The Art of Sparsity: Mastering High-Dimensional Tensor Storage

Bin Dong
Lawrence Berkeley National Laboratory
dbin@lbl.gov
ORCID: 0000-0002-0725-0833

Kesheng Wu
Lawrence Berkeley National Laboratory
kwu@lbl.gov
ORCID: 0000-0002-6907-3393

Suren Byna
The Ohio State University
byna.1@osu.edu
ORCID: 0000-0003-3048-3448

***Abstract*—Sparse tensors are prevalent in many applications. While numerous approaches have emerged to optimize the organization of sparse tensors, with the goal of reducing storage requirements and enhancing access performance, a comprehensive examination of the associated time and space complexities has been notably lacking. This study bridges this gap by conducting both theoretical and empirical investigations into various strategies for storing sparse tensors. Our major findings are as follows: (1) Linear address-based organization provides the best balance between storage size and access time; (2) Sparse high-dimensional tensor data can be transformed into lower-dimensional tensors, facilitating efficient storage and access; (3) In the absence of dimension transformation, tree-structured organizations offer compelling performance in low-dimensional tensors and exceptional performance in high-dimensional tensors.**

***Index Terms*—Sparse Tensor, High-dimensional Tensor, Organization, Time complexity, Space complexity, GCSC++, GCSR++, CSF, Compressed Sparse Fiber, TileDB, HDF5**

I. Introduction

Sparse tensors [1, 2] capture complex high-dimensional data with numerous zeroes aiming to conserve both storage space and computational resources. They play a pivotal role in diverse fields, including scientific simulations and machine learning, enabling the modeling of real-world phenomena with unparalleled accuracy. Their importance stems from their ability to handle vast and intricate datasets while preserving essential information, ultimately driving advancements in problem-solving across a multitude of applications.

Sparse tensors have been extensively studied from different perspectives, including sparse tensor algebra [3, 4], CPU access performance [5, 6], and sparse tensor organizations [7]. This work focuses on sparse tensor organizations for storage, which typically include in-memory representations [7–9] and external storage representations [10, 11]. We aim to address the following research questions in this work:

- What is the state of the art in the storage organizations for high-dimensional and sparse tensors? A storage organization represents the physical layout (or sometimes called materialization or storage format) of sparse tensors.
- What are the theoretical limits for time complexity and space complexity of these sparse tensor organizations? A theoretical analysis plays a fundamental role in comparing various sparse tensor organizations. It can provide insights into the inherent trade-offs between storage efficiency and computational performance.
- How to design an experimental system to verify theoretical analysis results for sparse tensor organizations?

To answer these questions, we conducted a comprehensive survey of recent advancements in data organization methods for sparse tensors. Through this survey, we identified five popular data organizations, including coordinate-based organization (COO) [10], linear-addresses-based organization (LINEAR) [10], Generalized Compressed Sparse Row/Column organization (GCSC++/GCSR++) [12, 13], and Compressed Sparse Fiber organization (CSF) [14, 15]. We analyzed the theoretical limits of these sparse tensor organizations in terms of their time and space complexities. Additionally, we selected three general sparsity patterns observed from real-world applications for our verification of these theoretical performance bounds. Subsequently, we designed and developed a general benchmark system to assess these sparse tensor organizations within these three general patterns. Our major findings include:

- The choice of storage organization for sparse tensors significantly impacts both read and write performance.
- The LINEAR organization, based on linear addressing, strikes the best balance between storage size and access performance for sparse high-dimensional tensors.
- GCSR++ and GCSC++ can transform high-dimensional and sparse tensors into lower-dimensional equivalents, thereby facilitating efficient storage and access.
- Tree-based organizations, exemplified by Compressed Sparse Fibers (CSF), offer an alternative approach to storing high-dimensional and sparse tensors without the need to transform coordinates into lower-dimensional ones.
- GCSC++ and GCSR++ can achieve better performance in organizing sparse tensors when their layouts are aligned with their preferred data access patterns.
- The Coordinate List (COO) organization can save time in building organizations (assuming the input data are coordinate-based as well). However, this time-saving advantage may be lost by the need to write large result files to disk or the high cost associated with accessing them.

II. Storage Organizations of Sparse Tensors

We explore five storage organizations for sparse tensors. Spatial hash [16] and R-tree [17] are not within the scope of this work, as they are primarily used to index blocks of points. Our focus is on general data organizations that can represent sparse points within these blocks when necessary. These organizations can also be applied when a sparse tensor is stored and accessed as a whole entity.

