1, p_0\rangle$, in the corresponding r-dimensional array space. The function must also have an inverse $f^{-1}()$, that maps the index $\langle p_{r-1}, \ldots, p_1, p_0\rangle$ back to $\langle n_{k-1}, \ldots, n_1, n_0\rangle$ in the k-dimensional array. We summarize this bijective mapping, termed *dimensional mapping*, as follows.

$$\langle n_{k-1}, \ldots, n_1, n_0\rangle \xleftrightarrow[f^{-1}()]{f()} \langle p_{r-1}, \ldots, p_1, p_0\rangle \tag{3.1}$$

One can conceive that the index $\langle p_{r-1}, \ldots, p_1, p_0\rangle$ addresses an element in an r-dimensional array space $F[M_{r-1}] \ldots [M_1]\,[M_0]$ that is also linearized into a sequence of consecutive locations $\mathcal{L}[s] = \mathcal{L}\langle 0\rangle, \mathcal{L}\langle 1\rangle, \ldots, \mathcal{L}\langle s-1\rangle$, or formed into $r-1$ independent vectors. For simplicity, we termed the resulting r-dimensional array index in the mapping above as the key. According to the *dimensional mapping*, both *offset-value pair* and *BESS* map a k-dimensional sparse array space to one-dimensional array space, so that the resulting *key* is

simply a scalar value. It is an offset value in the case of the former, and bit encoded index (an integer) in the case of BESS.

The dimensional mapping enables us to transform the array indexes in k-dimensions to the ones in a lower r-dimensions where often $r = 1$ or $r = 2$. Consequently, the original array indexes can be represented by more compact keys. To represent an MDSA, we construct the keys for those non-zero array elements only, and a *key-value* pair for each non-zero element is stored. The question, which naturally arises here, is how to organize these key-value pairs? The simplest way, yet efficient in storage utilization, is to store the keys and their corresponding non-zero values, in a number of one-dimensional arrays, or vectors, such as in the cases of index-value pair, offset-value pair and BESS. The drawback of these methods is that accessing a random element in the resulting structure can be done at best in $O(\log \mathcal{N}_{nz})$. It is possible to improve the data access efficiency by storing some 'extra' information, together with the key-value pairs, about the array structure. In the following sections, we introduce four new approaches for storing large multi-dimensional sparse arrays. Besides giving detailed descriptions of the methods, we also present some of the algorithms on their respective three basic operations. These operations include the following.

Construction: This operation constructs the data structure such as array or trie under a certain scheme to store the valid array elements and their corresponding keys from the inputs in MM-format.

Random Element Access (or Searching): Given an array index, $\langle n_{k-1}, \ldots, n_0 \rangle$, this procedure determines whether the corresponding array element exists or not, and retrieves the actual value if it exists.

Sub-Array Retrieval: Given the starting index, $L = \langle l_{k-1}, \ldots, l_0 \rangle$, and ending index, $H = \langle h_{k-1}, \ldots, h_0 \rangle$, of a sub-array, where $l_j \leq h_j$ for $0 \leq j \leq k-1$, the procedure retrieves all the valid array elements within the sub-array bounded by L and H.

3.2 Extended Compressed Row or Column Storage

Extended compressed row or column storage explores applying the idea of CRS or CCS for higher k-dimensional ($k > 2$) sparse arrays. This format is able to achieve the most efficient data access performance among the storage formats considered in this research. However, it suffers from poor storage utilization.

3.2.1 XCRS and Its Construction

We extended the basic ideas of CRS/CCS to map sparse arrays of k-dimensions ($k > 2$) to 2-dimensional array space. The result of such a mapping is that the original array can be defined as a total number of $\prod_{i=1}^{k-1} D_i$ rows or one-dimensional arrays. Alternatively, the array can also be defined as a number of columns depending on the traversal order of the elements in the 2-dimensional space. We refer to these extensions of CRS/CCS as *Extended Row* or *Column* Storage and they are abbreviated as xCRS or xCCS. Due to the similarity of xCRS and xCCS, we consider only xCRS in the rest of the dissertation. The same ideas can be easily applied to xCCS.

In the offset-value pair, an *offset* of a non-zero element is the displacement or relative address of the element in a linear address space. This definition can be extended to define an *offset* for an m-dimensional ($1 \leq m \leq k$) array in a linear address space. In xCRS, we set $m = 1$, and term the one-dimensional vectors as *rows*. The *row offset*, in xCRS, is defined using the leading $k-1$ index values, i.e., $\langle n_{k-1}, \ldots, n_2, n_1 \rangle$, and computed as $\sum_{i=1}^{k-1} n_i \prod_{j=1}^{i-1} D_j$ for an array element.

The *dimensional mapping* in xCRS allows us to obtain a unique key for each array element, using its *row offset* and the index value on the dimension d_0. Formally, given any array element with index, $\langle n_{k-1}, \ldots, n_1, n_0 \rangle$, in a k-dimensional array $A[D_{k-1}] \ldots [D_1] [D_0]$, the xCRS key for this element is defined as an index, $(\sum_{i=1}^{k-1} n_i \prod_{j=1}^{i-1} D_j, n_0)$, in a 2-dimensional array space.

We represent the xCRS key-value pairs for the non-zero elements in an MDSA following the idea of CRS as described below. The non-zero values and their corresponding index values on the dimension d_0 are stored in two one-dimensional arrays, denoted by val and $cind$, in row-major order respectively. We use another one-dimensional array, $rptr$, to store the starting positions for each row in $cind$ (or val). If a row in xCRS has no valid array element, the starting position of such a row is indicated by a special value, denoted by "-1". Note that each row in xCRS has an entry in array $rptr$, which is located according to its *row offset*. The xCRS representation of the 3-dimensional array example in Figure 2.1 is shown in Table 3.1.

To implement xCRS, we only need to construct the three one-dimensional arrays, namely val, $cind$, and $rptr$, by reading the input data once. This process is simple when the input data is already in the desired order, i.e., row-major order. Otherwise, preprocessing of the input data is necessary. An algorithm to construct xCRS is given in Algorithm 1. In this algorithm, the statements between Lines 4 and 11 are for updating array $rptr$. Line 4 finds the difference between the current row and previous row visited. If the difference is greater than 1, then there are some unoccupied rows between the current row and previous one. In such a case the value of

-1's are inserted (Line 7) for these rows in $rptr$.

offset	0	1	2	3	4	...	8	9	10	11	12	13	14	15
val	20.5	11.2	17.0	23.6	14.9	...	25.1	20.5	14.2	45.6	75.3	13		
$cind$	0	1	2	3	3	...	3	0	1	2	3	13		
$rptr$	0	-1	2	-1	-1	...	7	-1	-1	-1	9	11	12	13

Table 3.1: The xCRS representation of the MDSA in Figure 2.1.

Algorithm 1: xcrsConstruct($n, v, val, cind, rptr, ctr, lastvisited$)

Input: An index-value pair, $n[0..k-1]$ and v; xCRS arrays val, $cind$, $rptr$; a counter array $ctr[0..1]$ and the last visited row index $lastvisited[0..k-2]$.

Output: The xCRS arrays with the data v and relevant indexes are inserted.

```
begin
    cind[ctr[1]] ← n[0]
    val[ctr[1]] ← v
4   tmp ←getRowOffset(n) - getRowOffset(lastvisited)
    if tmp ≥ 1 then
        while tmp > 1 do Insert −1s for unoccupied rows
7           rptr[ctr[0]] ← −1
            ctr[0] ← ctr[0] + 1
            tmp ← tmp − 1
        rptr[ctr[0]] ← ctr[1]
11      ctr[0] ← ctr[0] + 1
    ctr[1] ← ctr[1] + 1
    /* Update the last visited row                                   */
    if lastvisited[0..k − 2] ≠ n[1..k − 1] then
        lastvisited[0..k − 2] ← n[1..k − 1]
    return
```

3.2.2 Random Element Access and Sub-Array Retrieval in xCRS

To access an array element, with its index as $\langle n_{k-1}, \ldots, n_1, n_0 \rangle$, in xCRS, we first compute its *row offset*, denoted by row_offset, then visit the array $rptr$ at the entry $rptr[row_offset]$. If the value of the entry is -1, that particular row has no valid array elements. Otherwise, the row has valid elements, and they are stored within a range which starts at position $rptr[row_offset]$ in val. The range ends at the starting position of the next occupied row. Once the starting and ending positions of the row is determined, we use *binary search* algorithm to retrieve the index value n_0 within that range in $cind$. If it is found, the value of this array element can be retrieved in the array val at the same position as n_0 in the array $cind$. This process is shown in Algorithm 2. The time complexity of a random element access in xCRS is $O(\log D_0)$ if the corresponding

row is occupied, and $O(k)$ otherwise. Due to the sparsity of the array, the actual size of a row in the array $cind$ (or val) is usually much less than the cardinality, D_0, on the dimension d_0.

Algorithm 2: xcrsSearch($val, cind, rptr, n$)

Input: XCRS arrays and the array index $n[0..k-1]$ of an element
Output: Returns the corresponding value v if the array element exists, or returns a $FALSE$ value.
begin
  $FALSE \leftarrow -1$
  $p \leftarrow$ getRowOffset($n[1..k-1]$)
  $q1 \leftarrow rptr[p]$
  if $q1 \neq -1$ **then**
    $q2 \leftarrow rptr[p+1]$
    if $q2 = -1$ **then**
      $tmp \leftarrow p+2$
      while $rptr[tmp] = -1$ **do**
        $tmp \leftarrow tmp+1$
      $q2 \leftarrow tmp$
    $v \leftarrow$ binarySearch($cind, val, q1, q2-1, n[0]$)
  else
    $v \leftarrow FALSE$
  return v

The random element access operation can be easily extended to retrieve the elements in a sub-array. In xCRS, the elements in a sub-array are often distributed in the array val with a constant stride. Within a single row, only a fraction of the elements (with contiguous addresses) might fall in the range of the sub-array, and every two neighboring fractions have the same stride (or distance). Based on this property, we can retrieve the elements in a sub-array by visiting the relevant *rows* one by one. We determine the range of a row in the same way as we did in the random element access. Once a row is located in $cind$, we examine each index value, by comparing the value n_0 with l_0 and h_0, to determine if it belongs to the sub-array. Note that the computation of the row offsets can be avoided by simply incrementing the row offsets by the constant stride. An algorithm to retrieve the non-zero elements of a given sub-array in xCRS is given as Algorithm 12, Appendix A. The time complexity of sub-array retrieval in xCRS is linear with respect to the number of rows that lie in the sub-array bounds.

3.2.3 Space Utilization of xCRS

Given a k-dimensional sparse array $A[D_{k-1}]\dots[D_1][D_0]$, and the sizes of the three xCRS arrays val, $cind$ and $rptr$ denoted as $\mathcal{S}_{val}$, $\mathcal{S}_{cind}$ and $\mathcal{S}_{rptr}$ respectively, the storage space usage of xCRS, $\mathcal{S}_{xcrs}$, is computed as

$$\mathcal{S}_{xcrs} = \mathcal{S}_{val} + \mathcal{S}_{cind} + \mathcal{S}_{rptr} = (C_f + C_i)\mathcal{N}_{nz} + C_{il}\mathcal{N}_{k-1} \tag{3.2}$$

where $\mathcal{N}_{k-1} = \prod_{i=1}^{k-1} D_i$. Such a space requirement is not optimal compared with the space requirement of $c\mathcal{N}_k$ using multi-dimensional array structure, except that $\mathcal{N}_{k-1}$ is much less than $\mathcal{N}_k$. The storage space requirement for xCRS becomes worse when the sparsity of the multi-dimensional array is very high. This is because the xCRS array $rptr$, which stores the starting positions of each row, may have become sparse itself. In the following sections, we discuss two different approaches to improve the storage utilization of xCRS while sacrificing its data access efficiency as little as possible.

3.3 Bit Encoded Extended Compressed Row Storage

In xCRS representation of MDSAs, the array $rptr$ easily becomes sparse itself when the sparsity of the MDSA is considerably high (say $\sigma > 90\%$). Recall that we define sparsity as $\sigma = 1 - \rho$. To address this issue, we applied a bitmap data compression method, following the idea of Word-Aligned Hybrid (WAH) code [50], to compress the array $rptr$. We term the resulting representation of an MDSA as Bit Encoded Extended Compressed Row Storage (BxCRS).

3.3.1 Word-Aligned Hybrid Code

WAH is a fast bitmap compression scheme that is based on the idea of run-length encoding. It not only offers good compression, but also supports fast bitwise logical operations that lead to improved query response time. In WAH, there are two types of regular words: *literal* words and *fill* words. The implementation in the work of Otoo et al. [50], used the most significant bit of a word to distinguish between a literal word (0) and a fill word (1). Let us denote a computer word length in bits as W. The $W-1$ bits following the most significant bit in a literal word contain the literal bit values from the original bitmap. The second most significant bit in a fill word indicates the type of the fill bits (0 or 1), and the remaining $W-2$ bits store the fill length. For example, assume that $w = 32$, a word $C0000002$ (Hex [1]) in WAH is interpreted as a fill word of fill bit 1, and the fill length is 2. If it is decompressed, the resulting bitmap is 62 bits of 1's. A WAH literal word, say $7111A023$

[1] Hex stands for Hexadecimal number.

(Hex), when decompressed, the resulting bitmap is the same as the lower 31 (or $w-1$ in general) bits in the literal word itself. Besides regular words, WAH uses an active word to store the last bits (less than $W-1$) that could not be stored as literal or fill words.

We applied WAH with slight modification in compressing the bitmap of the xCRS array $rptr$. The modification is that we restrict the lower $W-2$ bits in a fill word to be always 1. This restriction on the fill word enables us to have random access to any row using its row offset.

3.3.2 BxCRS and Its Construction

We construct BxCRS for an MDSA based on its xCRS representation. The process is carried out in the following 4 steps.

Step 1: Construct a bitmap for the xCRS array $rptr$. Each element in $rptr$ corresponds to an entry in the bitmap array. An entry in the bitmap is either 0 or 1 according to whether its corresponding entry in $rptr$ being -1 or not.

Step 2: Group the entries in the bitmap into groups of $W-1$ (W is the bit length of a computer word), then represent each group as a computer word.

Step 3: Transform each word into either a literal word or fill word by examining its value. The resulting compressed words are stored in a one-dimensional array, $compwrd$. Algorithm 3 shows the BxCRS bitmap compression procedure.

Step 4: Eliminate the entries with "-1" values in $rptr$ and store only the starting positions of those rows with valid array elements. The resulting array is denoted as $rptr_c$.

The process of compressing the array $rptr$ and the final BxCRS representation of the example array in Figure 2.1 are shown of Figure 3.1 and Table 3.2 respectively. The BxCRS represents a k-dimensional sparse array using four one-dimensional arrays. In Table 3.2, the arrays val and $cind$ are the same as the corresponding arrays in the xCRS representation (see Table 3.1); the array $rptr_c$ now contains only the starting positions of those rows which have at least one non-zero element; the fourth array $compwrd$ stores the compressed words of the bitmap constructed from the array $rptr$ in the xCRS representation.

Algorithm 3: compressBitmap($rptr_bitmap, total_row_no$)

Input: A bitmap array $rptr_bitmap[..]$, $total_row_no$

Output: An array, $compwrd$, with compressed words and the number of regular words $no_regular_wrds$

begin

`/* W is the bit length of a computer word */`

$wrdcnt \leftarrow 0$

for $i = 1$ *to* $total_row_no/(W-1)$ **do**

Group $W-1$ entries in $rptr_bitmap$ into a word wrd

if $wrd = onefill$ **then** The bits in wrd are all 1

Construct a fill word of bit 1 and store it to $compwrd[wrdcnt]$

$wrdcnt \leftarrow wrdcnt + 1$

else if $wrd = 0$ **then** the bits in wrd are all 0

Construct a fill word of bit 0 and store it to $compwrd[wrdcnt]$

$wrdcnt \leftarrow wrdcnt + 1$

else The bits in wrd are mixed

Store the wrd to $compwrd[wrdcnt]$

$wrdcnt \leftarrow wrdcnt + 1$

$no_regular_wrds \leftarrow wrdcnt$

`/* Group the remaining bits into one active word */`

Construct the active word and store it into $compwrd[wrdcnt]$

return $compwrd$ and $no_regular_wrds$

The storage space usage of BxCRS, $\mathcal{S}_{bxcrs}$ is computed as the sum of the sizes of the four BxCRS arrays, denoted as $\mathcal{S}_{val}$, $\mathcal{S}_{cind}$, $\mathcal{S}_{rptr_c}$ and $\mathcal{S}_{compwrd}$ respectively.

$$\begin{aligned} \mathcal{S}_{bxcrs} &= \mathcal{S}_{val} + \mathcal{S}_{cind} + \mathcal{S}_{rptr_c} + \mathcal{S}_{compwrd} \\ &= (C_f + C_i)\mathcal{N}_{nz} + C_{il}\mathcal{N}_o + C_i\lceil \frac{\mathcal{N}_{k-1}}{W-1} \rceil \end{aligned} \tag{3.3}$$

where $\mathcal{N}_o$ is the number of occupied rows. The difference between $\mathcal{S}_{xcrs}$ and $\mathcal{S}_{bxcrs}$ lies in the difference between $\mathcal{S}_{rptr}$ in xCRS and $\mathcal{S}_{rptr_c} + \mathcal{S}_{compwrd}$ in BxCRS. To illustrate this difference, let us first define a variable γ as the ratio of the number of occupied rows to the total row number, i.e., $\gamma = \mathcal{N}_o/\mathcal{N}_{k-1}$. The ratio of $\mathcal{S}_{rptr_c} + \mathcal{S}_{compwrd}$ to $\mathcal{S}_{rptr}$ is then computed as follows. For the definitions of the other notations, see

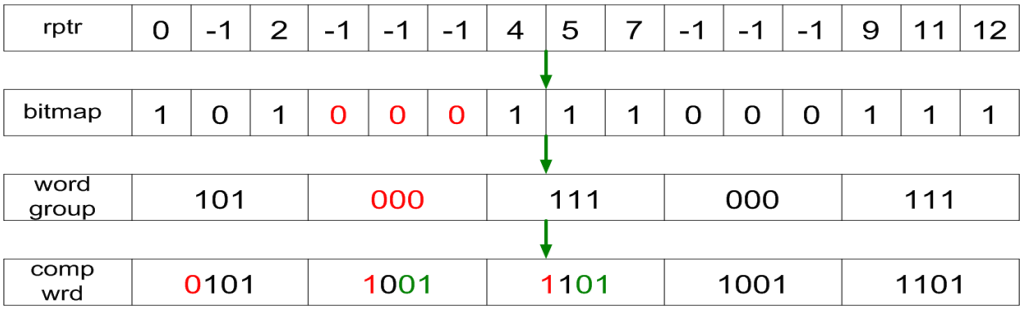


Figure 3.1: The process of compressing array $rptr$. (The word length is 4)

offset	0	1	2	3	4	...	7	8	9	10	11	12	13
val	20.5	11.2	17.0	23.6	14.9	...	21.3	25.1	20.5	14.2	45.6	75.3	13
cind	0	1	2	3	3	...	2	3	0	1	2	3	13
rptr_c	0	2	4	5	7	...	12	13					
compwrd	0101	1001	1101	1001	1101								

Table 3.2: The BxCRS representation of the MDSA in Figure 2.1.

Table 2.1.

$$\begin{aligned}\frac{\mathcal{S}_{rptr_c}+\mathcal{S}_{compwrd}}{\mathcal{S}_{rptr}} &= \frac{C_{il}\mathcal{N}_o + C_i\lceil\frac{\mathcal{N}_{k-1}}{W-1}\rceil}{C_{il}\mathcal{N}_{k-1}} \\ &= \frac{C_{il}\gamma\mathcal{N}_{k-1} + C_i\lceil\frac{\mathcal{N}_{k-1}}{W-1}\rceil}{C_{il}\mathcal{N}_{k-1}} \\ &= \gamma + \frac{C_i}{C_{il}\mathcal{N}_{k-1}}\lceil\frac{\mathcal{N}_{k-1}}{W-1}\rceil \\ &\leq \gamma + \frac{C_i}{C_{il}\mathcal{N}_{k-1}}(\frac{\mathcal{N}_{k-1}}{W-1}+1) \\ &= \gamma + \frac{C_i}{C_{il}(W-1)} + \frac{C_i}{C_{il}\mathcal{N}_{k-1}}\end{aligned}$$

For a certain MDSA, $\frac{C_i}{C_{il}(W-1)} + \frac{C_i}{C_{il}\mathcal{N}_{k-1}}$ is a constant value, and $\frac{C_i}{C_{il}\mathcal{N}_{k-1}} \ll \frac{C_i}{C_{il}(W-1)}$. Hence, the compression ratio of $rptr$ in BxCRS is determined by γ. Furthermore, if the non-zero values of the array are uniformly distributed, then $\gamma = 1-\sigma$, since $\mathcal{N}_o = (1-\sigma)\mathcal{N}_{k-1}$. Thus we can conclude that the overall space saving in BxCRS is approximately $(1-\gamma)\mathcal{S}_{rptr}$. Using BxCRS, the storage requirement of xCRS can be reduced considerably only when the value of γ is sufficiently small, which means the sparsity σ of the MDSA is very high.

3.3.3 Random Element Access and Sub-Array Retrieval in BxCRS

The processes of a random array element access and sub-array retrieval in BxCRS and in xCRS have many similarities. This is due to the fact that the two representations of an MDSA differ only in the arrays, $rptr$ in xCRS and $rptr_c$, $compwrd$ in BxCRS, that store the starting positions of the rows in the 2-dimensional array space. Given a certain row offset value, row_offset, in xCRS, the value $rptr[row_offset]$ could tell us whether the row is unoccupied or occupied. If it is occupied the starting position of the row in $cind$ is the value of $rptr[row_offset]$. In BxCRS, on the other hand, the array $rptr_c$ stores only the starting positions of those occupied rows. We need to visit the array $compwrd$ to determine whether a particular row is occupied or not (Algorithm 13 in Appendix A shows such a procedure). If it is occupied, we go further to find out the position

in $rptr_c$ that tell us the starting position of the row in $cind$.

Given a row_offset value, we visit the corresponding *bit* in the compressed word, $compwrd[\lfloor \frac{row_offset}{W-1} \rfloor]$. The bit value tells us if the row is occupied (1) or not (0). If the row is occupied, we need to count all the previous 1's, i.e., the occupied rows, in the compressed words up to the current word in $compwrd$. The count of these 1's is the position in $rptr_c$ that stores the starting position of the particular row in $cind$. This process is spelled out in Algorithm 4. In our implementation of random element access and sub-array retrieval, we built a table or array, denoted as count1Table, of the incremental counts of 1's for each word in $compwrd$. For example, the entry count1Table[12] stores all the 1's in the leading 12 compressed words in $compwrd$. Using this method, the time complexity of counting 1's is reduced to $O(1)$, from the linear time complexity of counting the 1's directly in $compwrd$. Hence, this table could remarkably speed up the counting process mentioned above, and it is only built at runtime.