While general compression algorithms are effective for reducing data size, including sparse tensors, the fundamental

COO	LINEAR	Value
(0, 0, 1)	1	v_1
(0, 1, 1)	4	v_2
(0, 1, 2)	5	v_3
(2, 2, 1)	25	v_4
(2, 2, 2)	26	v_5

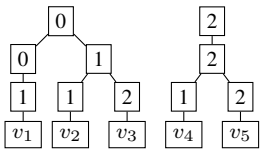
(a) COO and LINEAR

row_{ptr}: $0, 3, 5, 5$
col_{ind}: $0, 3, 4, 6, 7$
values: v_1, v_2, v_3, v_4, v_5

(b) GCSR++

col_{ptr}: $0, 3, 3, 3, 3, 5, 5, 5, 5, 5$
row_{ind}: $0, 0, 0, 2, 2$
values: v_1, v_2, v_3, v_4, v_5

(c) GCSC++



(d) CSF

Fig. 1: Representation of an example 3D sparse tensors with size $3 \times 3 \times 3$ in different organizations.

data organizations considered in this paper are orthogonal to these compression techniques. Common practice in the community, as observed in systems like TileDB [10] and HDF5 [18], is to choose a basic sparse organization first and then apply compression algorithms to further reduce data size. In addition to introducing the details of these sparse tensor organizations, we also analyze their time complexity and space complexity, as defined below:

- **Time Complexity** refers to the time required by corresponding algorithms to construct the organizations for sparse tensors with n non-empty cells.
- **Space Complexity** refers to the storage space necessary for storing the sparse tensors. We estimate the space size required (in units of the index type's size) to accommodate n points in sparse tensors. The analysis of space complexity does not account for the storage of values, as their size remains constant across all organizations. We also disregard metadata [19] such as tensor size, as these values remain consistent and are small enough to be negligible.

A. Coordinate COO Organization–The baseline

The input of our sparse tensor is assumed to be an unsorted 1D coordinate vector. This favors the COO organization for efficient building, Therefore, we will only consider the COO as the baseline of our analysis. The COO organization provides a straightforward means of representing sparse tensors. It is particularly well-suited for tensors with a relatively low number of non-zero elements. An example of the COO organization for a 3D sparse tensor is depicted in Fig. 1(a). Both the COO column and the Value column are serialized independently into 1D buffers. These two buffers are concatenated and written into a single fragment. The time complexity to build COO organization is $O(1)$. Conversely, the time complexity to read COO organization is $O(n \times n^{read})$, where n^{read} denotes the number of points to be read, and n represents the number of points to be checked.

Sorting the coordinates can reduce the complexity of read to $O(\max\{n, n^{read}\})$, but it may takes extra time: $O(n \log(n))$ to sort before write and will have $O(n^{read} \log(n^{read}))$ in read. So, there are some trade-off to consider here, For simplify, we only consider the non-sorted version of COO organizations. The space complexity of the COO organization is $O(d \times n)$. Several variants of COO organizations, such as F-COO [20] and HiCOO [21], exist as well. However, these variants are optimized to accelerate specific applications like SpMTTKRP [22] or are tailored for specific hardware like GPUs. Therefore, our work focuses solely on the fundamental COO organization from the perspective of a storage system.

B. LINEAR Organization

The LINEAR organization stores the linearized offsets of coordinates in sparse tensors. An example of LINEAR organization is illustrated in Fig. 1(a). LINEAR organization is particularly relevant when considering the performance gap between CPUs and storage devices [23]. The LINEAR organization invests extra time to transform the coordinates of each sparse point into a linear address. However, this approach leads to substantial storage space savings.

In this study, we use the row-major order to transform coordinates into linear addresses. For a point with coordinates $(c_1, c_2, \ldots, c_n)$, its linear address is calculated as $\sum_{i=1}^{n} c_i \cdot \prod_{j=i+1}^{n} m_j$. The time complexity for building LINEAR is $O(d \times n)$, and the time complexity for reading LINEAR is $O(n \times n^{read})$. The space complexity of the LINEAR organization is $O(n)$. The risk of using the LINEAR organization is that the overflow of linear address when converting a multiple dimensional coordinates for an extremely large tensor into a single value. A practical solution to this problem is to break large tensors into small blocks, as most block based structures [16] have done. Our algorithms can use local boundary of each block to perform the transform.

C. Generalized Compressed Sparse Row (GCSR)

Compressed Sparse Row (CSR) [24], also known as Compressed Row Storage (CRS), is a classic organization for 2D tensors (i.e., matrices). Generalized Compressed Sparse Row (GCSR) [12] maps points from high-dimensional tensors into 2D arrays and then uses CSR to store them. GCSR uses fixed 2D tensors for its storage system. GCSR has been implemented differently in [12] and [13]. This work uses the implementation presented in Algorithm 1, which is referred as GCSR++ in this paper. The GCSR++_BUILD converts input coordinate vectors b_{coor} to GCSR format. It also returns a map vector to reorganize the value if needed (see more details in the section III and Algorithm 3). The GCSR++_READ get the value for coordinates in b_{coor}.

We first extract the boundary from b_{coor} to determine the size of the sparse tensor. We then calculate the size of the resulting 2D arrays, selecting the smallest dimension as the first dimension in the 2D array and multiplying the other dimension sizes as the size of the second dimension. For each point, we transform coordinates from b_{coor} to coordinates in 2D space. Once the transformation is complete, we sort the