Algorithm 4: bxcrsSearch($val, cind, rptr_c, compwrd, n$)

Input: BxCRS arrays and the index $n[0..k-1]$ of an array element
Output: Returns the corresponding value v if the array element exists, or returns a $FALSE$ value.
begin
  $FALSE \leftarrow -1$
  $r \leftarrow$ getRowOffset(n)
  `/*` W `is the bit length of a word in` $compwrd$ `*/`
  $r \leftarrow r/(W-1)$
  $t \leftarrow r\%(W-1)$
  Get the bit, b, at position $W-1-t$ in $compwrd[r]$
  if $b \neq 0$ **then**
    Find the number q of bit 1's, in $compwrd[0..r-1]$, and up to position $W-1-t$ in $compwrd[r]$
    $p1 \leftarrow rptr_c[q]$
    $p2 \leftarrow rptr_c[q+1]$
    $v \leftarrow$ binarySearch($cind, val, p1, p2-1, n[0]$)
  else
    $v \leftarrow FALSE$
  return v

A random array element access and sub-array retrieval in BxCRS has the same time complexities as in xCRS (see Section 3.2.2). However, as the comparison in the previous paragraphs showed, the computational cost of accessing a row in BxCRS is slightly more expensive than in xCRS. More precisely, the 'extra' computing costs are mainly from the computation of the location of a row, i.e., $\lfloor \frac{row_offset}{W-1} \rfloor$, in the array $compwrd$, and finding the number of 1's in the last word in $compwrd$.

3.4 Hybrid Approach

Compressing the array $rptr$ in BxCRS using WAH achieves very good compression ratio only when the MDSA is very sparse, say $\sigma > 95\%$. Another way of reducing the size of the array $rptr$ is to store the starting positions of 'blocks' of more than one dimension instead of one-dimensional *rows* as in xCRS. The Hybrid Approach, *Hybrid*, explores combining xCRS with BESS to achieve a good balance between storage and data access efficiencies. The Hybrid method allows mapping a k-dimensional array to any r-dimensional array ($r \leq k$), instead of a 2-dimensional array only as in xCRS. The mapping transforms the k-dimensional array into a number of $(r-1)$-dimensional ($1 \leq r \leq k$) arrays that are each linearized. We use BESS to represent the non-zero elements in each $(r-1)$-dimensional array, and xCRS to represent the resulting r-dimensional array.

3.4.1 Hybrid and Its Construction

Similar to the definition of row_offset in xCRS, we define $block_offset$ in Hybrid as the displacement of an $(r-1)$-dimensional array in a linear address space. The $block_offset$ is defined using the leading $k-r+1$ index values, i.e., $\langle n_{k-1}, \ldots, n_{r-1} \rangle$, and computed as $\sum_{i=r-1}^{k-1} n_i \prod_{j=r-1}^{i-1} D_j$. Within each $(r-1)$-dimensional array, following the idea of BESS, we construct the bit encoded index, BEI, for each non-zero element. We find the BEI for a non-zero element using its lower $r-1$ index values $\langle n_{r-2}, \ldots, n_0 \rangle$. In constructing the BEIs, we use $\lceil \log D_i \rceil (0 \leq i \leq r-2)$ bits for each dimension d_i. The bits are concatenated in the order of $d_{r-2}, \ldots, d_0$, and interpreted as an integer. Applying this dimensional mapping to a k-dimensional array $A[D_{k-1}] \ldots [D_0]$, we are able to construct a unique key for each non-zero element in A, and the key is defined as $(block_offset, BEI)$.

The Hybrid approach organizes the key-value pairs as follows. The non-zero values and their corresponding BEIs are stored in two 1-dimensional arrays (or vectors), val and bei, respectively. We use the third array, $bptr$, to store the starting positions of each $(r-1)$-dimensional array in bei. If an $(r-1)$-dimensional array has no valid array elements, we store "-1" for that particular array. The entry for an $(r-1)$-dimensional array in $bptr$ is obtained using its $block_offset$, which is simply $bptr[block_offset]$. The Hybrid representation of the example MDSA in Figure 2.1 is shown in Table 3.3. In this example, we chose $r = 3$. The array $bptr$ in Table 3.3 stores the starting positions of each 2-dimensional array. For example all the non-zero elements with index value $n_2 = 0$, are stored within the range which starts at the position $bptr[0] = 0$, and ends at the starting position of next occupied block (which is 4) in the array val.

The dimensional mapping in Hybrid allows r to be any value between 1 and k. To show how to choose r with

offset	0	1	2	3	4	5	6	...	10	11	12	13
val	20.5	11.2	17.0	23.6	14.9	15.2	17.8	...	14.2	45.6	75.3	13
bei	0000	0001	1010	1011	0011	0100	0101	...	0001	0110	1011	13
bptr	0	−1	4	−1	9	13						

Table 3.3: The Hybrid representation of the MDSA in Figure 2.1.

respect to k, we define a parameter *hybrid ratio* α, where $r = \alpha k$, $\frac{1}{k} \leq \alpha \leq 1$. When $\alpha = \frac{1}{k}$, r becomes 1, the Hybrid representation results in the original MDSA. If $\alpha = \frac{2}{k}$, then $r = 2$, the Hybrid becomes xCRS. When α increases from $\frac{1}{k}$ to 1, the method gradually transforms into BESS. In another words, the data access efficiency is gradually traded off by storage space efficiency.

3.4.2 Random Element Access and Sub-Array Retrieval in Hybrid

Given the Hybrid representation of an array A, we access an array element, $\langle n_{k-1}, \ldots, n_0 \rangle$, using the following procedure.

Step 1: Compute the $block_offset$ of the sub-array where the array element belongs to.

Step 2: Examine the value $bptr[block_offset]$. If it is "-1", the array element has no valid data. Otherwise, we find the starting position of the corresponding sub-array.

Step 3: Determine the starting position of the next occupied sub-array by examining the value $bptr[block_offset + 1]$. We keep on increasing the value $block_offset$ until we found the next occupied sub-array.

Step 4: Construct the bit encoded index, BEI, of the array element.

Step 5: Search for the value BEI within the range determined by Steps 2 and 3.

Sub-array retrieval is efficient in Hybrid when one or more $(r-1)$-dimensional sub-arrays, which are represented in BESS, entirely belong to the sub-array being retrieved. This is true because the range of an $(r-1)$-dimensional array in bei or val is easily determined as shown in the procedure for random element access. In other cases, we need to retrieve the sub-array elements in multiple $(r-1)$ dimensional sub-arrays. A sequential scan or binary search then needs to be used to find those desired elements within each $(r-1)$-dimensional sub-array.

3.4.3 Some Properties of Hybrid

3.4.3.1 The Storage Overhead

The storage space usage of Hybrid approach, $\mathcal{S}_{hybrid}$ is

$$\mathcal{S}_{hybrid} = \mathcal{S}_{val} + \mathcal{S}_{bei} + \mathcal{S}_{bptr} = (C_f + C_i)\mathcal{N}_{nz} + C_{il} \prod_{i=r-1}^{k-1} D_i \tag{3.4}$$

where $\mathcal{S}_{val}$, $\mathcal{S}_{bei}$ and $\mathcal{S}_{bptr}$ represent the sizes of the Hybrid arrays val, bei and $bptr$ respectively. For a certain r, we can minimize the storage usage by reordering the dimensions so that $\prod_{i=r-1}^{k-1} D_i$ is minimum for a k-dimensional sparse array.

To compare the storage utilization of Hybrid with that of xCRS, consider the following ratio.

$$\begin{aligned}\frac{\mathcal{S}_{bptr}}{\mathcal{S}_{rptr}} &= \frac{C_{il} \prod_{i=r-1}^{k-1} D_i}{C_{il}\mathcal{N}_{k-1}} \\ &= \frac{\prod_{i=r-1}^{k-1} D_i}{\prod_{i=1}^{k-1} D_i} \\ &= \frac{1}{\prod_{i=1}^{r-2} D_i}\end{aligned}$$

When r approaches the value k, the ratio above decreases rapidly. This means that the storage overhead in Hybrid can be much less than the one in xCRS. On the other hand, when r approaches 2, the sizes of those $(r-1)$-dimensional arrays become smaller, and accessing the array elements within these 'blocks' becomes more efficient. At $r = 2$, the ratio is 1, and the Hybrid representation becomes equivalent to the xCRS. Hence, we conclude that a desirable balance between storage utilization and data access efficiency can be achieved by choosing some proper values for r. Similarly, if we compare the storage usage of Hybrid with that of BESS, the additional space requirement incurred in Hybrid is $C_{il} \prod_{i=r-2}^{k-1} D_i$. This value can be a very small fraction of $\mathcal{N}_{k-1}$ by reordering the dimensions.

3.4.3.2 The Time Complexities of Random Element Access and Sub-Array Retrieval in Hybrid

The time complexity of finding the starting position of an $(r-1)$-dimensional array in bei is $O(k-r)$. This is due to the costs of computing the $block_offset$ and BEI of an array element from its index $\langle n_{k-1}, \ldots, n_0 \rangle$. Recall that $block_offset = \sum_{i=r-1}^{k-1} n_i \prod_{j=r-1}^{i-1} D_j$. There are $k-r+1$ additions and $k-r$ multiplications (assuming $\prod_{j=r-1}^{t} D_j$, $r-1 \leq t \leq k-1$ is precomputed). The cost of computing the BEI involves $r-2$ *binary bit shift* and *binary bit or* operations, which are negligible compared with the cost of computing the $block_offset$. Similarly, the cost of the inverse operations, finding the array index of an element from its

$block_offset$ and BEI, is dominated by the cost of $k - r$ division operations to recover the leading $k - r$ index values ($\langle n_{k-1}, \ldots, n_{r-1} \rangle$).

Within each $(r-1)$-dimensional array, the random element access time is bounded by $O(\log \mathcal{N}_b)$, where $\mathcal{N}_b$ is the number of non-zero elements in a $(r-1)$-dimensional sub-array. $\mathcal{N}_b$ is usually much less than the $(r-1)$-dimensional space $\prod_{i=0}^{r-2} D_i$. The difference between $\mathcal{N}_{nz}$ and $\mathcal{N}_b$ depends on the value of r. When r increases from 1 to k, $\mathcal{N}_b$ increases gradually from the number of non-zero elements in a xCRS row to $\mathcal{N}_{nz}$. Similarly, in sub-array retrieval, the time complexity, to retrieve multiple elements within an $(r-1)$-dimensional sub-array, also depends on $\mathcal{N}_b$.

3.5 PATRICIA Trie Compressed Storage

PATRICIA trie compressed storage (PTCS) represents a multi-dimensional sparse array by constructing a PATRICIA trie for the non-zero elements only in the MDSA. Each node in the PATRICIA trie stores a *key-value* pair. This results in the storage space requirement for representing the MDSA is linear in the number of the non-zero elements. A unique characteristic of PTCS, compared with the array-based storage schemes, is its flexibility for *update* operations such as insertion and deletion. While an *update* operation on the data structures for sparse array, often requires rearranging the whole data, it is only a matter of single node insertion or deletion for a trie data structure.

3.5.1 PATRICIA

PATRICIA stands for Practical Algorithm to Retrieve Information Coded In Alphanumeric. It is a system for constructing an index for a binary coded library, and was introduced by Donald R. Morrison in 1968 [29]. The basic idea of PATRICIA is to avoid one way branching in binary trie by including in each node the number of bits to skip over before making the next test. A very important property of PATRICIA trie is that there are no void (NULL) external nodes, which means that if there are N external nodes, then there are $N - 1$ internal nodes. This property is exploited by 'folding' the external nodes into internal nodes plus one header node. Thus each internal node in a PATRICIA trie becomes an external or a *leaf* node as well. This method of representing the PATRICIA trie was first described by Knuth [25]. We adopt this approach for our sparse array representation. A PATRICIA trie constructed in this manner is shown in Figure 3.2. In this example, the decimal number in each node indicates the bit position (or bit-index) to be tested for branching, and the binary bit sequences are the *keys*. The key construction method is given later in the following section.

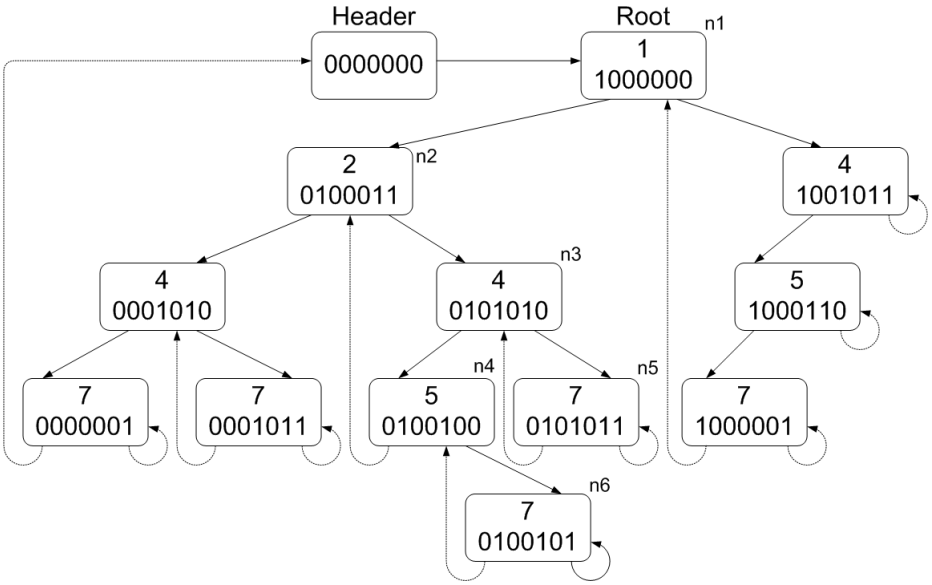


Figure 3.2: The Patricia trie representing the MDSA in Figure 2.1. (The bit index starts from 1 and increase from left to right)

3.5.2 PTCS and Its Key

PTCS maps k-dimensional sparse array space to one-dimensional array, or linear space, similar to the mapping in offset-value pair and BESS. In this work we construct the binary keys in the same manner as BESS. We implement this *dimensional mapping* by creating a unique *key* in linear space for each non-zero element. Each key-value pair is then inserted into a PATRICIA trie as a single node.

PTCS keys are constructed by encoding the array index of each non-zero element into a single binary bit sequence. The bit sequence is then interpreted as an integer using the minimum number of bits in a computer word. We use $\lceil \log_2 D_i \rceil$ $(0 \leq i \leq k-1)$ bits to encode the index value of dimension d_i. Given a k-dimensional index $\langle n_{k-1}, \ldots, n_1, n_0 \rangle$, let β_j be the $\lceil \log_2 D_j \rceil$ bit representation of n_j. The bit values of the k-dimensional index are concatenated in order such that $\beta = \beta_{k-1} || \ldots || \beta_1 || \beta_0$ and then stored in the low-order bits of an integer word. Both PTCS and BESS keys are generated in the same manner. For example, a possible key structure for a 4-dimensional sparse array $A[12][12][12][1200]$ is shown in Figure 3.3, assuming the length of an integer is 32. In Figure 3.3, the trailing 11-bits in positions, 10 to 0, store the bits of dimension d_0. The sequence of 4-bits in positions 14 down to 11 store the 4 bits of dimension d_1. The 4-bits of each of the remaining dimensions d_2 and d_3 are similarly concatenated in the key. The remaining leading bit positions are set to 0 signifying that they are not used. Following this simple key construction method, the keys and their corresponding values for the example 3-dimensional array in Figure 2.1 are shown in Table 3.4. In this example, we used 3, 2, 2 bits respectively for the dimensions d_2, d_1, d_0 (or I, J, K in the example), and the

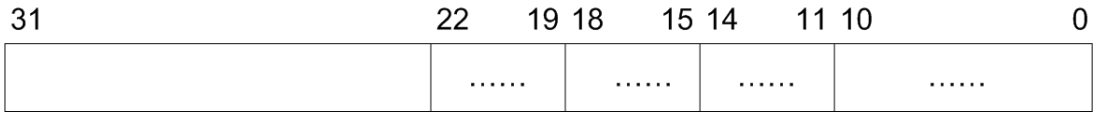


Figure 3.3: A PTCS key structure for a 4-dimensional sparse array.

size of an integer is assumed to be 8 bits only. The bit sequences are concatenated in the order of I-J-K. Only the trailing 7 bits of an integer are used in this case. Note also that traversing the *leaf* nodes from left to right, when the PTCS keys are constructed as described, gives the *row-major* traversal of the non-zero elements of the sparse array.

val	$index$	β	$ptcs_key$
20.5	$(0,0,0)$	00000000	0
11.2	$(0,0,1)$	00000001	1
17.0	$(0,2,2)$	00001010	10
23.6	$(0,2,3)$	00001011	11
14.9	$(2,0,3)$	00100011	35
15.2	$(2,1,0)$	00100100	36
17.8	$(2,1,1)$	00100101	37
21.3	$(2,2,2)$	00101010	42
		.....	

Table 3.4: The PTCS key-value pairs for the MDSA in Figure 2.1

3.5.3 PTCS Construction

To insert a key-value pair in a given PATRICIA trie, we examine first if the key is already in the trie or not. If it does not exist, a new node will be created and inserted into the trie. The PATRICIA trie node structure used in our implementation for PTCS is specified using an abstract *C struct* as follows.

```
struct PTCSnode
{
    int bit_pos
    long int key
    float val
    PTCSnode* llink
    PTCSnode* rlink
}
```

The PATRICIA trie node structure given above has five data fields. The fields key and val store the PTCS key and value respectively. The field bit_pos indicates the bit position in a key to be examined for branching purpose. In PTCS, the value of bit_pos starts from 1 at the most significant bit in a key and increases towards the least significant bit. The fields $llink$ and $rlink$ are the pointers to the left and right child nodes respectively.

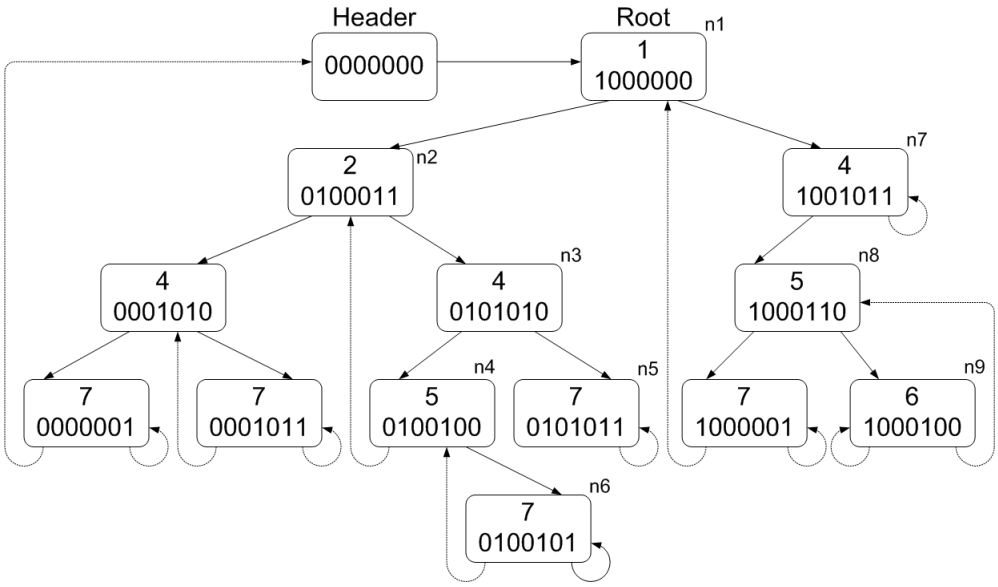


Figure 3.4: The insertion of a key $x = 1000100$ into the PATRICIA trie of Figure 3.2.

Given a PTCS key-value pair (x, v), we traverse down the trie by examining a number of bits (see next section for more details) in the key x. Once a *leaf* node is reached, a key comparison is done to see if the key in the *leaf* node is the same as the key to be inserted. If they do not match, we find the first bit position i that differs in the two keys. We then traverse the trie again from the root node until we reach a *leaf* node or a node where the value of the bit_pos field is not less than the value i. Let us denote this particular node as tmp. At this position, we create a new node with the key-value pair, and set the value of the bit_pos as i. For the new node, we need to adjust its parent and child links. In the case of the parent link, we inspect the bit value in x at the position which indicated by the bit_pos field in the parent node of tmp. If it is 1, the right link of the parent node of tmp will point to the new node, or the left link of the parent node of tmp will point to the new node if the bit value is 0. To adjust the child links, we examine the bit value in x at position i. If it is 1, the left link of the new node points to the node tmp, the right link points to the new node itself, and vice versa if it is 0. For example, the insertion of a key $x = 1000100$ into the PATRICIA trie of Figure 3.2 is illustrated in Figure 3.4. The node $n8$ in Figure 3.4 is equivalent to the node tmp, and the node $n9$ is the new node with the key value 1000100. The Algorithm 5 gives the insertion process for creating the PTCS storage scheme.

Algorithm 5: ptcsInsert(T, key, val)

Input: A PATRICIA trie T, a key (key) and value (val) pair.

Output: The PATRICIA trie T with the key-value pair inserted.

begin

if $T = NULL$ **then** Empty trie

Insert a header node

return T

else

$ptr1 \leftarrow T, tmp \leftarrow pt.rlink$

`/* Traverse the trie until a leaf node is reached */`

while $ptr1.bit_pos < tmp.bit_pos$ **do**

$ptr1 \leftarrow tmp$

if *getBit*$(key, tmp.bit_pos) = 1$ **then**

$tmp \leftarrow tmp.rlink$

else

$tmp \leftarrow tmp.llink$

if $key = tmp.key$ **then** The key is already in the trie

return T

else

$i = firstDifferentBit(key, tmp.key)$

$ptr1 \leftarrow T, ptr2 \leftarrow pt.rlink$

`/* Look for the right position for the new node between the root node and the node` tmp `*/`

while $ptr1.bit_pos < ptr2.bit_pos$ *and* $ptr2.bit_pos < i$ **do**

$ptr1 \leftarrow ptr2$

if *getBit*$(key, ptr2.bit_pos) = 1$ **then**

$ptr2 \leftarrow ptr2.rlink$

else

$ptr2 \leftarrow ptr2.llink$

$left = getBit(key, i)?ptr2 : NULL$

$right = getBit(key, i)?NULL : ptr2$

$ptr3 = createNewNode(i, key, val, left, right)$

if $ptr3.rlink = NULL$ **then**

$ptr3.rlink = ptr3$

else

$ptr3.llink = ptr3$

`/* Adjust the left or right pointer of node` $ptr2$ `*/`

if $getBit(key, ptr1.bit_pos)$ **then**

$ptr1.rlink = ptr3$

else

$ptr1.llink = ptr3$

return T

3.5.4 Random Element Access and Sub-Array Retrieval in PTCS

Random element access: Given a PTCS key x, we traverse the PATRICIA trie until a *leaf* node is reached. At the *leaf* node, we compare the stored key with x. If the keys match, we retrieve the value of the node. Note that only a single key comparison is done in the search process. We describe the random element access process in more detail in the following example. Suppose we search for, or try to access, a key x with bit sequence '0101010' in the PATRICIA trie of Figure 3.2. We start at the root node ($n1$) by checking its bit_pos field, which indicates that we examine the bit at position 1 (the leftmost bit) in the key x. That bit is 0, so we go to the left branch. At node $n2$, we check the bit in x at position 2, which is 1, that leads us to the right, node $n3$. From node $n3$, we move down to $n5$. At node $n5$, the bit in position 7 examined in x is 0. The left link of $n5$ takes us back to $n3$ again. However, the value of bit_pos in node $n3$ is less than that of node $n5$. This indicates a *leaf* node is reached. We find a desired *leaf* node by traversing down the PATRICIA trie until we come to a node, where the values of the bit_pos field do not increase any more. The random element access process is shown in Algorithm 6.

Algorithm 6: ptcsSearch(T, key)

Input: The PATRICIA trie T and a key, key.
Output: Return the data val if the key is found or NULL if not found.
begin
  $pt \leftarrow T$, $tmp \leftarrow pt.rlink$
  while $pt.bit_pos < tmp.bit_pos$ **do**
    $pt \leftarrow tmp$
    if *GetBit*$(key, tmp.bit_pos) = 1$ **then**
      $tmp \leftarrow tmp.rlink$
    else
      $tmp \leftarrow tmp.llink$
  if $key = tmp.key$ **then**
    return $tmp.val$
  else
    return NULL

Sub-array retrieval: In the design of sub-array retrieval algorithm, the following property of PATRICIA trie is exploited. **Pre-order traversal of Patricia trie in PTCS results in a sorted list of the array elements in row-major order.** Following this property, the elements in a sub-array resides only in one of the sub-tries. For example, all the array elements that have value 2 on dimension d_2 resides in the right sub-trie rooted at node $n2$ in Figure 3.2. Algorithm 14 in Appendix A is designed using the same property for sub-array retrieval in PTCS. The size of the sub-trie is dependent on the size of the sub-array defined. Furthermore, it is possible to traverse only part of this sub-trie to retrieve the desired sub-array elements.

3.5.5 Some Properties of PTCS

Note that the PATRICIA trie in PTCS has only data nodes, each node represents one data item, there is no node in the trie used only for a branching purpose. If an MDSA contains $\mathcal{N}_{nz}$ valid array elements, a PATRICIA trie with $\mathcal{N}_{nz}$ nodes will suffice to represent the entire non-zero values. Thus the storage used to represent a multi-dimensional sparse array is linear with respect to the number of non-zero values. The PATRICIA trie node structure is the same as the one given in Section 3.5.3 on Page 43, and the size of one node is denoted by $\mathcal{S}_{pnode}$. The exact amount of storage space required for PTCS, $\mathcal{S}_{ptcs}$ is computed as follows.

$$\mathcal{S}_{ptcs} = \mathcal{N}_{nz}\mathcal{S}_{pnode} \approx (C_i + 3C_{il} + C_f)\mathcal{N}_{nz} \tag{3.5}$$

In the above formula, we assumed the size of a pointer data type to be the same as the size of the longest integer. The basic operations, such as random element access in PTCS take time proportional to the height of the trie. If an MDSA, represented in PTCS, contains $\mathcal{N}_{nz}$ non-zero values, then the average height h of the trie is at least $\lceil \log_2 \mathcal{N}_{nz} \rceil$ [9]. The average number of bits examined during a successful random search in a PATRICIA trie is approximately $\log \mathcal{N}_{nz} + 0.33275$, and $\log \mathcal{N}_{nz} - 0.31875$ during an unsuccessful one [26]. Based on these results, we may expect that a random element access in PTCS has the time complexity of $O(\log \mathcal{N}_{nz})$. The time complexity of sub-array retrieval in PTCS is determined by the size, $\mathcal{N}_b$, of the sub-trie where the sub-array resides. Once the root node of the sub-trie is found, the traversal of the nodes in that sub-trie has the time complexity of $O(\mathcal{N}_b)$. The computational costs in sub-array retrieval include key comparisons and decomposing the PTCS key into array index.

A drawback of PTCS lies in its key construction. Recall that a PTCS key is constructed by encoding each index value on the dimension d_i using $\lceil \log D_i \rceil$ bits. When D_i is much less than $2^{\lceil \log D_i \rceil}$, this method could lead to the imbalanced distribution of bits 0 and 1 in the PTCS keys. For example, if $D = 5$, we use 3 bits to encode the index values. They are $0 \rightarrow 000$, $1 \rightarrow 001$, $2 \rightarrow 010$, $3 \rightarrow 011$, and $4 \rightarrow 100$. If we assume these index values appear randomly (or uniformly), then the occurrences of 0's are much more than that of 1's in the keys. A PATRICIA trie is insensitive to the order in which the keys are inserted, but it is very sensitive to the distribution of the bits in the keys [26]. If bit 0 appears far more often than the bit 1, or vice versa, the resulting PATRICIA trie is likely to become more imbalanced.

In this chapter, we have introduced four new storage schemes for organizing multi-dimensional sparse arrays. These methods share the same approach in handling sparsity of the array by storing only the valid array elements, and applying various *dimensional mappings*. Our main concern in the design of storage schemes for MDSAs is to reduce the storage overhead without compromising the data access efficiency. Analytical properties of each of these methods have been discussed to show their advantages and disadvantages. Experimental results and further analyses are given in Chapter 7.

Chapter 4

Multi-Dimensional Aggregations of Sparse Array Elements

Data warehousing and OLAP systems frequently use aggregations to answer complex queries. The data in these systems is characterized by massive volume and high dimensionality. In such a context, carrying out aggregation needs to be very efficient with respect to space and time. This is especially the case in computing the *cube*. A multi-dimensional data in MOLAP consists of *measure data* and *dimensional data*. It is typical that one measure data is associated with a set of dimensional data. Since the dimensional data is often represented as integers, it takes up the most volume of the data, especially in the case of high dimensions, in data warehouses. Reducing the storage of dimensional data could lead to reduced volume of the data. Each storage scheme we introduced in Chapter 3 reduces the storage for the dimensional data by a certain degree, while leaving the measure data as the original. In the design of these schemes, besides improving the storage utilization, we also take into consideration maintaining efficient data access. Accessing an array element, in any of these storage schemes, can be done independently and efficiently. In multi-dimensional aggregation and the *cube* computation, applying a storage scheme to represent the multi-dimensional data could lead to the following benefits. Firstly, more data can fit in the memory, thereby reducing the I/O. Secondly, accessing an array element is efficient, hence increasing the speed. Thirdly, the aggregation results may also be represented in a space efficient storage scheme.

To evaluate the performances of various storage schemes on the computation of multi-dimensional aggregation, we designed and implemented aggregation algorithms for PTCS, xCRS, BxCRS, Hybrid and BESS. We took the following approaches in the design of these algorithms.

1. We considered only one distributive aggregate function on *summation*.

2. For a k-dimensional sparse array, we assumed the dataset was organized and sorted in the dimension order of $d_{k-1}, \ldots, d_1, d_0$. The algorithms aggregate on $k-1$ group-bys in a single scan of the data. The $k-1$ group-bys are (d_{k-1}), (d_{k-1}, d_{k-2}), $\ldots$, $(d_{k-1}, \ldots, d_2)$, $(d_{k-1}, \ldots, d_2, d_1)$ respectively. We

left out computing the group-by of $(d_{k-1}, \ldots, d_1, d_0)$, since we assumed there were no duplicate array elements in the MDSA data.

3. A top-down approach was applied to compute the multiple aggregations.

4. The intermediate aggregation results were stored as a list of key-value pairs in memory, where the key was a BEI (Bit Encoded Index) representing the dimensional data. The final results were written into a file using index-value pair representation.

5. For simplicity, we only considered the case where the entire data set can fit in memory.

In the following sections, we discuss the methods to compute the multi-dimensional aggregation using various storage schemes. We also discuss our approach to compute the *cube* operator using BESS and PTCS.

4.1 Aggregation Using PTCS and BESS

Pre-order traversal of PATRICIA trie in PTCS visits the nodes in increasing order of their *keys*. Such an ordered list of the elements is exactly the same as the list of elements in BESS. Let us assume the keys, which encode the dimensional data, are sorted on the dimension order of $(d_{k-1}, \ldots, d_2, d_1, d_0)$. It implies the array elements are grouped on the same dimension order. This fact enables us to compute multiple aggregations in one traversal of the PATRICIA trie in PTCS, or one serial scan of the BESS arrays. The benefits of this approach include reduced I/O, sharing the cost of sorting, and using the result of one aggregation to compute another. In a top-down approach, we start aggregating at the finest level granularities, which is the group-by of $(d_{k-1}, \ldots, d_1)$ in our case. In order to determine if the two neighburing array elements are in the same group on dimensions $(d_{k-1}, \ldots, d_i)$, we only need to mask off the bits on dimensions $(d_{i-1}, \ldots, d_0)$, then compare the resulting new keys. If they are the same, the two array elements are in the same group. We can conclude even further that they will be in the same groups on the remaining group-bys of $(d_{k-1}, \ldots, d_{i+1}), \ldots, (d_{k-1}, d_{k-2}), (d_{k-1})$. If the new keys are not the same, we aggregate the data into different groups.

To compute the aggregation for each array element (or node in the case of PTCS), we keep note of two elements at the same time. They are the *current* element and the *previous* element, which were aggregated immediately before the *current* one. We first mask off the bits for dimension d_0 in both keys of the two elements. The resulting two new key values are then compared to determine whether the current data value should be aggregated to an older value or a new *key-value* pair should be created. In the latter case, the *key* is the new key of the *current* element with its bits on dimension d_0 masked off. The same operation is then carried out iteratively to the next level of aggregation, where the bits on the next dimension are masked off in the keys. For example, if

we masked off the dimension d_0 in the previous iteration, then we mask off the d_1 in the current iteration and so on. The iteration stops when either we exhaust all the dimensions or the array elements are already in the same group. That is to say the keys of the *current* element and *previous* element are the same. This procedure is depicted for the 3-dimensional example data for Figure 2.1 on Page 11, in Figure 4.1. Note that in order to compute $k-1$ aggregations in one scan, we need to create $k-1$ arrays to store the results. To save the memory usage at runtime, we store the results as a list of key (PTCS key or BESS key) and value pairs.

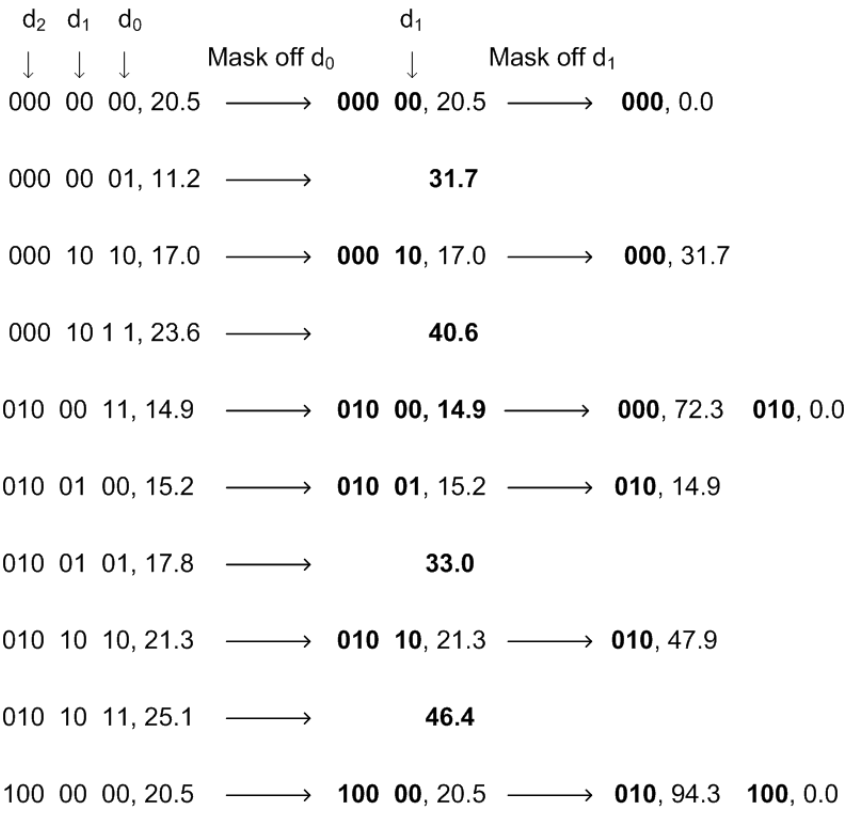


Figure 4.1: Aggregating the data represented in PTCS or BESS. Note that we only considered those dimensional bits in an integer here. The leading unused bits in PTCS and BESS keys are not shown. The keys and values in bold mark those pairs which are part of the results. The operation 'Mask off d_1' is not needed to be carried out where there is no arrow between 2nd and 3rd column.

4.2 Aggregation Using xCRS and BxCRS

XCRS and BxCRS use the similar algorithms for the basic array operations we discussed in Chapter 3. This is no exception for multi-dimensional aggregation as well. The difference between the two schemes lies in the computation of the starting position, in the array $cind$, of a certain row (see Section 3.3.3). In this section, we discuss computing multi-dimensional aggregation using xCRS. The methods we developed in this regard also applies to BxCRS.

Our approach to compute multiple aggregations in xCRS is to aggregate the array elements *row* by *row*. Within each row, we compute multiple aggregations using the top-down approach. A row in xCRS has the finest level of granularity, i.e., $(d_{k-1}, \ldots, d_1)$, among the aggregations we compute. The aggregation results of the rows can be used to compute the other aggregations with coarser level of granularities. Visiting the array elements row by row can be done by conducting a serial scan of the array $rptr$ in xCRS. Recall that, in xCRS, an array element is represented using index (row_offset, n_0), where row_offset is computed using the array index as $\sum_{i=1}^{k-1} n_i \prod_{j=1}^{i-1} D_j$, and n_0 is the dimension value on d_0. Once the range of a row in the array val (or $cind$) is found, we aggregate all the values in val within this range to a single value. To compute the aggregations on other levels, we need to recover the index values, $\langle n_{k-1}, \ldots, n_1 \rangle$, from the value of row_offset. However, computing the index values from the row_offset requires a number of modulus and subtraction operations, which are computationally expensive, especially the modulus operation. One way to avoid such a computation is to increment the index values according to the difference in $row_offsets$. For example, the index values corresponds to row_offset value 0 is $\langle 0, \ldots, 0, 0 \rangle$ (in C language). If the next occupied row has the row_offset value 4, the corresponding index values can be computed by incrementing the index value $\langle 0, \ldots, 0, 0 \rangle$ by 4, which only needs a small number of additions. The $k-1$ leading index values, i.e., $\langle n_{k-1}, \ldots, n_1 \rangle$, are then stored as BEI (bit encoded index) instead of using $k-1$ integers. Using BEI not only saves space, but also simplifies the index comparisons. Comparing two indexes in its conventional array index form usually needs more than one integer value comparisons. On the other hand, comparing two indexes represented in their BEIs needs only one comparison.

From the aggregation result on a row, and the BEI for $\langle n_{k-1}, \ldots, n_1 \rangle$, we can compute the remaining $k-2$ aggregations iteratively in the similar way as discussed in Section 4.1. During the process of aggregation, we always aggregate the current value according to the previous one aggregated immediately before it using their BEIs. At each level, we mask off, in the BEIs from the previous level, the bits on a certain dimension. We then compare the two new BEIs to decide whether the current value is aggregated to an older group or a new one. The aggregation results are stored as key-value pairs, where the key is the BEI of the corresponding dimension values on each level of aggregation.

4.3 Aggregation Using Hybrid

Hybrid represents MDSAs using the combination of xCRS and BESS. Consequently, computing the multi-dimensional aggregations for Hybrid takes the characteristics of both schemes into consideration. Given a k-dimensional sparse array, Hybrid organizes the MDSA as a number of $(r-1)$-dimensional $(1 \leq r \leq k)$ arrays in a linear address space. We address an $(r-1)$-dimensional array in Hybrid using its *block_offset* in

the linear address space. The array elements in each $(r-1)$-dimensional array are represented using BESS, and these $(r-1)$-dimensional arrays are represented using xCRS. Each array element in Hybrid is represented using an index $(block_offset, BEI)$. To compute multiple aggregations in Hybrid, we aggregate within an $(r-1)$-dimensional array using the method for BESS, then using these results to compute other less detailed aggregations using the same method for xCRS.

We compute multiple aggregations 'block' by 'block' in Hybrid, instead of aggregating row by row in xCRS. A 'block' is an $(r-1)$-dimensional array in Hybrid here. These $(r-1)$-dimensional arrays are visited in the increasing order of their $block_offsets$, i.e., $0, 1, 2, 3, \ldots, \prod_{i=r-1}^{k-1} D_i - 1$. An outline of computing multiple aggregations for each $(r-1)$-dimensional array or 'block' in Hybrid is described below.

1. Compute the array index (or block index), $\langle n_{k-1}, \ldots, n_{r-1} \rangle$, of the $(r-1)$-dimensional array. This can be done by incrementing the block index, of previously visited $(r-1)$-dimensional array, by the difference in their $block_offsets$. We then construct the bit encoded index, BEI for the block index.
2. Find the range of the $(r-1)$-dimensional array in bei by visiting array $bptr$.
3. Compute the aggregations on $r-1$ number of group-bys, $(d_{k-1}, \ldots, d_1)$, $\ldots$, $(d_{k-1}, \ldots, d_{r-1})$ using the aggregation method for BESS.
4. Using the aggregation result on $(d_{k-1}, \ldots, d_{r-1})$, compute the aggregations on the remaining $k-r$ number of group-bys, $(d_{k-1}, \ldots, d_r)$, $\ldots$, (d_{k-1}).

The aggregation results are stored as key-value pairs, where the key is the BEI of an aggregated group.

4.4 Comparative Analysis of Computing Aggregations Using Various Schemes

Table 4.1 summarizes the approximate computational costs (in the worst case scenario) of computing $k-1$ aggregations in a single scan of the data represented in PTCS, BESS, xCRS, BxCRS and Hybrid respectively. The nature of $k-1$ aggregations is given at the beginning of this chapter. In Table 4.1 C_{bit_ops} is the cost of bit operation; C_{x_row}, C_{bx_row} are the costs of finding the range of a row and computing the row index in xCRS and BxCRS respectively; C_{hy_blk} is the cost of finding the range of a $(r-1)$-dimensional array and computing the 'block' index in Hybrid; C_{pre_ord} is the cost of pre-order trie traversal in PTCS. C_{pre_ord} is specific to PTCS, because all the other storage schemes are array based. Finally, $C(f)$ is the total cost of carrying out the aggregation function $f()$ multiple times. $C(f)$ is the same for all the schemes considered here. Computing the 'block' index in Hybrid involves fewer number of dimensions than it is in computing the row index in xCRS

Scheme	Cost
PTCS	$(k-1)C_{bit_ops}\mathcal{N}_{nz} + C(f) + C_{pre_ord}$
BESS	$(k-1)C_{bit_ops}\mathcal{N}_{nz} + C(f)$
xCRS	$C_{x_row}\prod_{i=1}^{k-1} D_i + (k-2)C_{bit_ops}\prod_{i=1}^{k-1} D_i + C(f)$
BxCRS	$C_{bx_row}\prod_{i=1}^{k-1} D_i + (k-2)C_{bit_ops}\prod_{i=1}^{k-1} D_i + C(f)$
Hybrid	$C_{hy_blk}\prod_{i=r-1}^{k-1} D_i + (k-r)C_{bit_ops}\prod_{i=r-1}^{k-1} D_i + (r-2)C_{bit_ops}\mathcal{N}_{nz} + C(f)$

Table 4.1: The approximated costs of computing multiple aggregations using various storage schemes

or BxCRS. Further, finding the range of a row in BxCRS is slightly more expensive than that in xCRS. Hence we have $C_{hy_blk} \leq C_{x_row} \leq C_{bx_row}$.

By comparing the approximate costs in Table 4.1, we can conclude the following. For computing multiple aggregates in one scan of the multi-dimensional data, BESS is the most efficient storage scheme, followed by Hybrid, xCRS, and then BxCRS in that order. Unlike other operations, such as sub-array retrieval in MDSA, multi-dimensional aggregation requires visiting every non-zero values in the MDSA instead of only part of the valid elements. BESS organizes an MDSA as one-dimensional arrays of all the non-zero values, and their corresponding array indexes encoded in BEI. Moreover, BEI simplifies the array index comparisons to a single, in most cases, integer value comparison. These characters make BESS very suitable for carrying out multi-dimensional aggregations. Compared with BESS, PTCS incurs an overhead of traversing all the nodes in PATRICIA trie to compute an aggregate function.

4.5 Computing the Cube

The *cube* computation supports an important class of decision support queries in data warehousing and OLAP systems. Computing the cube for an MDSA of k dimensions involves the computation of aggregations at 2^k different granularities, where each granularity is one of the 2^k possible subsets of the dimension set $\{d_{k-1}, \ldots, d_0\}$. Clearly, this operation presents challenges on both space and speed. Our approach, in this research, to compute the cube is based on the classical algorithm, PipeSort, which was proposed by Sarawagi et al. [1]. An overview of the highlights in our approach is summarized as follows.

- Two storage schemes, PTCS and BESS, were used to represent the MDSA in computing the cube, and the cube results were represented using BESS.

- In generating the paths in the search lattice, we applied the algorithm, $Paths(\{d_0, \ldots, d_{k-1}\})$, introduced by Ross et al. [38]. The algorithm determines an optimal set of paths, hence minimum number of sorts.
- Sorting the data in BESS was done by sorting the BEIs as single integers, which greatly simplified the sorting process of multi-dimensional data. For PTCS, the sort operation was replaced by reconstruction of a PATRICIA trie, and traversing the new trie in pre-order.

The comparative analyses in Section 4.4 show that BESS is the most efficient method for computing the multi-dimensional aggregation. PTCS is similar to BESS except that it stores the same list of non-zero elements in a PATRICIA trie. Considering this similarity, we only chose PTCS and BESS to represent the MDSA when computing the cube.

It is not always that the entire data can fit into memory when performing complex operations, such as computing the cube, over massive volume of data in data warehouses. Divide and conquer is an effective method that often being applied in such circumstances. This requires that we partition the data into smaller fragments first, then carry out the operation over the data fragments independently, and merge the results at the end. In this work, however, we only aim at developing efficient methods to compute the cube over the data that can fit in memory.

4.5.1 Paths in the Search Lattice

In sort-based cube computing algorithm, a sort operation is performed at the root node of each path in the search lattice. To minimize the cost of sorting operation, it is desirable to partition the search lattice into a minimum number of paths that cover all the nodes in it. Ross et al. argued in [38] that $\binom{k}{\lceil k/2 \rceil}$ paths are the minimum number of paths that cover all the nodes in a search lattice with k attributes. Based on their brief argument, we prove this claim as follows.

For a search lattice with k attributes, there exists $k+1$ levels, i.e., level 0, level 1, ..., level k. See the example of Figure 2.4 on Page 19. At any level i ($0 \leq i \leq k$), there are exactly $\binom{k}{i}$ nodes, and each node consists of i attributes. The value $\binom{k}{i}$, $0 \leq i \leq k$, attains its maximum at $\binom{k}{k/2}$ when k is an even number, and $\binom{k}{\lfloor k/2 \rfloor}$, or $\binom{k}{\lceil k/2 \rceil}$, when k is an odd number. Hence, at level $\lfloor k/2 \rfloor$, or level $\lceil k/2 \rceil$, the search lattice has the maximum number of nodes. Moreover, no path in the search lattice passes more than one node at the same level. Thus there must exist $\binom{k}{\lfloor k/2 \rfloor}$, or$\binom{k}{\lceil k/2 \rceil}$, paths in the search lattice to cover all the nodes.