Algorithm 1 Build and Read algorithm used by GCSR++

function GCSR++_BUILD(b_{coor})
  Input: b_{coor}: buffer of coordinate to write
  Output: b: buffer of compressed coordinates
    map: records original index in sorting b_{coor}
  $s_l \leftarrow$ Extract local boundary from b_{coor}
  $s^{2D}_{new} \leftarrow$ Select smallest dimensions in s_l as the # of $rows$ and multiple the sizes of left dimensions as the # of $columns$ in the result 2D matrix.
  for each point p_{coor} in b_{coor} **do**
    $l_{orig} \leftarrow$ transform$^{row-major}(p_{coor}, s_l)$
    $p^{2D}_{coor} \leftarrow$ reverse_transform$^{row-major}(l_{orig}, s^{2D}_{new})$
    $b^{2D}_{coor} \leftarrow p^{2D}_{coor} \cup b^{2D}_{coor}$
  end for
  $(b^{2D,s}_{coor}, map) \leftarrow$ Sort b^{2D}_{coor} by the first dimension
  $(row_{ptr},\ col_{ind}) \leftarrow$ Package $b^{2D,s}_{coor}$ with the CSR.
  $b \leftarrow rows_{ptr}$ + $cols_{ind}$ ▷ Concatenate buffers
  return (b, map)
end function

function GCSR++_READ(b_{coor})
  Input: b_{coor}: buffer of coordinate to read
    f: fragment metadata
  Output: b_{data}: buffer of read data
  $(row_{ptr},\ col_{ind}) \leftarrow$ Extract metadata from f
  $b^{2D}_{coor} \leftarrow$ Convert b_{coor} into 2D coordinates with same algorithm in the above GCSC++_BUILD
  for point p_{coor} in b^{2D}_{coor} **do**
    $l \leftarrow row_{ptr}[p_{coor}[0]]$, $u \leftarrow row_{ptr}[p_{coor}[0]+1]$
    if p_{coor} in $(col_{ind}[l], \ldots, col_{ind}[u])$ **then**
      $d \leftarrow$ Get data of p_{coor} from f
      $b_{data} \leftarrow b_{data} \cup d$
    end if
  end for
  return b_{data}
end function

Algorithm 2 Build and Read Algorithm for CSF Tree

function CSF_BUILD(b_{coor})
  Input: b_{coor}: buffer of coordinate to read
  Output: b: buffer of compressed coordinates
    map: records original index in sorting b_{coor}
  $s_l \leftarrow$ Extract local boundary from b_{coor}
  $s^s_l, m^{dim} \leftarrow$ Sort s_{local} in ascending order, m^{dim} records map from original dimension to new dimension
  $b^s_{coor}, map \leftarrow$ Sort b_{coor} based on the m^{dim}
  Initialize $nfibs[1,\ldots,d][\ldots]$, $fids[1,\ldots,d][\ldots]$, and $fptr[1,\ldots,d-1][\ldots] \leftarrow 0$
  //Build leaf nodes of the last dimension r
  $nfibs[d-1] \leftarrow$ # of points in b_{coor}
  $fids[d-1] \leftarrow$ unique indexes in dimension $r-1$
  //Build root nodes of dimension 0
  $nfibs[0] \leftarrow$ # of unique indexes in dimension 0
  $fids[0] \leftarrow$ unique indexes in dimension 0
  $fptr[0] \leftarrow$ the split index in dimension 0
  //Build nodes from root to leaf
  for $i \leftarrow 1$ to $d-1$ **do**
    $fids[i] \leftarrow$ unique indexes in dimension i
    $nfibs[i] \leftarrow$ # of unique indexes in dimension i
    Update $fptr[i]$ based on fptr$[i-1]$
  end for
  $b \leftarrow nfibs$ + $fids$ + $fptr$ ▷ Concatenate buffers
  Return (b, map)
end function

function CSF_READ(b_{coor}, f)
  Input: b_{coor}: buffer of coordinate to read;
    f: fragment metadata
  Output: b_{data}: buffer of read data
  ($nfib$, $fptr$, $fids$) $\leftarrow$ Extract metadata from f
  for each point p_{coor} in b_{coor} **do**
    $l = 0$, $u = nfib[0]$
    $found$=TRUE
    for $i \leftarrow 0$ to $d-1$ **do**
      if $p_{coor[i]} \in fids[l:u]$ **then**
        $fi \leftarrow$ index of $p_{coor[i]} \in fids[l:u]$
        $l = fptr[fi]$, $u = fptr[fi+1]$
      else
        $found$ = FALSE
        break
      end if
    end for
    if $found$ **then**
      $d \leftarrow$ Get data of p_{coor} from f
      $b_{data} \leftarrow b_{data} \cup d$
    end if
  end for
  return b_{data}
end function

points based on the coordinates of the first dimension (rows) of the 2D array. After sorting, we use the CSR algorithm [24] to package the sorted coordinates, resulting in a row pointer (row_{ptr}) and a column coordinate vector (col_{ind}). We show an example of GCSR++ organization in Fig. 1(b).

The GCSR++_BUILD function comprises two major steps: transforming the coordinates and sorting the points. Consequently, the time complexity of GCSR++_BUILD is $O(n\log(n) + 2 \times n)$, where the $n\log(n)$ complexity arises from the sorting steps, and $2 \times n$ complexity arises from the steps involving the transformation of the coordinates and the construction of the CSR structure. The space complexity of GCSR++ is $O(n + \min\{m_1, \ldots, m_d\})$ for a sparse tensor containing n points. Essentially, GCSR++ stores the row indices for each point and also maintains pointers to the start of each row in the index vector. GCSR++ exhibits a space complexity that is very close to the LINEAR organization.