The Algorithm 7, $Paths(\{d_0, \ldots, d_{k-1}\})$, presents a constructive procedure that generates an optimal set of

paths in the search lattice [38]. The optimal set consists of precisely $\binom{k}{\lceil k/2 \rceil}$ paths. The main idea of this algorithm is based on the symmetry of the search lattice. For any level i ($0 \leq i \leq k$), there exist a level $k-i$, that the two levels have the same number of nodes, i.e., $\binom{k}{i} = \binom{k}{k-i}$. According to this property, we may construct the paths in the following way. We start from level k, choose one node at each consecutive level (in decreasing order), until we reach level 0. Next, we consider level $k-1$. We construct paths that start at level $k-1$, and end at its symmetric level, i.e., level 1. One of the nodes on level $k-1$ is already in the first path we constructed. The same is also true for level 1. Thus, we can construct exactly $\binom{k}{k-1} - \binom{k}{k}$ paths, each of them starting at level $k-1$ and ending at level 1. The same process is carried out level by level until we reach the level $\lceil k/2 \rceil$. The total number of paths constructed in this process will be

$$\binom{k}{k} + \binom{k}{k-1} - \binom{k}{k} + \binom{k}{k-2} - \binom{k}{k-1} + \cdots + \binom{k}{\lceil k/2 \rceil} - \binom{k}{\lceil k/2 \rceil - 1} = \binom{k}{\lceil k/2 \rceil} \quad (4.1)$$

Hence, the algorithm $Paths(\{d_0, \ldots, d_{k-1}\})$ constructs exactly $\binom{k}{\lceil k/2 \rceil}$ paths, which is the minimum number of paths required in a search lattice to cover all the nodes. The paths generated by executing the algorithm $Paths(\{d_0, \ldots, d_{k-1}\})$ on 4 attributes D, C, B, A (attribute D corresponds to d_0, C to d_1, and so on) are as follows. There are 6 ($\binom{4}{2} = 6$) paths in $G(4)$.

$$\begin{aligned} G(4) = & A \cdot B \cdot C \cdot D \rightarrow A \cdot B \cdot C \rightarrow A \cdot B \rightarrow A \rightarrow \phi \\ & A \cdot B \cdot D \rightarrow A \cdot D \rightarrow D \\ & A \cdot C \cdot D \rightarrow A \cdot C \rightarrow C \\ & B \cdot C \cdot D \rightarrow B \cdot C \rightarrow B \\ & B \cdot D \\ & C \cdot D \end{aligned}$$

The attribute orders in $G(k)$ are reordered to obey the prefix property. By prefix property, we mean the nodes in a path share the maximum length of prefix attributes among them, as well as between each adjacent nodes. Following the above example, the 6 paths in $G(4)$ after prefix sorting become the following.

$$G(4)_{prefix-sorted} = A \cdot B \cdot C \cdot D \rightarrow A \cdot B \cdot C \rightarrow A \cdot B \rightarrow A \rightarrow \phi \quad (1)$$

$$D \cdot A \cdot B \rightarrow D \cdot A \rightarrow D \quad (2)$$

$$C \cdot A \cdot D \rightarrow C \cdot A \rightarrow C \quad (3)$$

$$B \cdot C \cdot D \rightarrow B \cdot C \rightarrow B \quad (4)$$

$$B \cdot D \quad (5)$$

$$C \cdot D \quad (6)$$

To compute the *cube* in PipeSort, one sort operation is performed for each path according to the attribute order in the root node. Then all the cuboids in a path are computed in a pipelined fashion.

Algorithm 7: $Paths(\{d_0, \ldots, d_{k-1}\})$

Input: Attribute set $\{d_0, \ldots, d_{k-1}\}$ of an MDSA.

Output: A set $G(k)$ of $\binom{k}{\lceil k/2 \rceil}$ paths in the search lattice that cover all the nodes.

begin

if $k = 0$ **then**

Return a single node with an empty attribute list, ϕ

else

Let $G(k-1) \leftarrow Paths(\{d_0, \ldots, d_{k-1}\})$

Let $G_l(k-1)$ and $G_r(k-1)$ denote two replicas of $G(k-1)$

Prefix the attribute list of each node of $G_l(k-1)$ with d_{k-1}

for *Each path* $N_1 \rightarrow \ldots \rightarrow N_p$ *in* $G_r(k-1)$ **do**

Remove node N_p and the edge into N_p (if any) from $G_r(k-1)$

Add node N_p to $G_l(k-1)$

Add an edge from node $d_{k-1}.N_p$ to node N_p in $G_l(k-1)$

Return the union of the resulting $G_l(k-1)$ and $G_r(k-1)$

4.5.2 Computing the CUBE Using BESS

Efficient sorting plays an important role in the sort-based *cube* computation. In Section 4.5.1, we discussed how to generate an optimal set of paths in the search lattice so that the number of sorts are minimized when computing the *cube*. In order to compute the *cube* for an MDSA represented using BESS, we first address the problem of how to perform the sort operation efficiently for the BESS data.

Recall that the array index of each non-zero element of an MDSA is represented using BEI in BESS. BEI encodes each array index into a single integer value. To sort the MDSA data in a certain dimension order, we need to sort the corresponding BESS data according to their BEIs. With regard to sorting the BESS data, we can make the following claim.

Claim 4.1. *Sorting the BEIs as integers sorts the BESS data correctly on the corresponding array indexes.*

Proof. **Inductive basis:** If $n = 1$ (one-dimensional array), the value of BEI equals the corresponding array index on dimension d_0. Sorting on the BEIs is equivalent to sorting on the array indexes. The Claim 4.1 is true for $n = 1$.

Inductive step: Assume the Claim 4.1 is true for $n = k - 1$, i.e., the BEIs are already correctly sorted on the lower $k - 1$ index values. Let us prove the claim is also true for $n = k$. We concatenate the index values on dimension d_{k-1} with the BEIs of the lower $k - 1$ dimensions at the leftmost valid bit position. Consider two

elements $a = A\langle p_{k-1}, \ldots, p_0\rangle$, $b = A\langle q_{k-1}, \ldots, q_0\rangle$, with their new BEIs on k dimensions denoted as a_{bei} and b_{bei} respectively. If $p_{k-1} > q_{k-1}$, then $a_{bei} > b_{bei}$, regardless of the order of the BEIs on the lower $k-1$ dimensions. Similarly, if $p_{k-1} < q_{k-1}$, then $a_{bei} < b_{bei}$. If $p_{k-1} = q_{k-1}$, the order of a_{bei} and b_{bei} maintains its original order on the lower $k-1$ dimensions. Hence, Claim 4.1 is true for $n = k$. By mathematical induction, sorting the BEIs as integers sorts the BESS data correctly on the corresponding array indexes. □

Among the various efficient sorting algorithms, we chose *radix sort* to sort the BESS data. Radix sort is often used to sort keys with multiple fields. In our implementation of radix sort, we used a stable sort, *counting sort*, to sort the bit fields in BEIs in a number of passes. Given $\mathcal{N}_{nz}$ b-bit numbers and any positive integer $r \leq b$, radix sort sorts these numbers in $\Theta(b/r)(\mathcal{N}_{nz} + 2^r)$ time if the stable sort it uses takes $\Theta(\mathcal{N}_{nz} + q)$ time for inputs in the range of 0 to q [6]. We choose $q < \mathcal{N}_{nz}$ so that the counting sort in radix sort takes time $\Theta(\mathcal{N}_{nz})$. Further, if we choose $r \approx \log \mathcal{N}_{nz}$, then the radix sort gives us a linear time sorting. The Algorithm 8 shows the process of computing the *cube* for an MDSA represented using BESS. In Algorithm 8, Line 5 reorders the attribute values in the BEIs. For example, assuming the BEIs are currently in dimension order of $ABCD$, and the new order is of BCA; we extract the attribute values from a BEI in the current dimension order, then rearrange them into a new BEI in the new dimension order. For Line 7, we use the method for computing multiple aggregations introduced in Section 4.1. Line 8 writes the aggregation results of a path to the disk.

Algorithm 8: BESSCube$(bei, val, \{d_{k-1}, \ldots, d_0\})$

Input: The BESS arrays, bei and val, representing $A[D_{k-1}], \ldots, [D_0]$; the attribute set, $\{d_{k-1}, \ldots, d_0\}$, of A.

Output: The cube results for A over $\{d_{k-1}, \ldots, d_1\}$.

begin

`/* Generate the set of paths,` $G(k)$`, in the search lattice */`

$G(k) \leftarrow Paths(\{d_0, \ldots, d_{k-1}\})$

$G(k) \leftarrow PrefixSort(G(k))$

for *Each path,* $N_1 \rightarrow \ldots \rightarrow N_p$*, in* $G(k)$ **do**

5 Reset the attribute order of the BEIs in bei according to the dimension order in node N_1

`/* Sort` bei `and` val `*/`

6 $Sort(bei, val)$

7 Compute the multiple aggregations

8 Output the aggregation results

4.5.3 Computing the CUBE Using PTCS

In computing the *cube* for an MDSA represented using PTCS, we take a similar approach to the one in Algorithm 8. The difference between these two procedures lies in the way we sort the valid elements in the MDSA. For PTCS, we construct a new PATRICIA trie from an existing one by changing the dimension order in the current key for each node. The pre-order traversal in the resulting PATRICIA trie gives us a sorted list of the valid elements on the desired dimension order. In order to apply the Algorithm 8 to compute the *cube* using PTCS, we need to replace the functions on Lines 5, and 6, with the ones for traversing the current PATRICIA trie and constructing a new PATRICIA trie.

Chapter 5

Selected Array Operations on GPUs

5.1 Overview

The trends of multi-core and many-core microprocessor architectures, coupled with the compute- or data-intensive computing, require parallelism in both system and application softwares. There are a number of parallel programming models which span a wide range of hardware systems, from the supercomputing facilities at the high end to the normal PCs and the mobile devices at the lower ends. Our focus, in this research, is on the heterogeneous programming based on the multi-core CPU systems equipped with many-core GPUs as accelerators. GPUs, as one of the most prominent type of accelerator, offer high performance, good energy efficiency and low price. The main challenge with GPUs, however, is that they are only suitable for certain type of problems, particularly problems with sufficient data parallelism, regularity in control flow and memory access patterns. As we already indicated in the previous chapters, data warehousing and OLAP systems process and analyze large volumes of data sets, and require fast response time. Parallelism plays an important role in these type of applications. In this chapter, we investigate applying heterogeneous computing to some of the operations and in particular, the problem of the *cube* computation in data warehousing and OLAP. More specifically, based on the CPU-only implementations of various operations discussed throughout Chapters 3 and 4, we use GPUs to accelerate the data parallel tasks in the basic array operations, namely, large scale random element access and sub-array retrieval, as well as in computing the *cube*. By large scale random element access we mean accessing multiple array elements in large data set in a single operation (or function call). Our basic approach, in this regard, is to utilize both CPU and GPU in a computing task so that the overall performance can be improved.

We follow a design cycle, *assess, parallelize, optimize* [31], in the process of mapping the serial codes, or parts of them, to GPU. Each part of this cycle is discussed in more detail as subsequently.

Assess: A problem, that is particularly suitable for GPU computing, usually has plenty of data level parallelism, and the data can be processed independently on different processors for the same set of operations.

In this case, each thread in a GPU executes the operation on a small fraction of the data independently and in any order. Control flow is also an important factor in successful GPU accelerations. The problems with regular control flows can be processed on GPUs more efficiently. Unlike CPUs, GPUs have limited advanced processing capabilities, such as branch prediction and out-of-order execution. Such an architectural feature also imposes restrictions on the types of data structure that can be implemented efficiently on GPUs. Irregular data structures, such as a trie, are difficult to share among the GPU threads. Furthermore, recursion, which is needed to traverse a trie, is not supported on the GPU [11].

Parallelize: The tasks that are offloaded to GPUs are executed by CUDA kernels. A CUDA kernel is executed in parallel N times if there are N threads. To invoke a CUDA kernel, the host CPU specifies the number of threads to be created on GPUs in a hierarchical manner, i.e., the number of thread blocks, and the number of threads in each block. The numbers of threads and blocks should be chosen carefully in order to optimize the overall performance. A sufficient number of threads is necessary to saturate memory bandwidth. However, the number of threads in a block is often limited and hardware dependent. Similarly, a proper number of blocks is needed to keep all the streaming processors busy, and to achieve a good load balance at the same time. In this regard, different configurations can be tested to find an optimal one.

Optimize: There is a number of optimization rules frequently applied in CUDA C programming. The ones that are of particular interest in this research are as follows.

Memory hierarchy: Understanding the memory hierarchy of GPUs is crucial in programming them. Global memory access should be coalesced whenever possible to utilize the high memory bandwidth the GPUs offer. For example, if the memory interface width is 256 bit, at a double rate a total of 64 bytes are transferred with each memory block. When any amount of data, as little as one byte, is requested, the whole 64 byte block the data belongs to is actually transferred. By coalescing the memory access, we can increase the number of useful bytes in each memory block transferred. Hence, memory bandwidth could be utilized more efficiently. Due to the high latency of global memory, frequent access of it incurs heavy runtime penalty. Shared memory, on the other hand, offers a much lower latency. Frequently accessed data should be kept in shared memory during one kernel invocation. Constant and texture (read-only) memory in GPUs play the similar role as the cache for CPUs. We can use them for the data that remain constant.

Data transfer between the host and device: The data movement between the host and device is usually via PCIe channels. The data transfer rate of PCIe bus is much lower than the GPU local memory bandwidth, and similar to the rate of front side bus in the host system. To reduce the impact of data transfer between CPU and GPU, we should reuse the data as much as possible among different kernel invocations. Another way is to increase the amount of information within each transfer, i.e.,

transfer the data in a more compact form, such as using an efficient storage scheme to organize the data, or using a data compression method.

In the following sections, we present the algorithms and their implementations for the selected array operations, random element access and sub-array retrieval in large data sets, as well as the problem of the cube compution. Three storage schemes, xCRS, Hybrid and BESS, are chosen to represent the data sets in CPU+GPU co-processing. Due to the similarity of xCRS and BxCRS, we implement only one of them, xCRS on GPUs. In particular, the PTCS is not going to be considered for GPU implementation due to the irregular data structure of a trie.

5.2 Large Scale Random Element Access

Random element access is probably the most basic operation in data warehousing. Although it is theoretically a very simple operation, efficient access of single element in a large data set is not only desirable in data warehousing, it could also lead to improved performances in other operations, such as selection. In the mapping of CPU-only implementation to CPU+GPU co-processing, we consider the case where single or multiple elements can be retrieved in single operation, which means a single kernel invocation is needed. Note that random element access is also considered as a search operation in this work.

For the MDSAs represented using BESS or Hybrid method, *binary search* is one of the key algorithms in the operation of random array element access. Thus, a parallel search algorithm on large sorted data sets using GPUs could improve the performance. For this, we applied the *p-ary search* [22], as the parallel counterpart of binary search, for searching the large data sets on GPUs. An illustration of p-ary search is depicted in Figure 5.1. In each iteration, all the threads in a thread block compare the searched key with the first and last elements in one of the disjoint data segments. The segment which contains the key is again partitioned among the threads in the next iteration. This process is repeated until the size of the data segment to be partitioned is less than the number of threads in a block. The Algorithm 9 described in detail [22] gives the GPU implementation of p-ary search. In Algorithm 9, each GPU thread block searches for a unique key, and the threads in a block cooperate to search for the same key. Each thread in a block needs to access the global memory (see Line 9) once in each iteration. These accesses within a thread block are hard to be coalesced for the data in array format, except for the last data segment (see Line 14). To reduce the number of global memory accesses, once the data was read in from the global memory, it was stored in the shared memory for reuse. Note that the last element of a data segment is also the first element in the next data segment (see Figure 5.1). Hence, global memory access can be reduced by half. If the number of threads is p, a key is found in $O(\log_p N)$ time bound using *p-ary search*,

instead of $O(\log_2 N)$ using *binary search*.

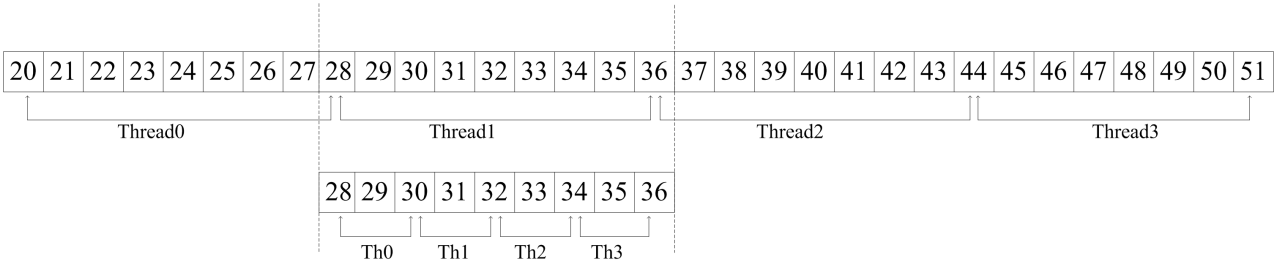


Figure 5.1: An example of *P-Ary* search. The searched Key is 31 and 4 threads are used.

For the MDSAs represented using xCRS or BxCRS, a random element access is highly efficient on CPUs, even for a large data set. However, in the case of accessing or searching large number of elements, GPUs can be used to process such tasks in parallel. For this purpose, we applied GPUs as follows. Firstly, the row_offset of each searched element is computed by one GPU thread. Secondly, we use one thread or one thread block to search for each element using their row_offset. In the former case, the control flow within a single warp [1] diverges easily, and the threads within the same warp could end up waiting for the one which has the worst case. In the latter case, we use only one thread in each block to compute the range of a row, then all the threads participate in searching within that range. The drawback of this approach is that only one thread is active during the computation of the range of a row. However, the global memory accesses are coalesced when searching within a row. In our implementation of GPU accelerated searching in xCRS and BxCRS data, we chose the first case considering the sparsity of the array data. For searching in the data represented using Hybrid, we took the second approach.

5.3 Sub-Array Retrieval

Our approach to retrieve a sub-array in the MDSA, represented using one of the array based storage schemes, is based on data parallel sequential scan. In a sequential scan, all the array elements within a range are checked against a certain condition, e.g., index comparisons in sub-array retrieval. If the condition is satisfied, then the element belongs to the results, otherwise it is filtered out. A detailed implementation of CUDA kernel, for retrieving a sub-array in an MDSA represented using BESS, is given in Algorithm 10. At the host, i.e., CPU, side, we first determine the range in the BESS array bei, then invoke the kernel. One of the advantages of sequential scan on GPU is that the global memory access is easily coalesced. The comparisons of the indexes (Lines 12 and 13) are done separately to keep the control flow more consistent within a single warp. To reduce the usage of device memory, as well as the data transfer between the host and device, we store only indication

[1] A warp is the unit of thread scheduling in streaming multiprocessors [24]. The size of a warp is typically 32 threads.

Algorithm 9: $ParySearch(data[0..length-1], skeys[..])$

Input: A sorted list of data, $data[0..length-1]$, the keys, $skeys[..]$, to be searched.

Output: The search results, $results[..]$.

begin

```
/* threadIdx.x and blockIdx.x are the built in variables in CUDA, which are
   used to identify a thread or thread block respectively.                  */
/* Allocate the space in shared memory                                      */
```

__shared__ int $d[0..blocksize-1]$

__shared__ int $range_length \leftarrow length$

__shared__ int $range_offset \leftarrow 0$

int $mykey \leftarrow skeys[blockIdx.x]$

while $range_length > blocksize$ **do**

```
/* Compute the new range length                                             */
```

$range_length \leftarrow \lceil range_length/blocksize \rceil$

$range_start \leftarrow range_offset + threadIdx.x * range_length$

9 $d[threadIdx.x] \leftarrow data[range_start]$

$__syncthreads()$

if $mykey >= d[threadIdx.x]$ *and* $mykey < d[threadIdx.x+1]$ **then**

$range_offset \leftarrow range_start$

$range_start \leftarrow range_offset + threadIdx.x$

14 **if** $mykey = data[range_start]$ **then**

$results[blockIdx.x] \leftarrow range_start$

values, such as 1 for true, 0 for false, in the result array (see Line 15). The result is then transferred back to the host for further processing.

The CPU+GPU co-processing of sub-array retrieval, in the MDSAs represented using xCRS, BxCRS, or Hybrid, are implemented in two steps. First, examine each 'block' to determine if it belongs to the sub-array to be retrieved. A 'block' refers to a *row* in the case of xCRS or BxCRS, and an $(r-1)$-dimensional array in the case of Hybrid. For xCRS and BxCRS, we compare the *row* index, i.e., $\langle n_{k-1}, \ldots, n_1 \rangle$ with the boundary *row* index of the sub-array defined by index L and H to determine if the *row* examined falls in the sub-array or not. For Hybrid, we compare the 'block' index $\langle n_{k-1}, \ldots, n_{r-1} \rangle$ with $\langle l_{k-1}, \ldots, l_{r-1} \rangle$ of index L and $\langle h_{k-1}, \ldots, h_{r-1} \rangle$ of index H. Second, within each 'block' that falls in the boundary of the sub-array, scan the array elements to select the ones that belong to the sub-array. Each of these two steps offer certain degree of data parallelism. Hence, there are two options for implementing them on GPUs. One is to implement the two

Algorithm 10: $gpuRetrieve_bess(d_bei, st_ind, end_ind, st_offset, end_offset)$

Input: BESS array bei, starting and ending index of the sub-array, and their corresponding offsets.

Output: The retrieved keys $d_results[..]$

begin

__shared__ int $maxSize$

int $myId = blockIdx.x * blockDim.x + threadIdx.x$

int $flag, ind[DIM], key$

if $threadIdx.x == 0$ **then**

$maxSize = end_offset - st_offset + 1$

`/* Synchronize threads within a thread block */`

$__syncthreads()$

$d_results[myId] = 0$

if $myId < maxSize$ **then**

$key = d_bei[st_bei + myId]$

`/* Call the function to compute the array index from a BEI */`

11 $gpuDeKey(key, ind)$

`/* Call the function to compare array indexes, returns TRUE if` $index$ `is greater than` st_ind `*/`

12 $flag = gpuIsGreater(index, st_ind)$

13 $flag = flag \&\& gpuIsLess(index, end_ind)$

14 **if** $flag$ **then**

15 $d_results[myId] = 1$

steps in one kernel, and the other is to implement each step in one kernel. We adopted the first option for the case of xCRS, or BxCRS, and the second option for Hybrid. The main concerns here are to gain better load balance and ensure enough data parallelism, particularly for the second step. In the xCRS, or BxCRS, data, it is possible to have not enough data parallelism within a single row when the array is sparse. Thus, we combined the two steps into one kernel on the expense of load imbalance. In the case of Hybrid, we can expect enough data parallelism within a 'block', especially when the dimensionality is high, so that using two kernels gives us better load balance.