The GCSR++_READ function extracts row_{ptr} and col_{ind} from the fragment. Subsequently, it converts all coordinates in b_{coor} into 2D tensors as well. Afterward, it reads each points only by searching a row. The current implementation of GCSR++_READ does not sort b^{2D}_{coor} like GCSR++_BUILD does. The primary reason for this is that sorting incurs a time complexity of $O(n^{read}\log(n^{read}))$. In contrast, the current implementation has a time complexity of $O(n^{read} \times \frac{n}{\min\{m_1,\ldots,m_d\}} + n)$, where n^{read} denotes the number of points in b^{2D}_{coor} being searched, n the one pass to transform the coordinates, and $\frac{n}{\min\{m_1,\ldots,m_d\}}$ the average number of points in each row to compare for each read point.

D. Generalized Compressed Sparse Column (GCSC)

GCSC++ is very similar to the GCSR++ algorithms in building and reading sparse tensors. To avoid the repetition of the

TABLE I: The Time and Space Complexity of Using Different Organizations to Store and Access Sparse Tensors.

Layouts	Time Complexity of Building	Time Complexity of Reading	Space Complexity
COO	$O(1)$	$O(n \times n^{read})$	$O(n \times d)$
LINEAR	$O(n \times d)$	$O(n \times n^{read})$	$O(n)$
GCSR++	$O(n \times \log(n) + 2 \times n)$	$O(n^{read} \times \frac{n}{\min\{m_1,\ldots,m_d\}} + n)$	$O(n + \min\{m_1, \ldots, m_d\})$
GCSC++	$O(n \times \log(n) + 2 \times n)$	$O(n^{read} \times \frac{n}{\min\{m_1,\ldots,m_d\}} + n)$	$O(n + \min\{m_1, \ldots, m_d\})$
CSF	$O(n \times \log(n) + n \times d)$	$O(n^{read} \times \frac{n}{d})$	$O(n \times d)$

same contents, we will ignore the detailed description of the GCSC++ in the paper. An example of GCSC++ organizations is presented in Fig. 1(c). In the following parts of this subsection, we only list the major differences between GCSC++ and GCSR++: (1)instead of selecting the $\min\{m_1, \ldots, m_d\}$ as the size of rows, GCSC++ selects the minimum size of the original tensor as the size of columns in the result 2D tensor; (2) After transforming the b_{coor} to the b^{2D}_{coor}, GCSC++ sorts all points by their column index; (3) The result 2D tensor is packaged with the classic Compressed Sparse Column (CSC) algorithm; (4) GCSC++ reads points column by column. In general, the space complexity for the GCSC++ is $O(n + \min\{m_1, \ldots, m_d\})$. The time complexity for the build algorithm for GCSC++ is $O(n\log(n) + 2 \times n)$. The time complexity for the read algorithm for GCSC++ is $O(n^{read} \times \frac{n}{\min\{m_1,\ldots,m_d\}} + n)$.

E. Compressed Sparse Fibers (CSF)

The Compressed Sparse Fibers (CSF) data organization [14, 15] provides a generalization of data organization for high-dimensional sparse tensors. We show an example of CSF organizations in Fig. 1(d). CSF uses a tree structure to represent sparse tensors. The number of levels in this tree corresponds to the number of dimensions of the sparse tensors. At each level of the tree, CSF strives to minimize duplicated coordinates. In the subsequent sections of this subsection, we will elaborate on the methods for constructing the CSF tree, materializing it, and retrieving data through tree-based queries.

Following the established algorithms outlined in [14, 15], the CSF tree can be represented using three key data structures: $nfibs[1, \ldots, d]$: a 1D vector records the count of coordinates at each level; $fids[1, \ldots, d][\ldots]$: a 2D vector that stores all coordinates at each level; $fptr[1, \ldots, d-1][\ldots]$: a 2D vector that records the split index from the previous dimension to the current dimension. For instance, referring to the example we provided in Fig. 1(d), the data structures are populated as follows: $nfibs = \{2, 3, 5\}$, $fids = \{\{0, 2\}, \{0, 1, 2\}, \{1, 1, 2, 1, 2\}\}$, and $fptr = \{\{0, 2, 3\}, \{0, 1, 3, 5\}\}$.

The CSF_BUILD function in Algorithm 2 identifies the local boundary size, denoted as s_l, from the b_{coor} tensor. These sizes are then sorted in ascending order to maximize the opportunity for reducing duplicated coordinates in the first dimension (i.e., the root node). Simultaneously, this sorting operation helps reduce the size of the leaf nodes. Because sorting s_l may alter the order of data in the buffer, CSF_BUILD ensures that the b_{coor} buffer is sorted accordingly. CSF_BUILD initializes the data structures $fids$, $fptr$, and $fptr$. Subsequently, it proceeds to create the leaf nodes of the CSF tree starting from the last dimension. Following this, CSF_BUILD creates the root nodes of the CSF tree, beginning with the first dimension. The intermediate nodes of the CSF tree are constructed by identifying unique coordinates for each branch of nodes in the previous level, with these unique coordinates being stored in $fids$. The $fptr$ is updated accordingly for each intermediate level. Finally, the buffers containing $fids$, $fptr$, and $fptr$ are serialized and concatenated into a single buffer at the conclusion of the algorithm. The time complexity of CSF_BUILD is $O(n\log(n) + n \times d)$, where the $n\log(n)$ term arises from the sorting algorithms, and the $n \times d$ term originates from the steps involved in building the CSF tree.