5.4 Computing The Cube

Based on the work of computing the *cube* in Section 4.5, we discuss in this section how to accelerate this computation using GPUs. The problem of computing the cube not only demands speed and space, it also requires data parallelism. Lots of work has been done on parallelizing the cube computation, especially on multi-core and clusters [7, 8]. Our approach, to parallelize computing the cube for MDSAs in this research, is to represent the MDSA using one of the storage schemes, and utilizing GPUs for co-processing with CPUs. With regard to the storage scheme, we chose BESS due to its space efficiency, and suitability for computing multi-dimensional aggregation. In the CPU+GPU co-processing, we offload some of the tasks in computing the cube to GPUs for acceleration.

The algorithm given in Algorithm 8 on Page 57 for computing the cube consists of a number of independent tasks. We revise the major tasks among them as follows and analyze their suitability for GPU processing.

1. Generate the paths in the search lattice. This task does not have enough data parallelism, and it is best suited for CPU processing.
2. Reset the attribute order of the BEIs. We need to reset the attribute order in the BEIs for each path we generate in the search lattice. For example, if the root node of a path is in attribute order $ABCD$, and the one in the next path is BDC, we reset the order of the attributes for all the array elements in their BEIs before we go on to the sorting step. We offload this part to the GPUs in our implementation.
3. Sort the data according to the attribute order in the root node of a path. This part actually takes up most of the time in the sort-based cube computation. Accelerating the sort operation dramatically improves the speed of the cube computation. We applied the highly efficient GPU sort function from the Thrust library [20] to sort the data in our implementation.
4. Compute the multiple aggregations in one scan of the data. Although the aggregation operation offers good parallelism, we compute multiple aggregations in a pipelined fashion, which are not data independent. Hence, we chose to implement this task on CPUs.

5.4.1 Resetting the Attribute Order

In Section 4.5.1, we showed that for a k-dimensional sparse array, $\binom{k}{\lceil k/2 \rceil}$ paths are the minimum number of paths that cover all the nodes of the search lattice with k attributes. Moreover, we discussed the Algorithm 7, that generates the optimal set of paths that contain exactly $\binom{k}{\lceil k/2 \rceil}$ paths [38]. Consider the prefix sorted

paths $G(4)_{prefix-sorted}$, for a 4-dimensional array generated by the Algorithm 7. The four dimensions are represented by A, B, C, D respectively. For example, to reset the attribute order from $A \cdot B \cdot C \cdot D$ in Path 1 of $G(4)_{prefix-sorted}$ to $D \cdot A \cdot B(\cdot C)$ in Path 2, we extract the dimension values in the order of D, A, B, C from each BEI, then concatenate them into a new BEI. We replace the previous BEIs with the new BEIs, so that no extra GPU memory space is required. In our implementation of CPU+GPU co-processing, the original input data is represented using BESS, and the attribute order of the BEIs is the same as the order of the root node in the first path of $G(k)_{prefix-sorted}$, i.e., Path 1 of $G(4)_{prefix-sorted}$ in the case of $k = 4$. Note that the input data needs to be transferred from the CPU memory to GPU memory only once, and no extra GPU global memory is required beyond the input data. In the CUDA kernel invocation, we assign each thread to reset the attribute order of one BEI, so that global memory access is coalesced.

5.4.2 Sorting

The array elements with new BEIs need to be sorted for grouping in order to compute multi-dimensional aggregation. For this purpose, we applied the sorting function in the Thrust template library for CUDA [20]. The important aspects we exploited in Thrust include the following three abstract interfaces: *container*, *iterator* and *fundamental parallel sorting algorithm*.

Container: Thrust provides two vector containers, *host_vector* and *device_vector*. The former resides on the host memory, and the latter on the device memory. These two containers are both generic and they hide the explicit memory allocation and deallocation. The Listing 5.1 shows the code snippets in our implementation that defines the host and device vectors, and also copies data from the host to the device.

```
......
//allocate host vectors with no_nnzs elements
thrust::host_vector<int> h_keys_vec(no_nnzs);
thrust::host_vector<float> h_values_vec(no_nnzs);
//allocate device vectors with no_nnzs elements
thrust::device_vector<int> d_keys_vec(no_nnzs);
thrust::device_vector<float> d_values_vec(no_nnzs);
......
//copy data from host to device
d_keys_vec = h_keys_vec;
d_values_vec = h_values_vec;
......
```

Listing 5.1: The code snippets to define Thrust containers and copy data from the host to the device

Iterator: Thrust uses a pair of iterators to define the range in a sequence. Iterators act like pointers in C/C++ language, and they can be inter-operated with raw pointers such as C/C++ pointers. The Listing 5.2 shows the relevant code snippets in our implementation. The interoperability makes it possible to apply Thrust algorithms for raw pointers or apply the results of Thrust algorithms in CUDA C algorithms.

```
int *keys; float *values, *d_keys;
......
//obtain raw pointer to host vector's memory
keys = thrust::raw_pointer_cast(h_keys_vec.begin());
values = thrust::raw_pointer_cast(h_values_vec.begin());
//obtain raw pointer to device vector's memory
d_keys = thrust::raw_pointer_cast(d_keys_vec.begin());
......
```

Listing 5.2: The code snippets showing the interoperability of Thrust iterators and raw pointers

Parallel Sorting Algorithm: We applied sorting function $sort_by_key()$ in the Thrust to sort the data represented using BESS. Function $sort_by_key()$ performs a key-value sort, i.e., the function sorts the key-value pairs according to the key. Thrust statically selects one of the two sorting algorithms. One is a highly optimized radix sort for the primitive data types, such as $char, int, float$ in C, and the other is a merge sort for all the other data types. It should be noted that the radix sort is considerably faster than the merge sort in Thrust [21]. The BESS representation of an MDSA consists of two vector arrays. One stores the integer keys and the other the measure data which is typically $float$ or $double$ data type in C. Thus in our implementation, the data types being used are both primitive data types. Radix sort is the algorithm selected in executing the $sort_by_key()$ function. As a result, the sorting process is done very efficiently, hence the performance of cube computing is greatly improved.

Finally, we use the multi-dimensional aggregation method presented in Section 4.1 to compute the aggregation for each path on CPU. However, the length of each path differs. The aggregation process computes the same number of aggregations as the length of each path in the implementation. For example, 4 aggregates are computed for Path 1 in $G(4)_{prefix-sorted}$, while only 1 aggregate needs to be computed for Path 5.

Chapter 6

Experimental Setup

6.1 Experimental Environment

All the serial computations in our experiments were implemented in the C programming language, and hybrid computations in CUDA C. The compilers being used are GNU GCC $4.6.3$ for C, and NVIDIA NVCC 4.0 with the options "$-arch = compute_30$" and "$-code = sm_30$" for CUDA C. The Thrust library being used in our experiment was the version released in the CUDA Toolkit 4.2.

The experiments were run on a PC with Intel(R) Core(TM) $i7 - 3770$ multi-core processor (with hyper-threading enabled) at 3.40GHz and 8GB memory, running Ubuntu 64-bit Linux 12.04. The GPU used in our system is Nvidia GeForce GTX 670 GPU at 1.08GHz and 1344 cores, with 2GB GDDR5 device memory and 192GB/sec peak memory bandwidth.

6.2 Experimental Data

The following specifications are taken as the main factors of an MDSA that have impacts on the performance of a storage scheme, for the selected operations. They are:

- The number of dimensions;
- The number of non-zero elements;
- The cardinality of each dimension;
- The sparsity;
- The distribution of those non-zero values within the array structure.

To have better control of these specifications over the experimental data, and to fit the data in the main memory, we generated a set of synthetic data. This leaves out the option of using the benchmark data of decision support system from the Transaction Processing Performance Council (TPC) [45]. Synthetic data offers us the flexibility of the control of the specifications of an MDSA, especially the dimensionality, the sparsity, the number of non-zero elements and the cardinalities. The main drawback of using synthetic data is that it is hard to simulate the distribution of the non-zero elements in the multi-dimensional array. In our experiment, we mainly consider the case where the non-zero elements are distributed uniformly in an MDSA. Although effort is made to generate the synthetic data so that the non-zero elements are uniformly distributed in their corresponding arrays, this is not guaranteed. The distribution of the non-zero elements in an MDSA could have significant impact on the performances of various array operations under a certain storage scheme. This is exactly the reason why there are numerous storage formats for sparse matrices in the application of numeric methods. They are designed to optimize certain types of matrix operations, such as multiplication, lower and upper triangular decompositions etc., for matrices with certain structures. Certainly, we can take the similar approaches in dealing with the MDSAs of higher dimensions ($k > 2$). However, this is beyond our current project scope. In this dissertation, we only concentrate on some general data structures to represent MDSAs, and hopefully this could lead us to more specific ones in the future.

The synthetic multi-dimensional data consists of dimensional data and measure data. For the dimensional data, we generated them as integer values on four different dimensionalities of 2, 3, 4 and 8. The measure data is generated as floating point values. The data sets are in the Matrix Market format [4], and stored as ASCII files. For each of the 4 dimensionalities, we generated data files with 4 different sparsities. The sparsity levels chosen are 90%, 95%, 99%, 99.9%. On each sparsity level, 5 data files with 2, 4, 6, 8 and 10 million valid array elements are generated. The first line of a data file contains the cardinalities of each dimension followed by the number of non-zero elements. Each line in the rest of the file represents one non-zero array element using index-value pair format, and they are in the row-major order. As such, we generated two sets of data, termed as *data set A* and *data set B*. Both the data set A and B consist of 80 data files. The data set A was used for all the operations implemented in our experiment. The data set B was only used in the large scale random element access operation as the data to be accessed. Each data file in data set B corresponds to one data file in data set A. Every pair of such data files contain the array elements from the same sparse array, i.e., they have the same specifications such as cardinality, sparsity, etc. Table 6.1 shows the cardinalities, the number of non-zero elements and the size of the data files for a set of 4-dimensional experimental data (in data set A) generated for the sparsity $\sigma = 90\%$.

Once the dimensionality, the cardinalities and the sparsity of an MDSA are determined, we generate the array elements in the following manner. We first divide the size of the array, i.e., the product of the cardinalities, by the number of non-zero elements to get an average incremental value q. Then starting from an initial array index

D_3	D_2	D_1	D_0	$\mathcal{N}_{nz}$(million)	Size(MB)
40	50	100	100	2	41.2
20	100	100	200	4	83.9
40	100	100	150	6	126.2
50	80	100	200	8	169.8
50	100	100	200	10	212.5

Table 6.1: The specifications of a set of experimental data with k= 4 and $\sigma = 90\%$.

(randomly generated), we obtain a new array index by incrementing the initial array index by a random value, which is less than or equal to q. Similarly, we generate the next index by incrementing the newly obtained array index. The Algorithm 11 gives a detailed description of this process. A measure value is generated randomly for each array index, and it ranges between the floating point values 1.0 and 100.0. Each measure value is generated only at the time of writing an array index into the ASCII file. In this manner, we obtain an ASCII file which contains a list of the desired number of index-value pairs. Note that for the data set B, which contains the data files with the array elements to be searched, we only need to generate the dimensional data.

Algorithm 11: generateIndex(k, $card$, no_nnzs)

Input: The dimensionality k, the array $card[0..k-1]$ for the cardinalities, the number of non-zero values no_nnzs.

Output: The array $ind[0..no_nnzs][0..k-1]$ with no_nnzs number of array indexes.

begin

$N_k \leftarrow \prod_{j=0}^{k-1} card[j]$

`/* Compute the incremental value */`

$q \leftarrow N_k / no_nnzs$

for $i = 0$ *to* $k-1$ **do**

$ind[0][i] \leftarrow 0$

$cnt \leftarrow 1$

while $cnt < no_nnzs$ **do**

`/* Function` $rand()$ `generates a random integer */`

$tmp2 \leftarrow rand()\%q$

for $i = 0$ *to* $k-1$ **do**

$tmp1 \leftarrow ind[cnt-1][i] + tmp2$

if $tmp1 >= card[i]$ **then**

$tmp2 \leftarrow tmp1/card[i]$

$tmp1 \leftarrow tmp1\%card[i]$

else

$tmp2 \leftarrow 0$

$ind[cnt][i] \leftarrow tmp1$

$cnt \leftarrow cnt + 1$

Chapter 7

Performance Evaluation

Storage utilization and *computational efficiency* are the main performance criteria in the evaluation and comparisons of the different storage schemes for MDSAs. For storage utilization, we simply measure the storage space usage of each storage scheme for a given MDSA, as this reflects the performance on this criterion. The storage space usage of each scheme, namely, $\mathcal{S}_{xcrs}$, $\mathcal{S}_{bxcrs}$, $\mathcal{S}_{hybrid}$ and $\mathcal{S}_{ptcs}$, was computed in Chapter 3. More analyses and comparisons are given in this chapter. Computational efficiency, on the other hand, is measured according to the time to construct the scheme, retrieve a random element, retrieve a sub-array and compute multi-dimensional aggregation. We designed and implemented algorithms for these operations, and tested them on large synthetic data sets. In the following sections we present the experimental results and give some discussions.

7.1 Storage Utilization of Various Storage Schemes

To compare the storage utilizations of different storage schemes, we define a parameter *storage ratio* as the ratio of storage space of a scheme to the storage space of multi-dimensional array, i.e. $\mathcal{S}_{scheme}/\mathcal{S}_{MDSA}$. The graphs in Figure 7.1 show the storage ratios of various storage schemes. The storage ratios decrease for all the storage schemes when the sparsity increases. The storage ratios of PTCS and BESS are about 80% and 30% respectively for MDSAs with $\sigma = 90\%$, and decrease to 8% and 3% respectively for $\sigma = 99\%$. This shows that we save more space for MDSAs with higher sparsity. The reason for the higher storage ratio of PTCS is that a PATRICIA trie node in PTCS includes two 'pointers', or memory addresses, for its two child nodes. The storage space for the 'pointers', with a memory unit (or cell) address is usually represented using 8 bytes. The storage space for for these 'pointers' becomes dominant when the number of trie nodes increases. By setting *hybrid ratio* $\alpha = \frac{\lceil \alpha k \rceil + 1}{k}$ and then $r = \alpha k$, we can achieve storage ratios for Hybrid being very close to those of BESS. In the cases of xCRS and BxCRS, their storage ratios are similar and comparable to those of BESS and Hybrid when the sparsity is around 90% (see the left graph in Figure 7.1). However, when the sparsity increases to about 99% as in Figure 7.1, the storage ratios of xCRS and BxCRS both become worse,

but BxCRS varies at a much slower rate than xCRS.

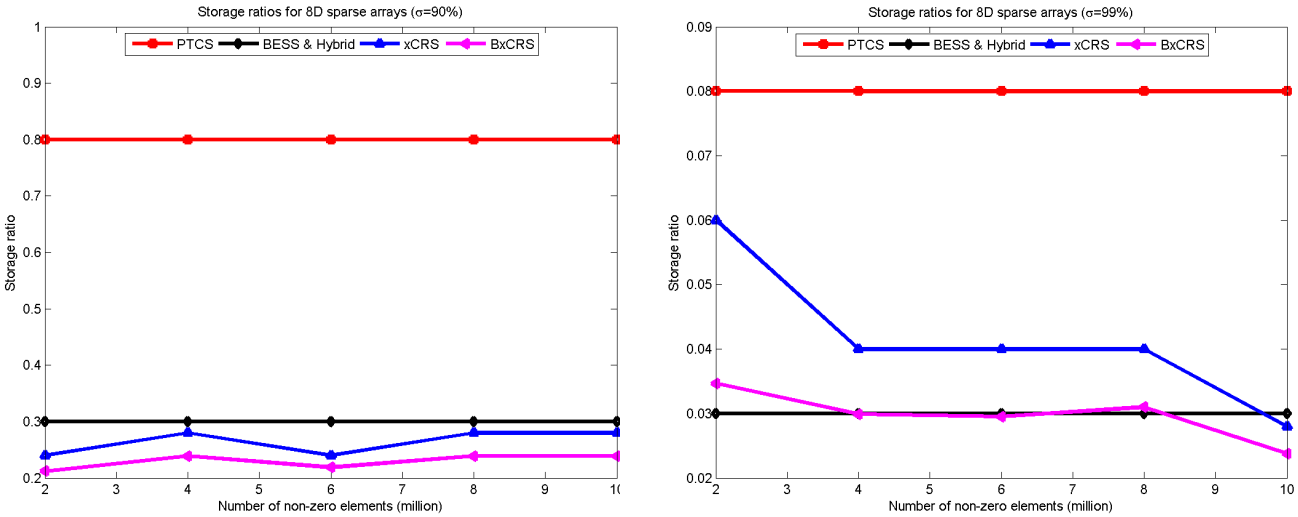


Figure 7.1: Storage ratios for the MDSAs with $k = 8$, $\sigma = 90\%$ (left) and $\sigma = 99\%$ (right).

7.2 Experimental Results and Comparative Analyses

The experiments are carried out in two parts, one is for the CPU only processing, and the other is for the CPU+GPU co-processing. The results of these experiments together with additional analyses based on both theoretical and experimental results are presented in the following sub-sections.

7.2.1 Results on CPU Only Processing

7.2.1.1 Performance of Storage Scheme Construction

Given an MDSA with $\mathcal{N}_{nz}$ non-zero elements, the complexities of constructing all the storage schemes concerned in this research, namely xCRS, BxCRS, Hybrid, PTCS and BESS, are $O(\mathcal{N}_{nz})$, which is linear with respect to the number of non-zero elements in the sparse array. The running times for different methods differ only by a constant factor. The reason for the construction of PTCS being slower than the others is that PTCS is trie based and the other methods are array based. However, PTCS is insensitive to the order of input array elements, while the other methods require the input data to be pre-processed into a certain order. The construction time of various storage schemes are compared for 8-dimensional sparse arrays with varying sparsity in Figure 7.2. When the sparsity increases, the construction time of all the storage schemes increases slightly.

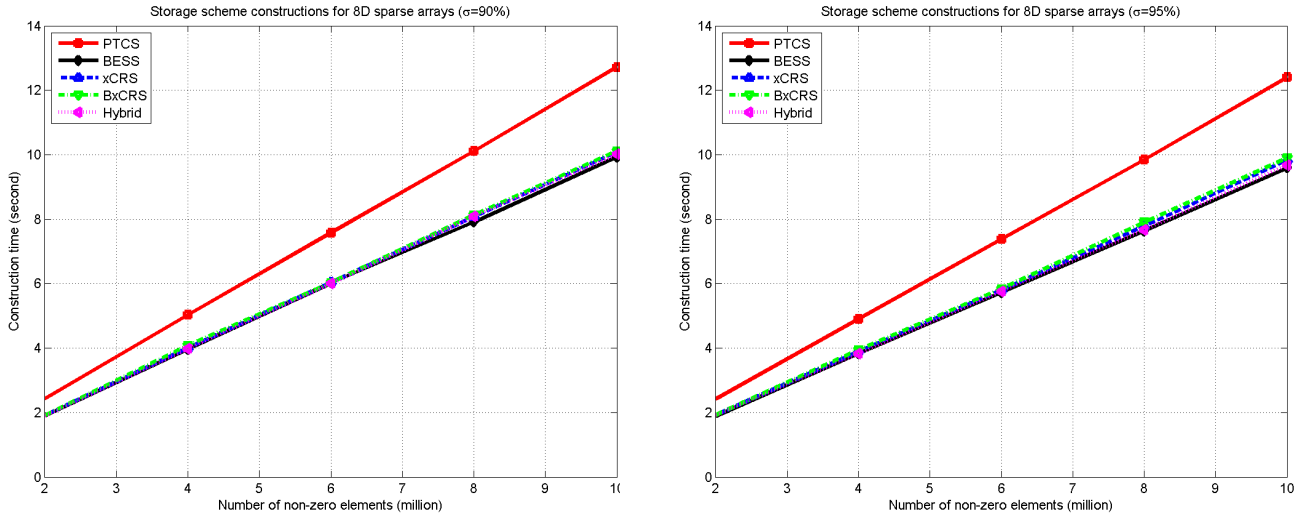


Figure 7.2: The construction time for MDSAs with $k = 8$, $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

This is due to the growing cardinalities of individual dimensions, even though the number of non-zero elements being handled remains the same.

7.2.1.2 Performance of Large Scale Random Element Access

In random element access or searching operation, we either retrieve the measure data associated with an array index if it exists, or determine that the array element does not exist. For this operation, we consider the case that a large set of data is collectively searched for in one operation. To evaluate the efficiency of random element access, we used the data sets that contain both existing and non-existing array elements in the original data sets. There were much more number of non-existing elements than those that exist. The total time for searching all the requested array elements in a data file is recorded, and the average time is taken. Figure 7.3 shows the results of the performance of the random element accesses of various schemes. The random element access in xCRS is the most efficient among all the methods compared. This is because the random array element access time in xCRS is $O(k) + O(\log_2 D_0)$. Since the array is sparse, the actual size of a row could be much less than the cardinality D_0 on dimension d_0. A random element access in BxCRS has the same time complexity as in xCRS. However, the performance of BxCRS is slightly worse than xCRS due to the fact that the computational cost of accessing a row in BxCRS is more expensive than in xCRS.

For sparse matrices, the method Hybrid becomes xCRS if we choose $r = 2$ (or $\alpha = 1$), which is the case in our experiment. Thus, for matrices, the performances of Hybrid and xCRS are almost the same as shown in both graphs in Figure 7.3. For 3-, 4- and 8-dimensional sparse arrays, the performance of Hybrid lies between those of xCRS and BESS. The lower the value of r is, the more efficient Hybrid performs on random

element accesses. By varying the hybrid ratio α, we can choose a favorable balance between the space and data manipulation efficiencies in Hybrid. In Figure 7.3, the results for Hybrid are for the cases where r is set to $3, 3, 5$ for 3-, 4-, 8-dimensional sparse arrays respectively.

The average time complexities of random element access in PTCS and BESS are both $O(\log \mathcal{N}_{nz})$. For BESS, we use the *binary search* algorithm to access a random array element. However, the performance of PTCS, shown in Figure 7.3, is worse than that of BESS. The main reason is that the traversal of a trie is more expensive than visiting an element in an array. The performance of PTCS can aslo be negatively affected by an imbalanced PATRICIA trie. The main advantage of the PTCS scheme is that it is the only one that tolerates *insertions* and *deletions* after the representative structures are built.

It is also shown in the graphs how the random element access algorithms for various schemes are affected by the different levels of sparsity. The PTCS and BESS are affected little by the sparsity because their random element access performance is mainly determined by the number of non-zero elements in the array. When the sparsity of the MDSA increases, there are some improvements in the performances of xCRS, BxCRS and Hybrid. These are because there are more empty rows (for xCRS and BxCRS), or more empty 'blocks' (for Hybrid), when the sparsity increases. It is very efficient to determine if a row, or 'block', is unoccupied in these 3 methods.