The space complexity for the CSF_BUILD organization is highly dependent on the characteristics of the input sparse tensors: (1) **Worst Case**: In the worst-case scenario, where there are no duplicated coordinates from the root to the leaf, and each point forms its own tree, the space complexity for CSF is $O(d \times n)$. This occurs when there is maximum divergence in the coordinate values. (2) **Average Case**: In an average case, where approximately half of the points are duplicated at each level of the CSF, the space complexity for CSF is approximately $O(2n \times (1 - (1/2)^d))$. This case represents a balance between duplication and divergence in coordinate values. (3) **Best Case**: In the best-case scenario, where only a single point exists in the non-leaf nodes (minimal branching), the space complexity for CSF is $O(n + d)$. This case happens when the tree structure is very compact with minimal branching. These space complexities give a clear understanding of how the organization of data in CSF is affected by the distribution of coordinates.

The CSF_READ performs a search for each point in the b_{coor} against the CSF tree. This search essentially involves checking if the coordinate exists in the CSF tree starting from the root. The $fptr$ data structure helps narrow down the search space for each point. The time complexity of the CSF_READ algorithm is indeed $O(n \times d)$. This linear time complexity arises because, for each point, the algorithm traverses the CSF tree from the root to locate the coordinate, and this process is repeated for all n points. It's an efficient algorithm for querying sparse tensors organized in the CSF format.

In Table I, we provide a comprehensive overview of the time and space complexities associated with different organizations for storing and accessing sparse tensors.

III. Experimental Evaluation of Different Sparse Data Organizations and Access Algorithms

We conducted our experiments on the Perlmutter supercomputer [1] at NERSC, which contains a minimum of 3072 nodes. Each node has an AMD EPYC 7763 CPU. The system is

[1]https://www.nersc.gov/systems/perlmutter/

configured with a Lustre fifile system. Our benchmarks are based on Algorithm 3. The WRITE function takes three key parameters: b_{coor}, b_{data}, and m. The b_{coor} and b_{data} parameters contain buffers for coordinates and data from a sparse tensor to be written, respectively. The WRITE function transforms b_{coor} into b_{coor}^{new} by calling functions of each organization. It may reorganize b_{data} based on the new order, which is stored in the map vector. The $map[i]$ records the new index of the i-th point in the new b_{coor}^{new}. It concatenates b_{coor}^{new} and b_{data} and write result into a binary fragment file. The READ function takes a parameter, b_{coor}. READ identifies all fragment files that overlap with the coordinates in b_{coor}. For each discovered fragment, READ extracts data from it and appends it to a list L. The list L is structured with coordinate vectors and their corresponding values. After retrieving data from all fragments, READ sorts the data based on the linear address. Finally, READ populates the buffer b_{data} from L.

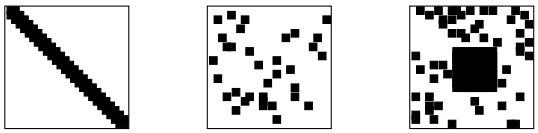
TABLE II: Size and density of the synthetic data sets

Dimension and Size	TSP	CGP	MSP
2D (8192×8192)	1.67%	0.99%	0.19%
3D ($512 \times 512 \times 512$)	3.47%	0.99%	0.19%
4D ($128 \times 128 \times 128 \times 128$)	8.22%	0.90%	0.21%

A substantial collection of sparse datasets is readily available [25]. Assessing the efficiency of different sparse tensor organizations across all these datasets is an impractical endeavor. Therefore, we commence by examining these datasets comprehensively, subsequently distilling three prevalent patterns characterizing sparse tensors:

- Tridiagonal Sparse Pattern (TSP): An example of TSP is illustrated in Fig. 2(a). In TSP, values are concentrated along the tridiagonal bands, while the remaining elements are either zeros or missing values.It can be found in one-hot encoding for categorical variables [26] and stencil computing for solving partial differential equations [27].
- General Graph Sparse Pattern (GSP): An example of GSP is depicted in Fig. 2(b). GSP describes a tensor where data points exist at random coordinates. GSP is frequently observed in the adjacency matrices of graphs [28], which are used for representing social networks or recommendation systems. GSP can represent most tabular datasets.
- Mixed Sparse Pattern (MSP): An example of MSP is presented in Fig. 2(b). MSP pattern has a dense area among the random sparse points. The MSP pattern is used by scientific applications, such as the Linac Coherent Light Source (LCLS) II experiment [29], to store their experimental data.

The Table II summarizes the synthetic datasets employed for the evaluation of various organizations. These synthetic datasets encompass dimensions ranging from 2D to 4D, effectively covering a wide range of sparse tensors studied. The sparsity of these tensors has been carefully controlled to remain under 10%. The data type for the synthetic data coordinates is standardized as *unsigned long long int*, occupying 8 bytes of storage space o. Other data types should have the same results. Additional factors governing the generation of these synthetic datasets are elaborated upon as follows:



(a)TSP (b)GSP (c)MSP

Fig. 2: Three major patterns in sparse tensors.

- For the TSP, the length of the tridiagonal band is set to 9. In the CGP, a (0,1) random number generator is employed to determine whether a cell of the sparse tensor should have a value (when the number is bigger than 0.99 threshold). In the MSP, the probability threshold is increased to 0.999, and the contiguous region is defined with a starting address of ($\frac{m_1}{3}$, ..., $\frac{m_d}{3}$) and a size of ($\frac{m_1}{3}$, ..., $\frac{m_d}{3}$).
- In the reading test on sparse tensors, we extract a contiguous region with a starting address of ($\frac{m_1}{2}$, ..., $\frac{m_d}{2}$) and a size of ($\frac{m_1}{10}$, ..., $\frac{m_d}{10}$) from the sparse tensor. This region covers independent points in TSP and CGP but includes both independent points and contiguous points in MSP.
- Time measurements are taken in seconds for the WRITE and READ function in Algorithm 3) using different storage organizations. The size of the result files is measured in bytes, representing the file stored in the Lustre file system.

Algorithm 3 Main Algorithm of Our Benchmark

function WRITE(b_{coor}, b_{data})
  Input: b_{coor}: buffer of coordinate to read
    b_{data}: buffer of data to write
  $(b_{coor}^{new}, map) \leftarrow$ Call BUILD function of an organization to package b_{coor}
  Reorganize b_{data} based on map if necessary
  $b_{frag} \leftarrow b_{coor}^{new} + b_{data}$ ▷ Concatenate buffers
  Write b_{frag} as a fragment f
end function

function READ(b_{coor})
  Input: b_{coor}: buffer of coordinate to read
  Output: b_{data}: buffer of data to read
  $F \leftarrow$ Find all fragments containing b_{coor}
  $L \leftarrow$ Initialize empty list of $\langle$coor, value$\rangle$
  for each fragment f in F **do**
    $b_{coor}^{frag} \leftarrow$ Extract and unpack index from f
    $b_{coor}^{com} \leftarrow b_{coor} \cap b_{coor}^{frag}$
    $b_{data}^{com} \leftarrow$ Read values of b_{coor}^{com} from f with organization-specific methods, such as CSF_READ() in Alg.2 and GCSR++_READ in Algo. 1
    Append $\langle b_{coor}^{com}, b_{data}^{com} \rangle$ to L
  end for
  Sort L based on linear address
  Extract value from L into b_{data}
  return b_{data}
end function

A. The Time of Writing Sparse Tensors

The process of writing sparse tensors involves several steps. Hence, the time of writing include the time required to construct the structures for coordinates, the time needed to reorganize values, and the time taken to write both the coordinates and values to fragments to Lustre. In the preceding section, we presented our predictions for the time required to build coordinate structures and ranked in the following order from the fastest to the slowest: COO > LINEAR > GCSR++ $\geq$ GCSC++ > CSF. The actual measured times for writing sparse tensors are presented in Fig. 3. Our experiments align

with the trends predicted by our theoretical analysis. Specifically, both COO and LINEAR outperform CSF, GCSR++, and GCSC++. However, it is worth noting that while our theoretical analysis suggests that COO should be faster than LINEAR, our experimental results indicate the opposite. To shed light on this discrepancy, we provide a breakdown of the time taken by the write function in Table III.

As the data reveals, the actual time required for COO is negligible (zero), whereas for LINEAR, it amounts to 0.0109 seconds. This finding aligns with our theoretical analysis. However, it is important to note that the time it takes for COO to write the organizations of sparse tensors to the Lustre file system is nearly three times longer than that of LINEAR. As we will delve into in the next section, COO results in a file that is four times larger in size. Consequently, COO spends more time writing data to the file system, making it slower than LINEAR in terms of overall writing performance. In essence, while COO can save time in constructing the organization, it incurs a time cost in saving a larger result file.

TABLE III: Breakdown for the total time to write sparse tensors for the 4D MSP pattern.

	COO	LINEAR	GCSR++	GCSC++	CSF
Build	0	0.0109	0.1888	0.4484	0.3014
Reorg.	0	0	0.0073	0.0195	0.0073
Write	0.1217	0.0504	0.0493	0.0513	0.0751
Others	0.0177	0.0167	0.0179	0.0174	0.0179
Sum	**0.1393**	**0.0780**	**0.2634**	**0.5366**	**0.4017**

Another noteworthy disparity between our theoretical predictions and the experimental findings pertains to GCSC++. According to our theoretical analysis, GCSC++ should exhibit faster performance compared to CSF and be on par with GCSR++. The critical point to highlight is that GCSC++ and GCSR++ share the same underlying algorithmic implementation. Their primary distinction lies in the data layout within the buffer. The implementation of Algorithm 3 takes input in a row-major ordering, which aligns seamlessly with the access pattern of GCSR++ but doesn't correspond to the data access sequence in GCSC++. This discrepancy significantly impacts the performance of line 12 and line 13 in Algorithm 1, as well as the time required for value reorganization. This is clearly evident in the breakdown of time presented in Table III, where GCSR++ and GCSC++ demonstrate similar times for writing organizations, indicative of similar organization sizes. However, the substantial difference arises from the time needed to construct these organizations. As previously elucidated, this disparity arises because the input coordinate buffer for GCSC++ is in row-major ordering, whereas GCSC++ must convert it into column-major ordering during the sorting process. In summary, GCSC++ and GCSR++ can indeed exhibit superior performance when their buffers are organized according to their respective layouts.