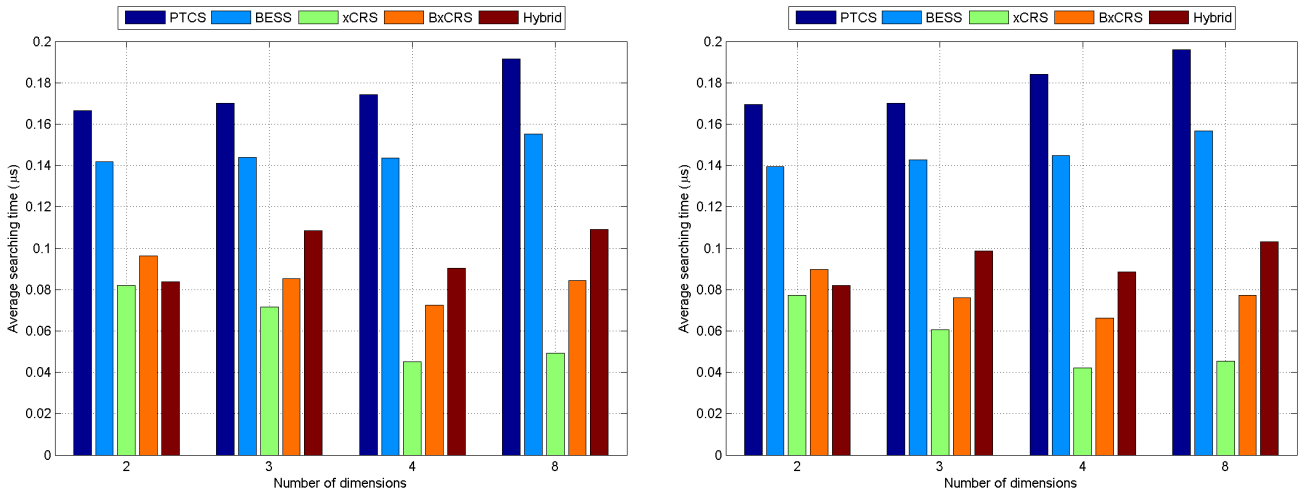


Figure 7.3: The average random element access time of various schemes for MDSAs with $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

7.2.1.3 Performance of Sub-Array Retrieval

To evaluate the performance of sub-array retrieval in the data structures of different storage schemes, three sub-arrays were retrieved for each input data set. The starting and ending indexes of the sub-arrays are chosen in a way so that different sub-spaces of the MDSA can be retrieved. The following three pairs of starting index and ending index define the 3 sub-arrays being retrieved in our experiment.

1. $L1 = \langle l1_{k-1}, \ldots, l1_0 \rangle$, $l1_i = 0$ where $0 \leq i \leq k-1$; $H1 = \langle h1_{k-1}, \ldots, h1_0 \rangle$, $h1_0 = D_0$, $h1_1 = \lfloor D_1 \times 0.25 \rfloor$, $\lfloor h1_i = D_i \times 0.95 \rfloor$ where $2 \leq i \leq k-1$.
2. $L2 = \langle l2_{k-1}, \ldots, l2_0 \rangle$, $l2_i = 0$ where $0 \leq i \leq k-1$; $H2 = \langle h2_{k-1}, \ldots, h2_0 \rangle$, $h2_0 = \lfloor D_0 \times 0.25 \rfloor$, $h2_1 = D_1$, $\lfloor h2_i = D_i \times 0.95 \rfloor$ where $2 \leq i \leq k-1$.
3. $L3 = \langle l3_{k-1}, \ldots, l3_0 \rangle$, $l3_i = \lfloor D_i \times 0.1 \rfloor$ where $0 \leq i \leq k-1$; $H3 = \langle h3_{k-1}, \ldots, h3_0 \rangle$, $h3_i = \lfloor D_i \times 0.85 \rfloor$ where $0 \leq i \leq k-1$.

The structures of these sub-arrays are further illustrated for a 3-dimensional array in Figure 7.4. The selected sub-arrays in Figure 7.4 are the rectangles in dotted bold line, and they are bounded by three pairs of indexes, $(L1, H1)$, $(L2, H2)$ and $(L3, H3)$. The efficiency of a sub-array retrieval is dependent on the structure of the sub-array. Given that the array elements are traversed in row-major order, i.e., the index values change the fastest on dimension d_0, then the retrieval of sub-array in (a) of Figure 7.4 is more efficient than the retrieval of the one in (b). The structure of sub-array in (c) of Figure 7.4 is chosen to be general so that it is less affected by the scan order of the array elements. In designing the sub-array retrieval algorithms for different schemes, we

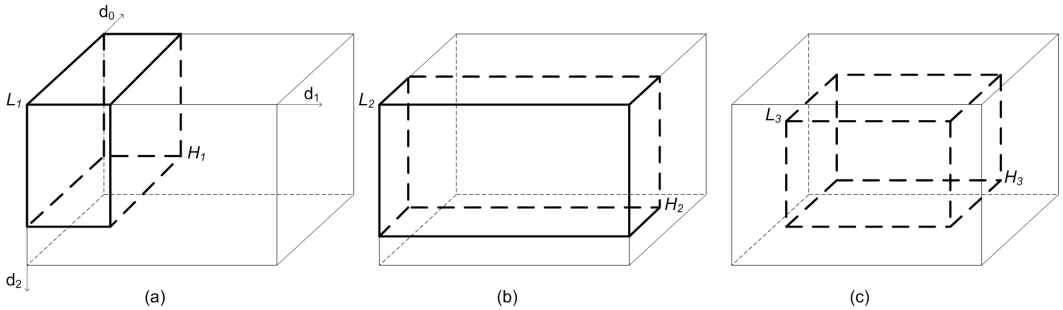


Figure 7.4: The structures of the sub-arrays to be retrieved ($k = 3$)

explored the locality of array elements in the various sparse array representations to some extent. In particular, the following properties were exploited.

1. Locating a 'row' in xCRS and BxCRS, or a 'block' in Hybrid, takes constant time;

2. The array elements in a particular row reside in the same branch of a PATRICIA trie.

For each MDSA, the average time of sub-array retrieval was taken as the total time to retrieve all the elements in the three sub-arrays divided by the number of these elements. This process was repeated for 5 MDSAs with $2, 4, 6, 8$ and 10 million non-zero elements on each sparsity. The final average time was obtained as the average value of 5 average time obtained above. The results are compared in Figures 7.5 and 7.6. Figure 7.5 shows the results for 2- and 3-dimensional sparse arrays, and Figure 7.6 for 4- and 8-dimensional sparse arrays.

The performances of PTCS and BESS are not affected by the different sparsity levels in general. The sparsity, σ, increases when the occupancy ratio, ρ, decreases. Recall that ρ is defined as the ratio of the number of non-zero elements to the product of the cardinalities of an MDSA. When the occupancy ratio is very low, the number of non-zero array elements is far less than the product of the cardinalities (or the array space) of the MDSA. The algorithms for sub-array retrievals in PTCS and BESS are based on either traversing the trie in the case of PTCS, or scanning serially the vector arrays in the case of BESS. PTCS and BESS store only those non-zero array elements in their data structures. Thus, their performances of sub-array retrieval are mainly determined by the number of non-zero elements and affected little by the sparsity. Sub-array retrieval in xCRS and BxCRS are very efficient. However, when the sparsity becomes extremely high, e.g., $\sigma = 99.9\%$, their performances tend to get worse for higher dimensions. The reason behind this is that the chances of an entire row having very few or no valid elements increases rapidly with the growing sparsity. This results in the scan of more rows to retrieve the wanted elements. By tuning the *hybrid ratio* α, ($\alpha = \frac{\lceil k/2 \rceil + 1}{k}$ and $r = \alpha k$ in the results shown here), the Hybrid outperformed the other methods for sub-array retrievals considerably on higher dimensions at all sparsity level.

7.2.1.4 Performance of Aggregation

Given the specifics in Chapter 4, for implementing multi-dimensional aggregation, we compute $k - 1$ aggregations in one scan of the data. The total time, excluding the time to write the results, to compute $k - 1$ aggregations were recorded. Some of the results are shown in Figures 7.7, 7.8, and 7.9 for comparison. Note that we did not include Hybrid for sparse matrices in Figure 7.7, since Hybrid method becomes either BESS or xCRS in such cases. The results show that for lower dimensions, such as 2, 3 and 4, all the schemes except PTCS perform within a close range. When the dimensionality increases to 8, the performances of xCRS and BxCRS become worse, while BESS and Hybrid perform the best. The aggregation time increases linearly with the growing sparsity for all the schemes on all dimensionalities (see Figure 7.9). In this case, BESS, Hybrid and PTCS are less affected by the sparsity compared with xCRS and BxCRS. The aggregation time of PTCS is slower than the other schemes on lower dimensions, but is stable with the varying dimensionality

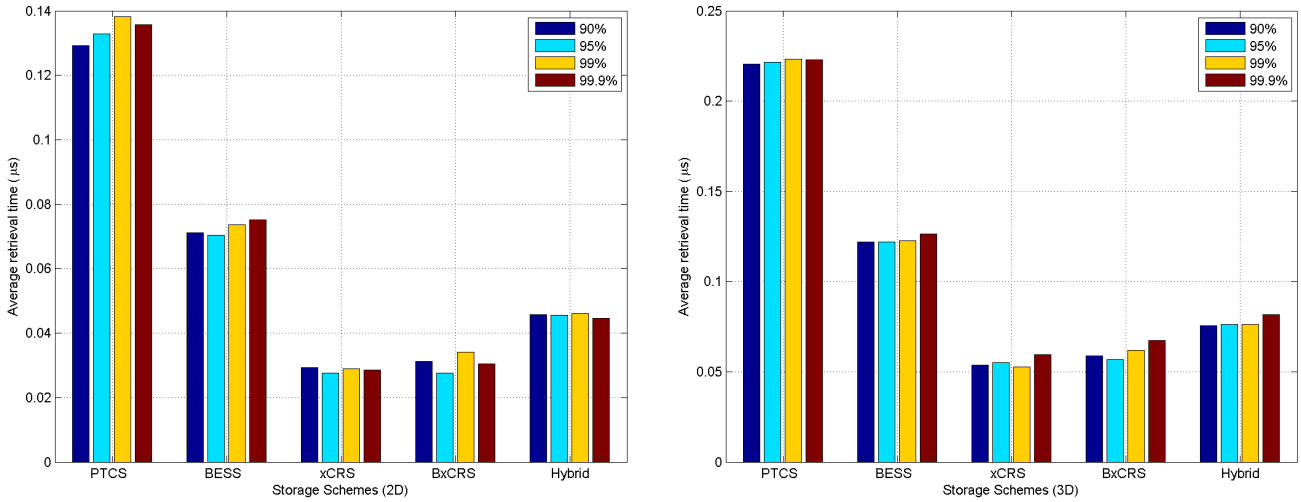


Figure 7.5: The average sub-array retrieval time of various schemes for MDSAs with $k = 2$ (left) and $k = 3$ (right).

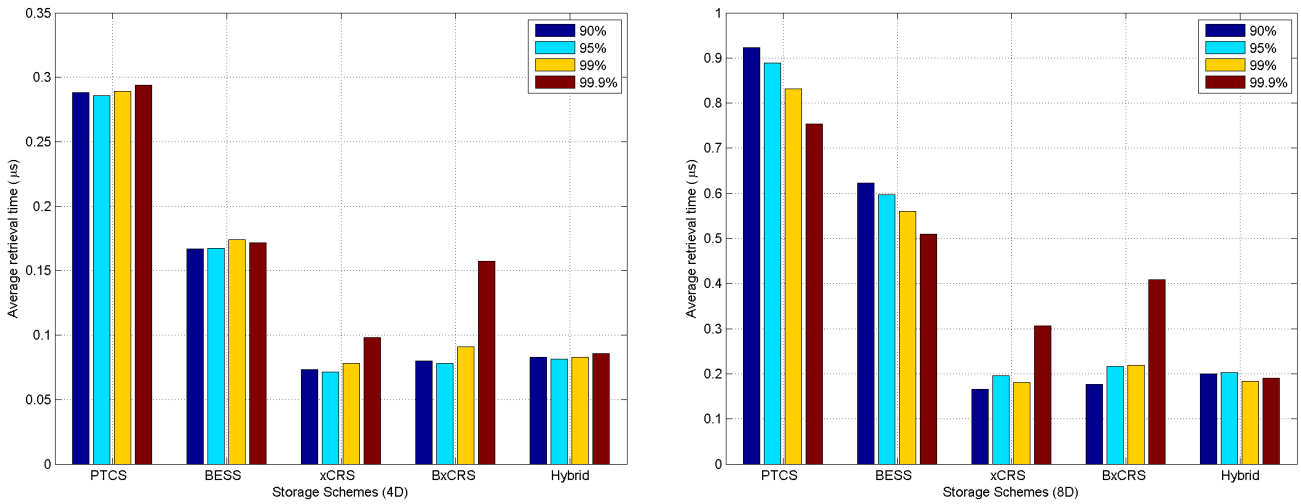


Figure 7.6: The average sub-array retrieval time of various schemes for MDSAs with $k = 4$ (left) and $k = 8$ (right).

and sparsity. The sudden decreases of aggregation time in Figures 7.8 and 7.9 for 8-dimensional sparse arrays are mostly due to the varying sparsity of the aggregated results, which are also multi-dimensional arrays with certain sparsities.

7.2.1.5 Performance of Computing the CUBE Using PTCS and BESS

We present in this section the experimental results on the computation of the cube using PTCS and BESS. PTCS and BESS share the same approach in the encoding of the dimensional data of an MDSA. The difference is that they store the resulting key-value pairs using different data structures. As a result, the key-value pairs need to be sorted for BESS, while a PATRICIA trie needs to be constructed for PTCS to get the data sorted

when computing the cube. Figures 7.10 and 7.11 show the total time to compute the cube for MDSAs with $k = 2, 3, 4, 8$, and $\sigma = 90\%$. The BESS approach outperformed the PTCS by 3 to 6 times, which showed that constructing a PATRICIA trie is computationally more expensive than sorting the BESS data in the cube computation. Furthermore, PTCS has high demand for the runtime memory during the computation of the cube. This is shown in the Figure 7.11 for the case of $k = 8$, where we were unable to finish the computation due to the lack of memory.

7.2.2 Results on CPU+GPU Co-Processing

The results for CPU+GPU co-processing of the two selected array operations, and the *cube* computation are presented in this section. The two selected array operations are large scale random array element access and sub-array retrieval. For each array operation, the CPU+GPU co-processing is implemented for three storage schemes, xCRS, Hybrid and BESS, respectively. Due to the similarity of xCRS and BxCRS, the implementation for BxCRS is omitted. Compared with the CPU only processing of these array operations, a minimum of 2, up to over 20 times of speed-up is observed. The CPU+GPU co-processing of the cube computation is implemented using only BESS. A minimum of 5 times of speed-up is achieved for the co-processing compared with the CPU only processing.

7.2.2.1 Performance of Large Scale Random Element Access

For BESS, *p-ary search* was used to access multiple array elements in the combined CPU+GPU implementation. Recall that large scale random element access refers to accessing multiple array elements in a single operation. Compared with the CPU only processing, of large scale random element access, our implementation of CPU+GPU co-processing using BESS achieved an average of 2.3 times speed-up for different dimensionalities and sparsities. The time to transfer the input data (the data to be searched) and the results between the host and the device were included in the timing for all the GPU implementations of large scale random element access. Figure 7.12 shows the results for 4- and 8-dimensional sparse arrays. For the implementation of p-ary search discussed in Chapter 5, each CUDA block searches for a single element, and the threads within the block cooperate to locate the element. The number of blocks was chosen as the number of all the elements being searched, and the number of threads within a block as 32 in the CUDA *kernel* configuration. Note that 32 threads per block performs better than the other alternatives such as 64, 128, 256, and so on, in the experiment. When the number of blocks is large (in millions in our experiment), increasing the number of threads in a block does not improve the performance, since each block has limited resources shared among the threads.

In the CPU+GPU implementation of large scale random element access using xCRS, one thread was assigned to each element to be accessed. If there are N elements to be searched, and the number of threads in a CUDA block is $BLOCKSIZE$, the number of blocks we need in the kernel configuration is at least $(N+BLOCKSIZE-1)/BLOCKSIZE$. Each thread finds the range of the corresponding row of the element being searched. A serial scan of the elements within the row is then carried out to determine if the searched element exists or not. Figure 7.13 shows the results of this implementation for 4- and 8-dimensional arrays with 4 different sparsities. The average search time is at least 2 times faster than the CPU only processing for 4-dimensional sparse arrays, and this is also true for 8-dimensional sparse arrays with $\sigma \leq 95\%$. When $\sigma > 95\%$, the average GPU search time gets worse due to the increase in the number of unoccupied rows. Hence increased time to determine the range of a row in such cases, and other threads in the same block end up idling even the elements assigned to them do not exist. On the contrary, CPU processing can take the advantage of increased number of empty rows.

Figure 7.14 shows the results for CPU+GPU co-processing of large scale random element access using Hybrid. We used the same approach as it was in xCRS in this implementation. The average speed-up for Hybrid using GPU is approximately 2 except for a few cases. The slower average time in the case of $\sigma = 90\%$ is due to the increased number of elements within a Hybrid block. We can improve the performance for lower sparsities by using binary search within the block. Another feasible method to implement the same process for Hybrid is to use the p-ary search following the implementation in BESS.

7.2.2.2 Performance of Sub-Array Retrieval

In Figures 7.15, 7.16 and 7.17, the average time of the sub-array retrieval using CPU+GPU was compared with that of using only CPU, for each of the methods considered. The results in these figures are for retrieving three sub-arrays within each data set. The details of these three sub-arrays were given in Section 7.2.1. The average retrieval time is then obtained by dividing the total time by the number of elements retrieved. The time to transfer the results from the device to the host is not included in the timing. The speed up of BESS ranges from 2 to 10 times for differing dimensionality. For xCRS, the speed up is much higher, and is between 20 to 60 times. For Hybrid, the speed up is about 16 times for $k = 4$, and only 2.5 for $k = 8$, with improved performance for higher sparsity of $\sigma = 95\%$.

The problem we experienced in sub-array retrieval using GPU was the high demand for GPU global memory space to store the results temporarily. This is especially the case when the retrieved sub-array is considerably large. Since the dynamic memory allocation is inefficient on GPUs, memory is pre-allocated in the implementations. The size of the memory needs to be allocated differs for different storage schemes. For xCRS, the size

is determined by the product of the maximum number of rows and the size of each row. Similarly, for Hybrid, we determine the required memory size according to the maximum number of blocks and the size of each block. The size, for BESS, is the maximum number of elements to be retrieved. For very large sub-arrays, it is not always possible to pre-allocate the required memory on GPUs. However, this could be solved by retrieving the sub-array in smaller blocks.

7.2.2.3 Performance of Computing the CUBE Using BESS

Using CPU+GPU co-processing, the cube operator was computed for all the experimental data sets, with different sparsities and dimensionalities. The results are compared with the ones using CPU only processing. An approximate of 5 to 8 times of speed-up is obtained. Figures 7.18 and 7.19 show the results for MDSAs with $k = 4, 8$, and $\sigma = 90\%, 95\%$. The time to write the output file was not included in the timing of the cube computing. The much faster cube computation of CPU+GPU co-processing is as a result of accelerated attribute resetting and sorting operations using GPUs. In Figures 7.20 and 7.21, the times in the total time to compute the cube were compared for CPU and CPU+GPU processing. The time labeled as 'sort' in these figures is the sum of the time for resetting the attribute and sorting. For each group of bars, the left bar is of CPU only processing, and the right side one is of CPU+GPU processing. Although a speed up of over 20 times was achieved for the sort and resetting attribute operations, the overall speed up is only 5 to 8 times due to the serial processing part, the aggregation operation. This is clearly shown in the Figures 7.20 and 7.21, where the total cube computation time is dominated by the time of computing the aggregations in the CPU+GPU processing. The same reason also applies to the slow growth rate of the CPU+GPU time in the Figures 7.18 and 7.19. Compared with the highly efficient sort algorithm in the Thrust library, our CPU implementation of the sort operation is not optimized.

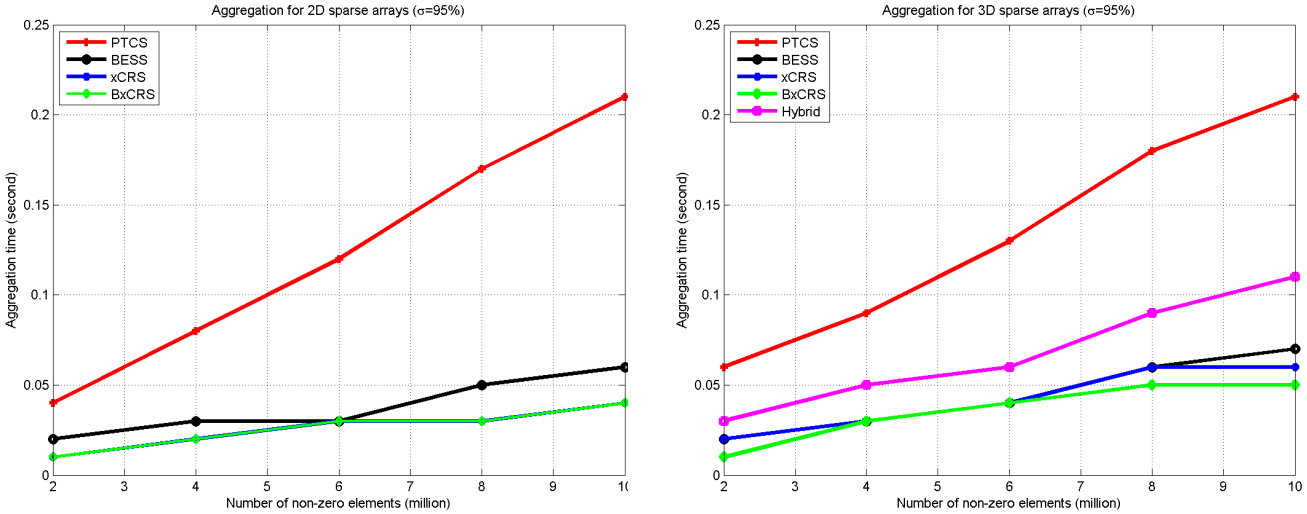


Figure 7.7: The multi-dimensional aggregation time of various schemes for MDSAs with $\sigma = 95\%$, $k = 2$ (left) and $k = 3$ (right).

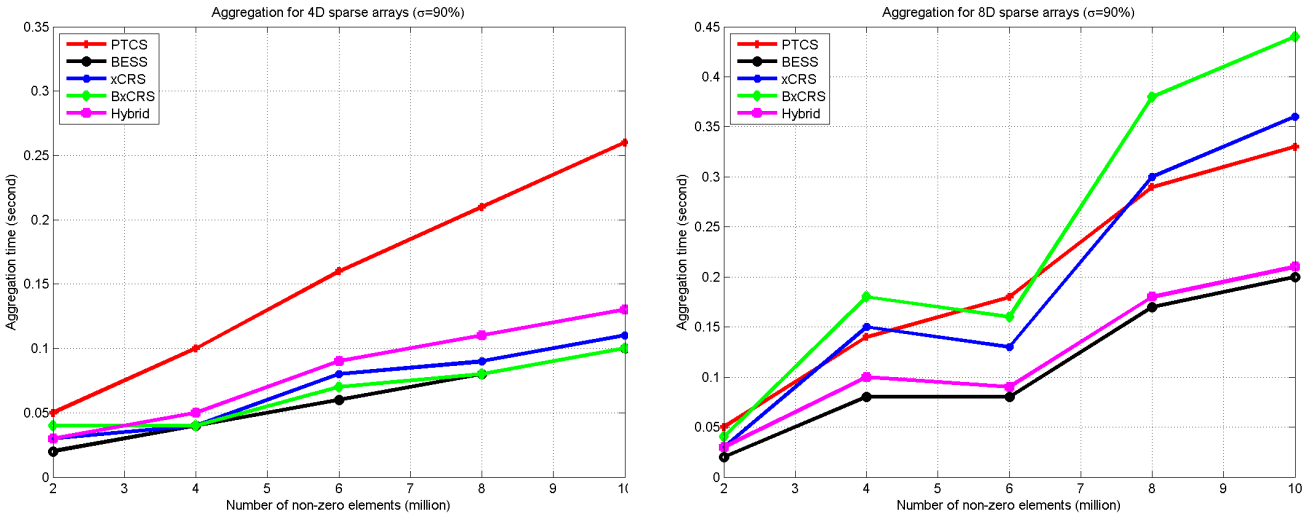


Figure 7.8: The multi-dimensional aggregation time of various schemes for MDSAs with $\sigma = 90\%$, $k = 4$ (left) and $k = 8$ (right).

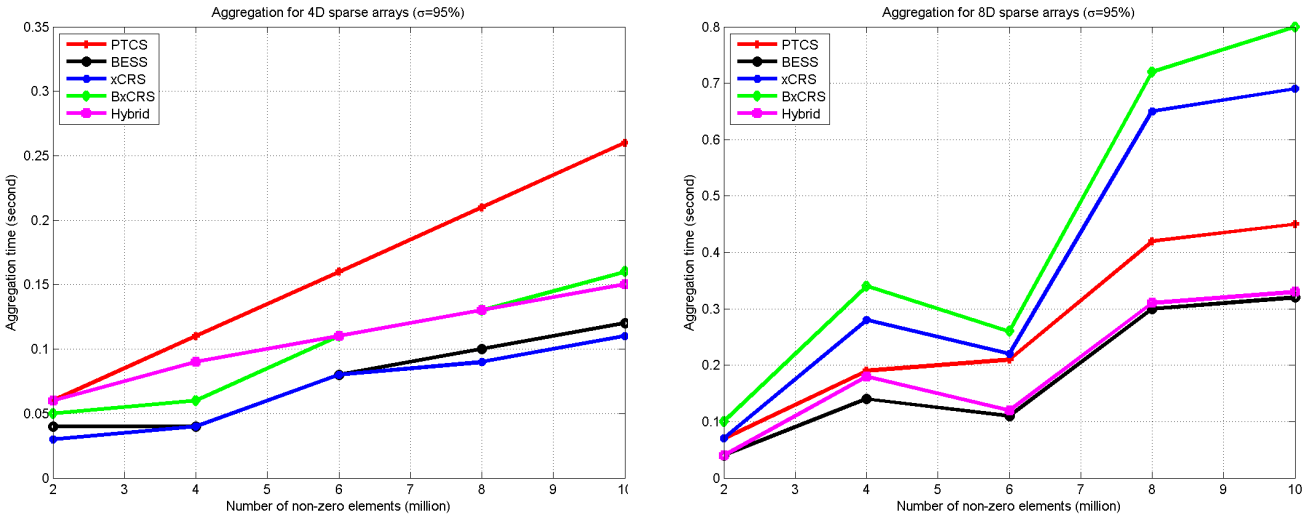


Figure 7.9: The multi-dimensional aggregation time of various schemes for MDSAs with $\sigma = 95\%$, $k = 4$ (left) and $k = 8$ (right).

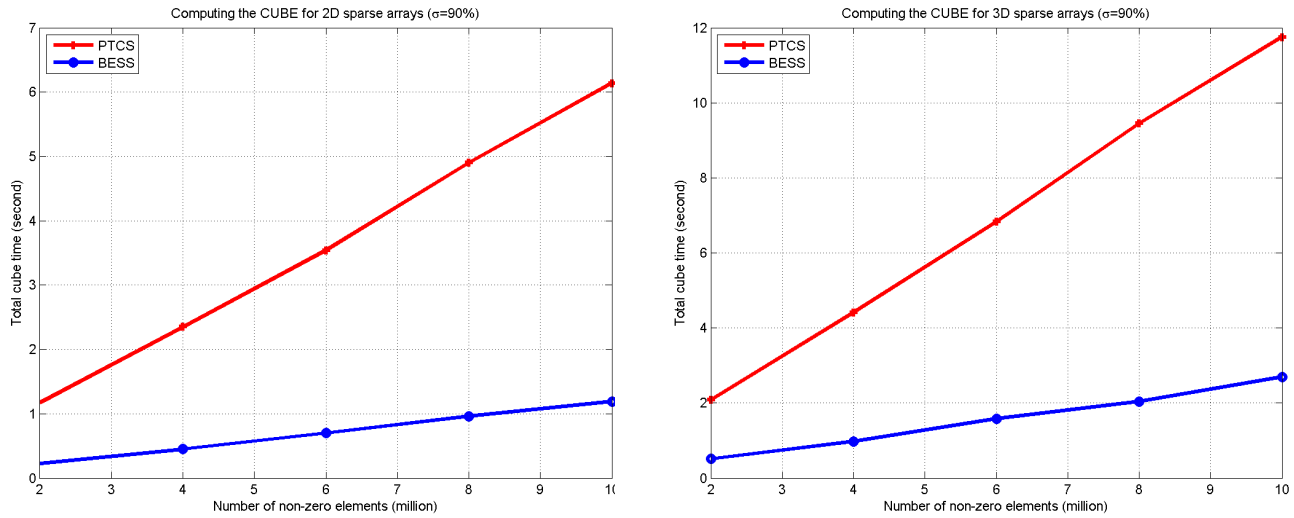


Figure 7.10: The time for computing the cube using PTCS and BESS for MDSAs with $\sigma = 90\%$, $k = 2$ (left) and $k = 3$ (right).

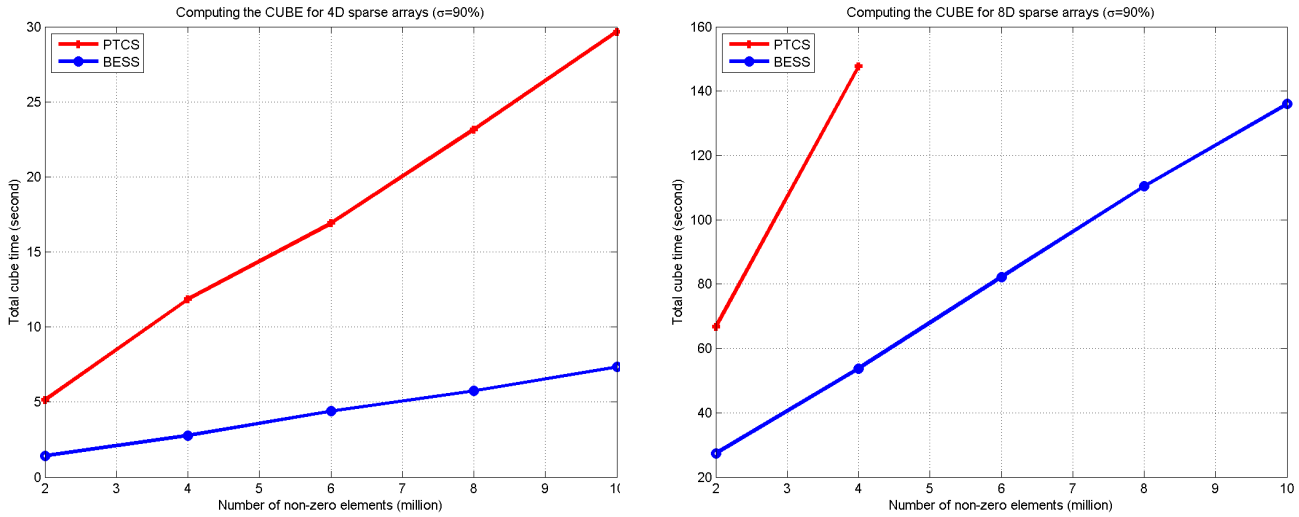


Figure 7.11: The time for computing the cube using PTCS and BESS for MDSAs with $\sigma = 90\%$, $k = 4$ (left) and $k = 8$ (right).

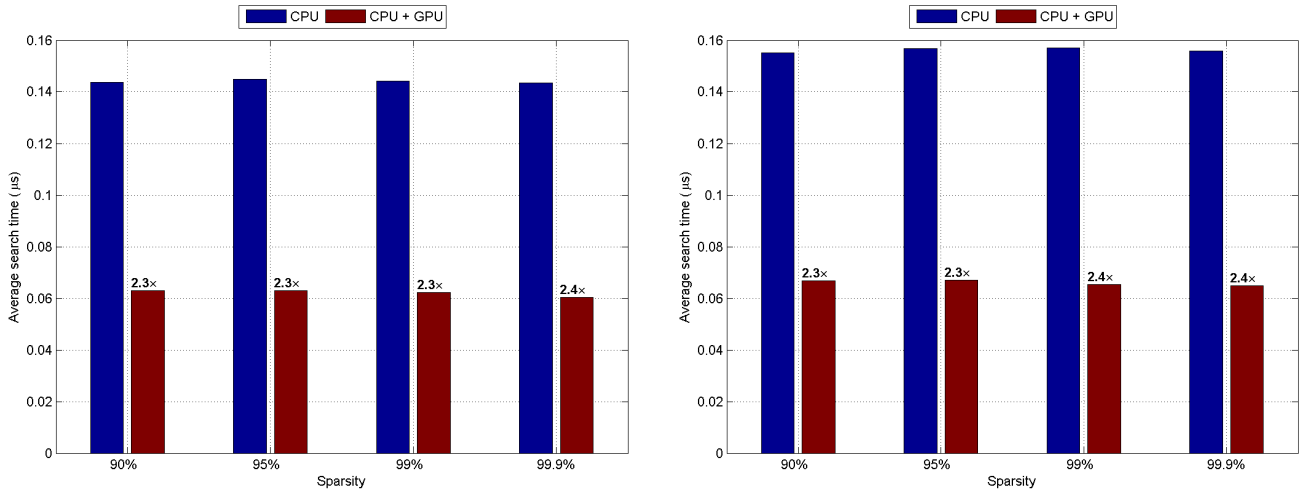


Figure 7.12: The average CPU+GPU random element access time using BESS for MDSAs with $k = 4$ (left) and $k = 8$ (right).

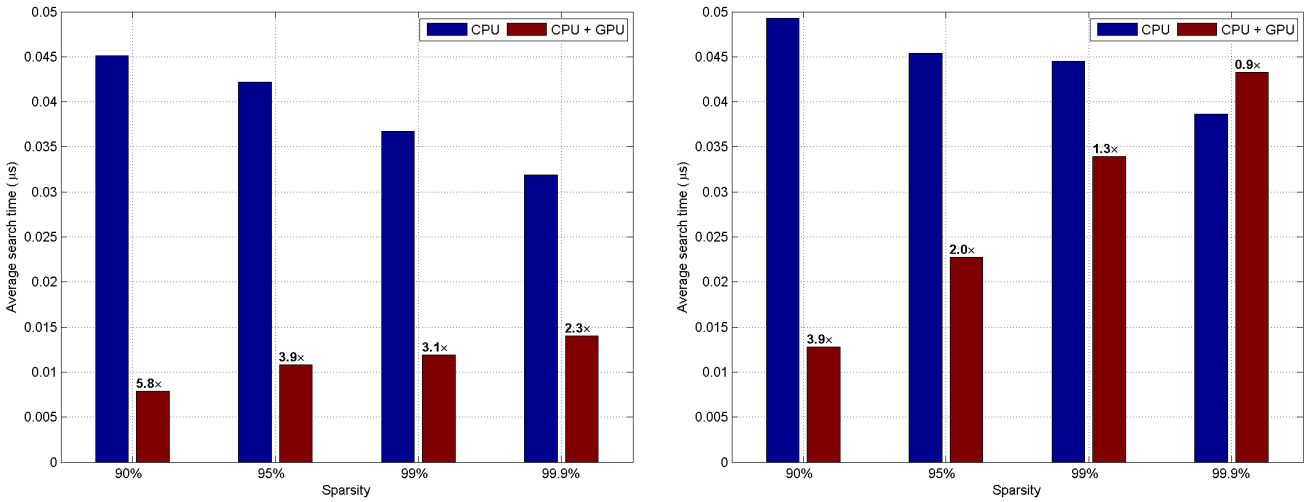


Figure 7.13: The average CPU+GPU random element access time using xCRS for MDSAs with $k = 4$ (left) and $k = 8$ (right).

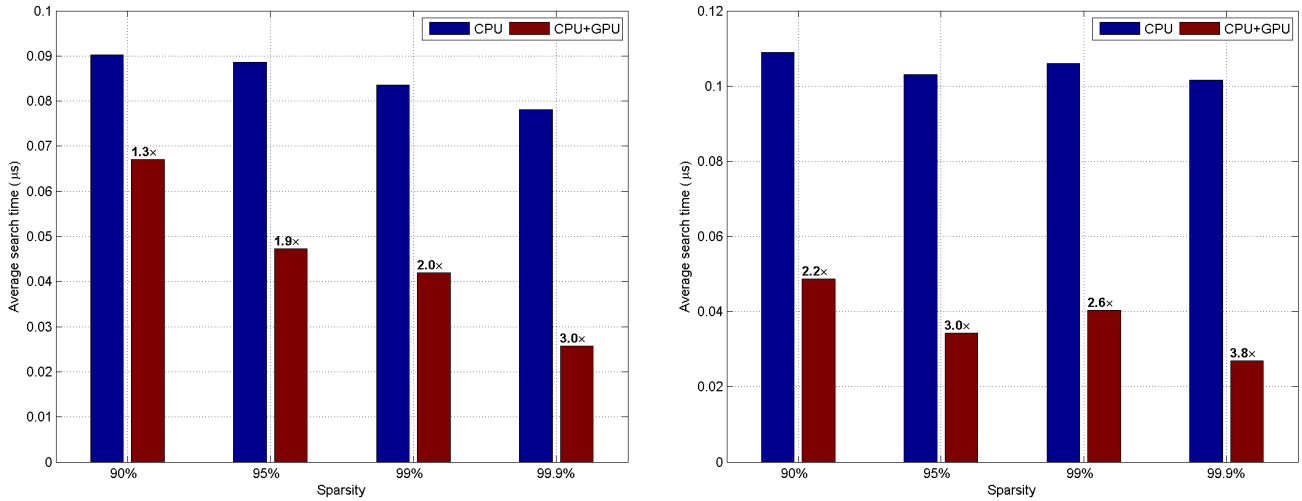


Figure 7.14: The average CPU+GPU random element access time using Hybrid for MDSAs with $k = 4$ (left) and $k = 8$ (right).

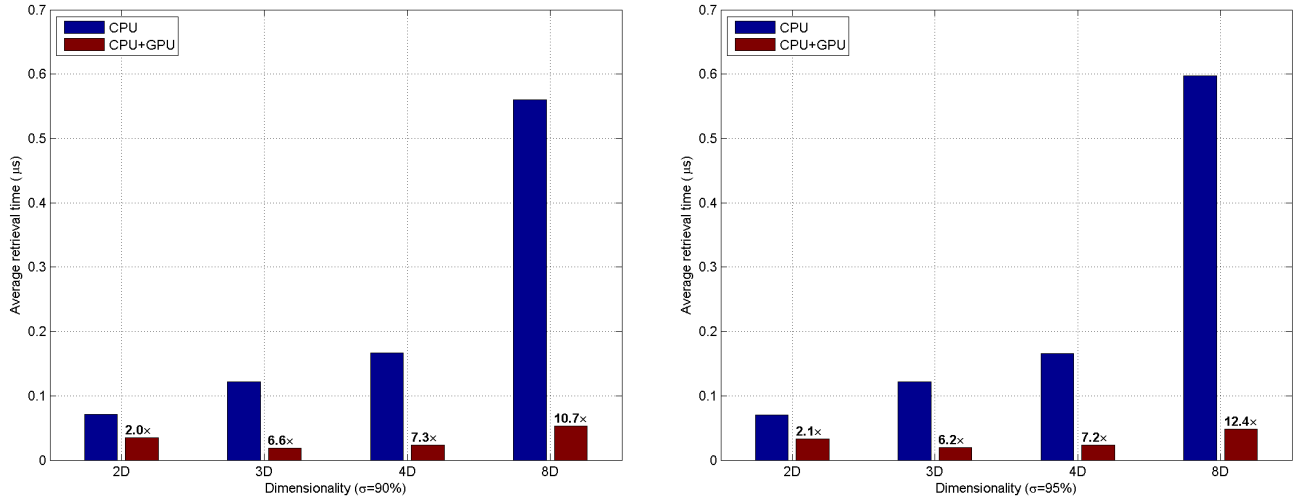


Figure 7.15: The average CPU+GPU sub-array retrieval time using BESS for MDSAs with $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

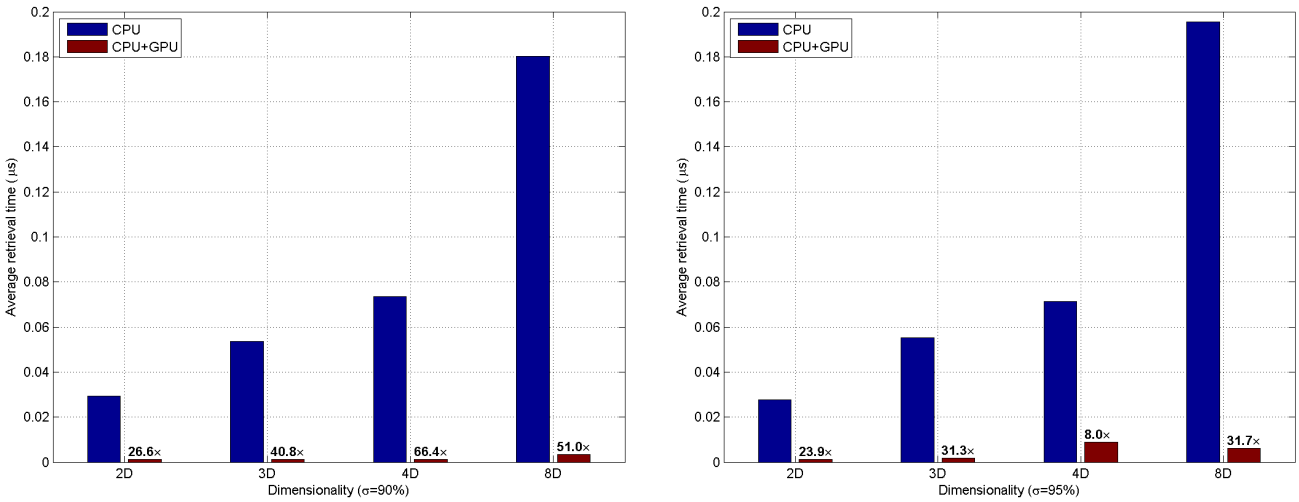


Figure 7.16: The average CPU+GPU sub-array retrieval time using xCRS for MDSAs with $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

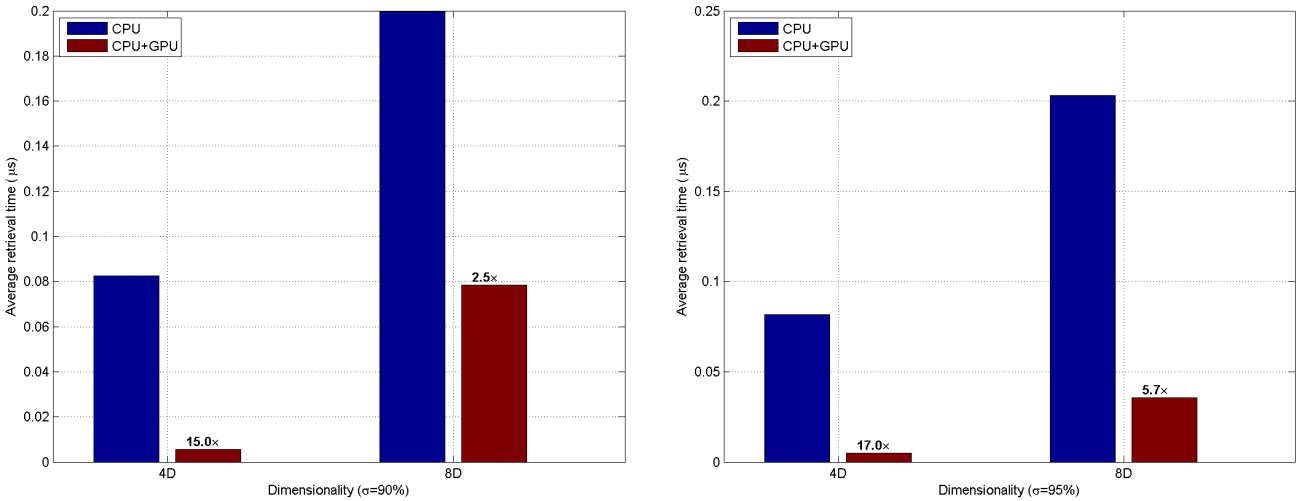


Figure 7.17: The average CPU+GPU sub-array retrieval time using Hybrid for MDSAs with $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

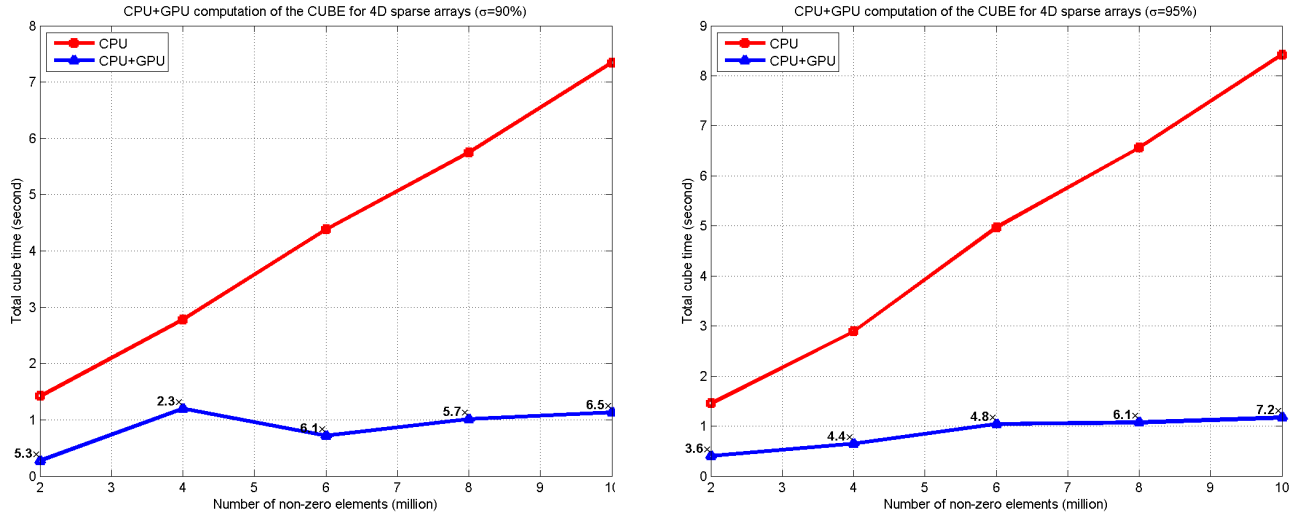


Figure 7.18: The time for computing the cube using BESS for MDSAs with $k = 4$, $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

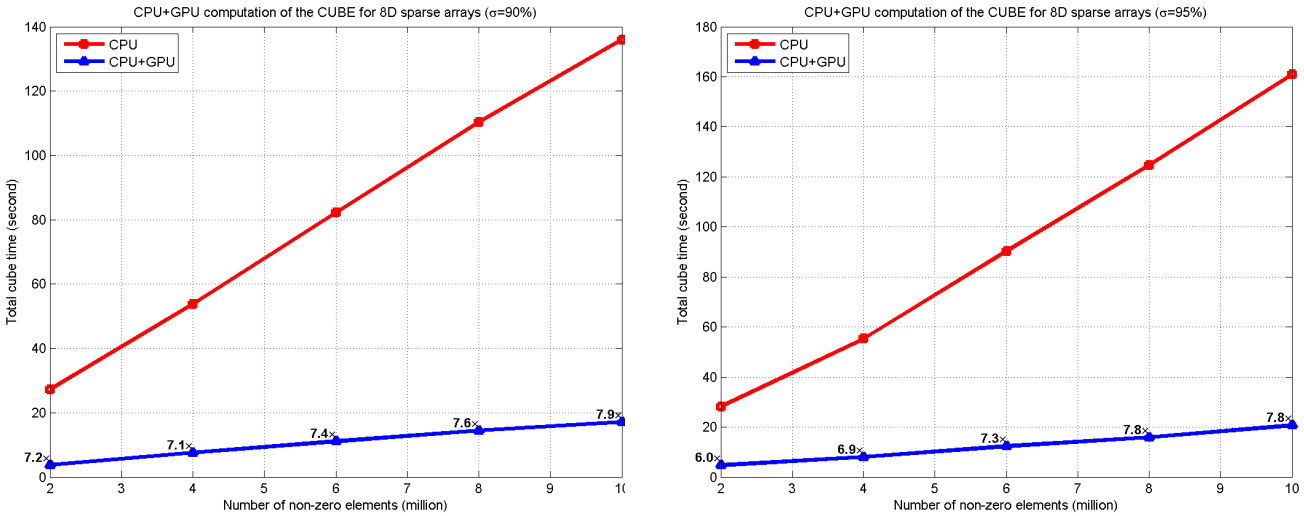


Figure 7.19: The time for computing the cube using BESS for MDSAs with $k = 8$, $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

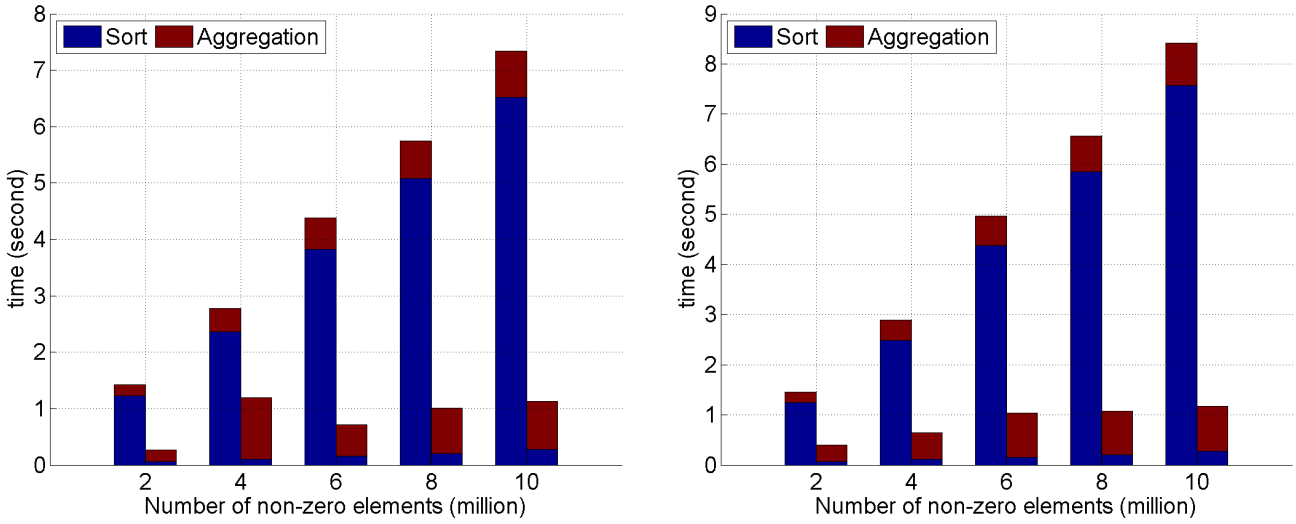


Figure 7.20: The CPU, GPU sort (includes attribute resetting) and aggregation time for computing the cube using BESS for MDSAs with $k = 4$, $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

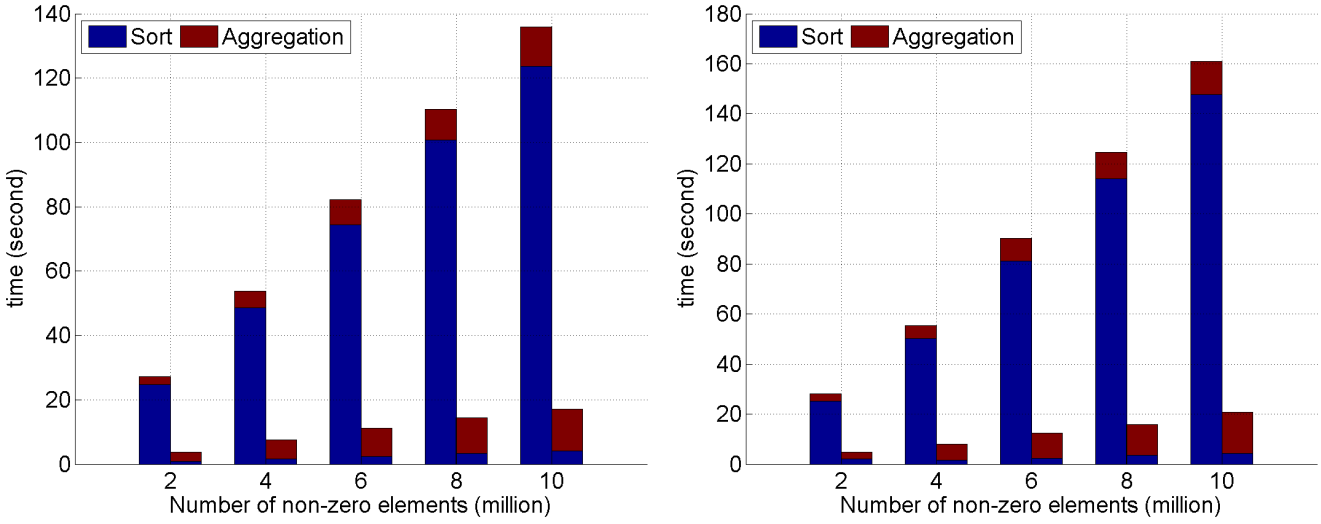


Figure 7.21: The CPU, GPU sort (includes attribute resetting) and aggregation time for computing the cube using BESS for MDSAs with $k = 8$, $\sigma = 90\%$ (left) and $\sigma = 95\%$ (right).

Chapter 8

Conclusion

8.1 Main Objectives

Our primary objective of this dissertation was to develop efficient storage schemes for multi-dimensional sparse arrays which have their significance in data warehousing and On-Line Analytical Processing (OLAP) applications. The data in data warehousing and multi-dimensional OLAP (MOLAP) is characterized by large volume, high dimensionality, and high sparsity. A multi-dimensional array is a desirable data structure to represent the data in these systems. By designing storage schemes for multi-dimensional sparse arrays, we aim at handling the sparsity of the multi-dimensional array structure in an efficient manner, so that the performance improvements can be achieved in various array operations. The basic principles we followed in the design of new storage schemes are storing only those valid non-zero array elements and applying *dimensional mappings*. We proposed and accomplished evaluating, comparing the storage schemes including the new ones and a known method, on a number of array operations using a set of synthetic large, sparse data sets.

The second objective of this dissertation was to parallelize selected array operations by utilizing GPUs. The multi-dimensional sparse arrays in this regard were to be represented using some of the new storage schemes developed in this research, as well as an existing method, BESS. GPUs provide massive computing power and high memory bandwidth. We explored the feasibility and challenges of applying the GPUs to the problems in data warehousing and OLAP applications. In particular, we proposed and accomplished, as part of the second goal, applying GPUs to accelerate the computation of the *cube*, which is an important problem in data warehousing and OLAP.

8.2 Main Contributions

In this dissertation, we designed, implemented and evaluated four efficient schemes to store multi-dimensional sparse arrays, and utilized massive parallelism of GPUs for some data warehousing operations. In the design of the new storage schemes, we considered extending existing sparse matrix storage schemes to higher dimensions and developed some new approaches that combine different storage schemes. Four new schemes were introduced. These are:

1. The extended compressed row storage (xCRS) which extends CRS method for sparse matrix storage to sparse arrays of higher dimensions;
2. The bit encoded xCRS (BxCRS) which optimizes the storage utilization of xCRS by applying data compression methods with run length encoding;
3. Hybrid approach (Hybrid) combines the xCRS and a known method, BESS;
4. The Patricia trie compressed storage (PTCS) which uses PATRICIA trie to store the valid non-zero array elements.

These schemes were evaluated on two basic array operations to reflect their data access efficiency. Storage utilization was computed for each scheme and measured in the implementations as well. The results showed that xCRS has the best data access efficiency among all the schemes. Its storage overhead increases linearly with respect to the sparsity of the MDSA. BxCRS has the similar data access efficiency as xCRS, and is able to reduce the storage overhead of xCRS considerably for highly sparse arrays. We observed in our experimental results that both xCRS and BxCRS perform well for MDSAs with lower sparsity, but turn less efficient when the sparsity becomes very high ($\sigma > 99\%$). Hybrid achieves the best control of the balance between storage utilization and data access efficiency. Both storage utilization and data access efficiency of PTCS are worse than the other methods. However, its performance is least affected by the sparsity, and conveniently supports *update* operations. The details of *updates* are not addressed in this dissertation.

We further evaluated the storage schemes on multi-dimensional aggregation. The experimental results show that the known method, BESS outperforms the new methods. The performance of Hybrid method is very close to that of BESS. XCRS and BxCRS perform well on lower dimensionality, but get worse for higher dimensionality and sparsity. The performances of PTCS and BESS are stable with respect to the changing dimensionality and sparsity. We also studied computing the *cube* operator; a special problem of multi-dimensional aggregation, using PTCS and BESS. In this regard, we used sort-based method for the cube compution. We observed

that sorting BESS data is more efficient than constructing PTCS. Hence, we conclude that among the schemes we evaluated, the BESS is the most efficient storage scheme for the problem of computing the *cube* for MDSA.

We explored using GPUs as the co-processor for CPU to accelerate the two basic array operations, namely large scale random element access and sub-array retrieval. For large scale random element access, a minimum of 2 times speed up was obtained for the three schemes of xCRS, Hybrid and BESS (including the data transfer time between the host and the device), chosen to represent the data in the CPU+GPU co-processing. The speed ups for sub-array retrieval were $2-10$, 16, $20-60$ times respectively for BESS, Hybrid and xCRS. This excludes the data transfer time between the host and device. The challenges of using GPUs for these two operations are the large GPU global memory space requirement, and difficulty in utilizing the high GPU memory bandwidth fully.

We carried our work on computing the cube using BESS further to utilize GPUs for selected parts; namely for sorting and attribute resetting. We applied the highly optimized sorting algorithm in the CUDA Thrust template library for the sort operation, and implemented the attribute resetting process based on its CPU version. As a result, the CPU+GPU co-processing accelerated the cube computation by $5-8$ times, compared with the CPU only version. More specifically, the cube computation time in the CPU+GPU co-processing is dominated by the serial processing part, which is the aggregation. We did not implement the aggregation operation on GPUs taking the following reasons into considerations. Firstly, the aggregation results are also multi-dimensional array data. Hence, storing the results temporarily on GPUs demand large amount of global memory space. Secondly, the time to transfer the results from device to the host becomes dominant for higher dimensionality, such as $k = 8$.

8.3 Future Work

Extending the current work to very high dimensional data set when the data can not fit into main memory is a research issue to be pursued in the future. An optimal chunking scheme can be applied to the multi-dimensional sparse array. Within each chunk, the valid non-zero array elements are represented using some suitable storage schemes. Applying a storage scheme to represent the MDSA data for problems such as nearest neighbor queries, top-k query, and other related problems will also be addressed. Exploring hybrid computing on heterogeneous systems, or different parallel computing models is another interesting research issue to be pursued.

Appendix A

Additional Algorithms

A.1 The Sub-Array Retrieval Algorithm in XCRS

Algorithm 12: xcrsSubArrayRetrieval($val, cind, rptr, l, h$)

Input: xCRS arrays $val[..]$, $cind[..]$ and $rptr[..]$, an array $dim[0..k-1]$ contains the cardinalities for k dimensions, a starting index $l[0..k-1]$ and an ending index $h[0..k-1]$.

Output: A list of non-zero elements and their indexes in the sub-array defined by $l[..]$ and $h[..]$.

begin

`/* Find the row offsets of` $l[..]$ `and` $h[..]$ `*/`

$lra \leftarrow computeRowOffset(l[1..k-1])$

$hra \leftarrow computeRowOffset(h[1..k-1])$

$tmp \leftarrow h[1] - l[1] + 1$

$ra \leftarrow lra$, $i \leftarrow 0$

while $ra \leq hra$ **do**

$ind[1..n-1] \leftarrow computeIndex(ra)$

`/* If` $ind[i] \geq l[i]$, $2 \leq i \leq k-1$ `then` $ind[2..k-1] \geq l[2..k-1]$. `*/`

if $ind[2..n-1] \geq l[2..n-1]$ *and* $ind[2..n-1] \leq h[2..n-1]$ **then**

for $j = 0$ *to* tmp **do**

`/* Find the starting position of the row` ra `in` $rptr[]$ `*/`

$p \leftarrow rptr[ra]$

if $p \neq -1$ **then**

Find the starting position q of the next row with non-zero elements

for $j = p$ *to* q **do**

if $cind[j] \geq l[0]$ *and* $cind[j] \leq h[0]$ **then**

Add the indices and data $val[j]$ to $subArrayBlk$

$ra \leftarrow ra + 1$

$i \leftarrow i + 1$

$ra \leftarrow lra + i * dim[1]$

return $subArrayBlk$

A.2 The Algorithm to Search the Array $compwrd$

The Algorithm 13 determines whether a row, given its row_offset, has non-zero values or not in the array $compwrd$ of BxCRS.

Algorithm 13: searchCompWrd($compwrd$, row_offset, $total_row_no$, no_reg_wrds)

Input: An array of compressed words $compwrd[..]$, a row offset row_offset, the total row number $total_row_no$, and the number of regular words no_reg_wrds.

Output: TRUE if row row_offset has non-zero values, FALSE if not.

begin

`/* The bit length of a word is` W `*/`

$m \leftarrow total_row_no \,/\, (W-1)$

$m0 \leftarrow row_offset \,/\, (W-1)$

$r \leftarrow row_offset \,\%\, (W-1)$

`/* Set the bit corresponds to the` row_offset `to 1 */`

$mask_r \leftarrow 1 << (W-2-r)$

Set fil_mask with the most significant bit 1 and 0 for the rest

Set fil_bit_mask with the second most significant bit 1 and 0 for the rest

if $row_offset < m \times (W-1)$ **then** Check within regular words

  if fil_mask & $compwrd[m0] = 0$ **then** The word is literal word

    if $compwrd[m0]$ & $mask_r \neq 0$ **then** The row has non-zero value

      return TRUE

    else

      return FALSE

  else

    if fil_bit_mask & $compwrd[m0] \neq 0$ **then** Onefill word

      return TRUE

    else Zerofill word

      return FALSE

else Check within active words

  Set $active_mask$ corresponds to row_offset

  if $active_mask$ & $compwrd[no_reg_wrds] \neq 0$ **then**

    return TRUE

  else

    return FALSE

A.3 The Sub-Array Retrieval Algorithm in PTCS

Algorithm 14: ptcsSubArrayRetrieval(T, l, h)

Input: A PATRICIA trie T, the starting and ending indexes, $l[0..k-1]$ and $h[0..k-1]$ of a sub-array in the given sparse array.

Output: A list of non-zero elements and their indexes in the sub-array defined by $l[..]$ and $h[..]$.

begin

$root \leftarrow T.rlink$

$pt \leftarrow T$, $tmp \leftarrow root$

`/* Find the PTCS keys for indexes` $l[..]$ `and` $h[..]$ `*/`

$lKey \leftarrow ptcsKey(l[..])$

$hKey \leftarrow ptcsKey(h[..])$

$pos \leftarrow$ *firstDifferentBit*$(lKey, hKey)$

while $pt.bit_pos < tmp.bit_pos$ *and* $tmp.bit_pos \leq pos$ **do**

$pt \leftarrow tmp$

if *getBit*$(lKey, tmp.bit_pos) = 1$ **then**

$tmp \leftarrow tmp.rlink$

else

$tmp \leftarrow tmp.llink$

if $pt.bit_pos < pos$ **then**

`/* Traverse the left or right sub-trie */`

if *getBit*$(lKey, pt.bit_pos) = 1$ **then**

$sub_root \leftarrow pt.rlink$

else

$sub_root \leftarrow pt.llink$

$subArrayBlk \leftarrow$ traverseTrie(sub_root, l, h)

if $pt.bit_pos = pos$ **then**

`/* Traverse the left and right subtrie */`

$subArrayBlk \leftarrow$ traverseTrie$(pt.llink, l, h)$

$subArrayBlk \leftarrow$ traverseTrie$(pt.rlink, l, h)$

return $subArrayBlk$

Bibliography

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multi-dimensional aggregates. In *Proceedings of the 22nd VLDB Conference*, pages 506–521, Mumbai (Bombay), India, 1996.

[2] R. Barrett, M. Berry, T. F. Chan, J. Dammel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 1994.

[3] R. F. Boisvert, R. Pozo, and K. A. Remington. The matrix market exchange formats: Initial design. Technical report, National Institute of Standards and Technology, 1996.

[4] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barret, and J. J. Dongarra. Matrix market: A web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software: assessment and enhancement*, pages 125–137, London, UK, 1997. Chapman & Hall, Ltd.

[5] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, Mar. 1997.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 3rd edition, 2009.

[7] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing data cube. *Distributed and Parallel Databases*, 11:181–201, 2002.

[8] F. Dehne and H. Zaboli. Parallel construction of data cubes on multi-core multi-disk platforms. *Parallel Processing Letters*, 23(1), 2013.

[9] L. Devroye. A note on the height of binary search trees. *Journal of the Association for Computing Machinery*, 33(1):489–498, 1986.

[10] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, Sept. 2010.

[11] W. Fang, K. K. Lau, and M. L. et al. Parallel data mining on graphics processors. Technical Report HKUST-CS08-07, Hong Kong University of Science and Technology, Oct 2008.

[12] S. Goil and A. Choudhary. Sparse data storage of multi-dimensional data for OLAP and data mining. Technical Report CPDC-TR-9801-005, Center for Parallel and Distributed Computing, Northwestern University, 1997.

[13] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUterasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, Chicago, IL, USA, 2006. ACM.

[14] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Survey*, 25(2):73–170, 1993.

[15] G. Graefe. Implementing sorting in database systems. *ACM Computing Survey.*, 38(3), Sept. 2006.

[16] J. Gray, S. Chaudhuri, and A. B. et al. Data cube, a relational aggregation operator generalizing group-by, cross-tables and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[17] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM-SIGMOD Conference*, pages 205–216, Montreal, Canada, 1996.

[18] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[19] B. He and Q. Luo. Cache-oblivious databases: Limitations and opportunities. *ACM Trans. Database Syst.*, 33(2):8:1–8:42, June 2008.

[20] J. Hoberock and N. Bell. Thrust: A parallel template library. http://thrust.github.io/, 2010. Version 1.7.0.

[21] W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[22] W. Hwu, editor. *GPU Computing Gems*, chapter Large Scale GPU Search. Morgan Kaufmann Publishers, Wolftham, MA, 2012.

[23] Khronos Group. OpenCL: The open standard for parallel programming of heterogeneous systems. http://www.khronos.org. 2014-01-07.

[24] D. B. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[25] D. E. Knuth. *The art of computer programming*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

[26] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison Wesley, Reading, MA, 2nd edition, 1998.

[27] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, Saint-Malo, France, 2010. ACM.

[28] D. Merrill and A. S. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[29] D. R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *Journal of the Association for Computing Machinery*, 15(4):514–534, 1968.

[30] T. B. Nguyen, A. M. Tjoa, and R. Wagner. An object oriented multidimensional data model for OLAP. In *Proceedings of the First International Conference on Web-Age Information Management*, WAIM '00, pages 69–82, Shanghai, China, 2000. Springer-Verlag.

[31] NVIDIA CUDA. CUDA C best practices guide. https://developer.nvidia.com/category/zone/cuda-zone. 2013-07-12.

[32] NVIDIA CUDA. CUDA C programming guide. https://developer.nvidia.com/category/zone/cuda-zone. 2013-07-12.

[33] NVIDIA OpenCL. OpenCL programming guide for CUDAarchitecture. https://developer.nvidia.com/opencl. 2014-01-07.

[34] NVIDIA Tesla. NVIDIA tesla GPU computing processor ushers in the era of personal supercomputing. http://www.nvidia.com/object/IO_43499.html. 2013-07-12.

[35] OpenACC. OpenACC: Directives for accelerators. http://www.openacc-standard.org. 2014-01-07.

[36] J. D. Owens, D. Lubeke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[37] Y. Perl, A. Itai, and H. Avni. Interpolation search: a $\log \log N$ search. *Commun. ACM*, 21(7):550–553, July 1978.

[38] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 23rd VLDB Conference*, pages 116–125, Athens, Greece, 1997.

[39] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations. Technical Report CSRD-TR-1029, University of Illinois, Urbana, IL, 1990.

[40] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994.

[41] S. Samtani, M. K. Mohania, V. Kumar, and Y. Kambayashi. Recent advances and research problems in data warehousing. In *Proceedings of the Workshops on Data Warehousing and Data Mining: Advances in Database Technologies*, ER '98, pages 81–92, Singapore, 1998. Springer-Verlag.

[42] S. Sarawagi. Indexing OLAP data. *Data Engineering Bulletin*, 20:36–43, 1996.

[43] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Rome, Italy, 2009. IEEE Computer Society.

[44] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw Hill, 6th edition, 2010.

[45] TPC. Transaction processing performance council. http://www.tpc.org. 2013-11-21.

[46] P. Trancoso, D. Othonos, and A. Artemiou. Data parallel acceleration of decision support queries using cell/be and GPUs. In *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, pages 117–126, Ischia, Italy, 2009. ACM.

[47] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 13–13, Berkeley, CA, 2010. USENIX Association.

[48] H. Wang. Data warehouse operations on sparse multi-dimensional array storage. Wits University Honours Research Report, 2012.

[49] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2433–2442, Shanghai, China, 2012. IEEE Computer Society.

[50] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[51] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *ACM-SIGMOD International Conference on Management of Data*, pages 159–170, Tucson, AZ, USA, 1997.