B. File Size of Different Organizations for Sparse Tensors

The size of the resulting file in a storage system constitutes another critical factor when comparing various organizations for sparse tensors. Typically, users prefer the file size to be as compact as possible. The experimental results, quantifying the sizes of result files for different organizations, are presented in Fig. 4. In our theoretical analysis, we ranked the organizations as follows: LINEAR $<$ GCSR++ $\leq$ GCSC++ $\leq$ CSF $\leq$ COO.

Evidently, our experimental results substantiate the rankings established in our theoretical analysis. Additionally, our experiments confirm that CSF indeed exhibits variable space sizes across different sparse patterns. Even for the same pattern, such as MSP, CSF shows significant variations in size from 2D to 4D. As previously mentioned, this variability arises because CSF's space complexity ranges from $O(n + d)$ to $O(n \times d)$.

Furthermore, we note that both GCSR++ and GCSC++ yield very similar file sizes, slightly larger than LINEAR. This similarity arises from the fact that they only store additional pointers for the number of rows or columns. When considering COO as the baseline, the potential reduction in storage space can be as much as $O(d)$ times.

C. Time for Reading Sparse Tensors

The process of reading sparse tensors encompasses several key steps. These steps involve the time required to retrieve and extract metadata (index) from fragments, the time spent querying the existence of a value within the index, and the time needed to retrieve the value into the output buffer. Among these steps, the time allocated to querying the existence of a value in the index is particularly significant. Our theoretical analysis yields the following ordering for the time required to query the existence of a value for different organizations: CSF $\geq$ GCSR++ $\geq$ GCSC++ $>$ LINEAR $\geq$ COO.

Our experimental results are depicted in Fig. 5, and for the most part, they corroborate the findings of our theoretical analysis. Specifically, COO and LINEAR exhibit significantly slower performance compared to GCSR++, GCSC++, and CSF. However, there is an exception when it comes to CSF, particularly in its application to organizing 2D sparse tensors. In the context of 2D sparse tensors, CSF should theoretically be faster or at least on par with GCSR++ and GCSC++.

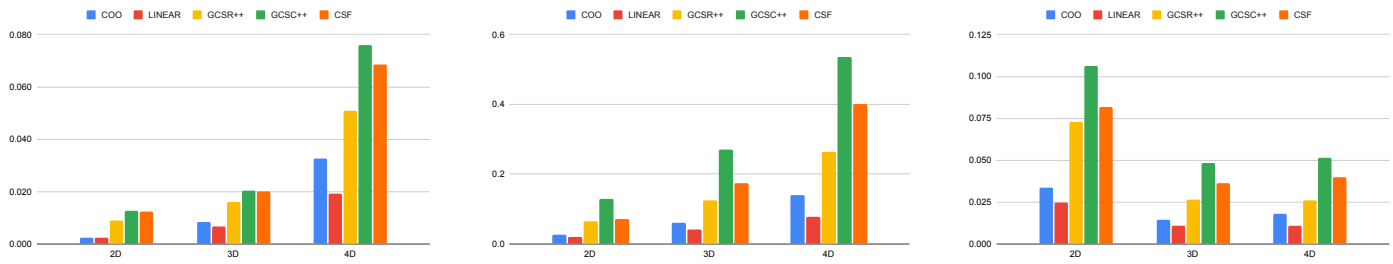
The primary reason for this phenomenon lies in the fact that GCSR++ and GCSC++ essentially represent the 2D CSR and CSC structures. As a result, there is no overhead associated with coordinate transformation for GCSR++ and GCSC++. As indicated by the time complexity analysis presented in Table I, the read time complexity of GCSR++ and GCSC++ increases as the number of dimensions rises. Conversely, the read time complexity of CSF decreases as the number of dimensions decreases. Consequently, CSF exhibits lower performance when handling 2D tensors but surpasses the performance of GCSR++ and GCSC++ when dealing with 3D or 4D tensors.

TABLE IV: Overall scores of different organizations

	COO	LINEAR	GCSR++	GCSC++	CSF
Scores	0.76	0.34	0.36	0.50	0.48

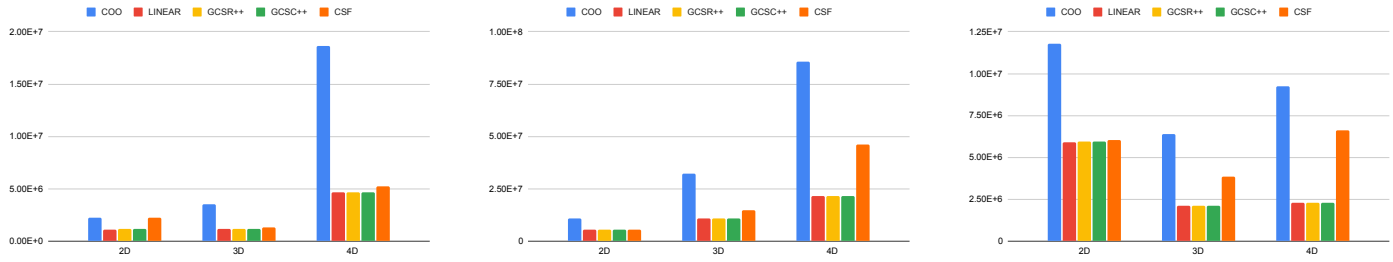
IV. LESSONS LEARNED

Based on our experimental results, we can highlight the valuable insights we've gained from this study:



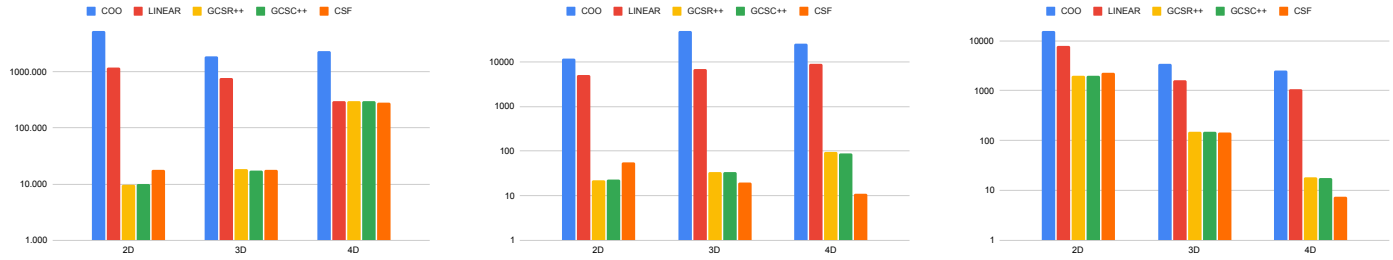
(a) TSP (b) GSP (c) MSP

Fig. 3: Writing Time of Different Storage Organizations for Sparse Tensors with TSP, GSP, and MSP patterns



(a) TSP (b) GSP (c) MSP

Fig. 4: File Size of Different Storage Organizations for Sparse Tensors with TSP, GSP, and MSP patterns



(a) TSP (b) GSP (c) MSP

Fig. 5: The time required to read sparse tensors organized using various storage schemes.

- Trade-offs Between Write, Read and File Siz. Overall, the LINEAR has the highest scores in providing balanced performance among the time of access and the file size. GSCR++ has very close score to the LINEAR. The score ($\bar{s}$) is defined as below: $\bar{s} = \frac{1}{w_i^{pattern}} \sum^{\text{TSP, GSP, MSP}} (\frac{1}{w_i^{dimension}} \sum^{\text{2D, 3D, 4D}} r_i)$, $r_i = \frac{m_i}{\max\{m_1, m_2, m_3, m_4, m_5\}}, i \in \{1, 2, 3, 4, 5\}$, where m_i is the measurement of each metric for each organizations and it is normalized as r_i with maximum value of all organizations. The r_i is summarised with weight for patterns. Here we assume all weights are equal. Scores of different organizations are presented in Table IV.
- Locality is preserved very well by transferring high-dimensional and sparse tensors to 2D tensors for storage and access with GSCR++ and GCSC++. GCSC++ and GCSR++ have better performance in writing high-dimensional and sparse tensors when their buffer. COO can save time in building organization but it pays back with time to save larger result file. LINEAR spends a bit extra time to linearize coordinates but it save the time to write result file to disk.
- The file size can be significantly reduced (at most $O(d)$) by converting high-dimensional and sparse tensors into low-dimensional and sparse tensors. LINEAR, GCSR++, GCSC++, and COO has determined size to organize sparse tensors. CSF has significant variances in space size because its tree structures to organize sparse tensors.
- CSF has good performance scalability in reading data from the low-dimensional and sparse tensors to high-dimensional and sparse tensors. GCSR++ and GCSC++ may suffer from scalability issues from low-dimensional and sparse tensors to high-dimensional and sparse tensors.

V. Related Work

In the case of 1D sparse tensors (vectors), they are typically stored using a "vector of pairs" approach. Most research efforts in this domain have focused on 2D sparse tensors, with the CSR and CSC formats [24] being the most classic. These formats compress duplicated coordinates of one dimension of 2D tensors using pointers. Variants of CSR and CSC, such as block CSR or block CSC [30], have been proposed. Unfortunately, these organizations are not directly applicable to high-dimensional and sparse tensors.

Managing sparse tensors with more than two dimensions has become crucial in the context of machine learning [31]

and scientific activities [32]. Numerous efforts have aimed to extend CSR and CSC for the storage of high-dimensional sparse tensors. These efforts include GCRS/GCCS [12], pydata/sparse [13], CISR [5], and CISS [6]. Among these, GCRS/GCCS and pydata/sparse represent more general extensions from a storage perspective, while CISR and CISS focus on cache-efficient optimizations when reading data from memory into the CPU or hardware like FPGA. Another trend involves exploring tree structures to store sparse tensors, offering a more general solution for high-dimensional and sparse tensors. For example, Tensor algebra [4] organizes sparse tensors as row-major coordinate storage trees or column-major coordinate trees, while CSF [14, 15] provides a more versatile representation for sparse tensors. Despite the merits and drawbacks of different organizations, none of the existing work has conducted a comprehensive study to compare them. To the best of our knowledge, this is the first work to comprehensively compare and study these representative approaches for high-dimensional sparse tensor storage.

VI. Conclusion

After an extensive study of sparse storage formats, we identified key data organizations for a comprehensive investigation, balancing theoretical analyses and empirical assessments. Our findings highlight a storage organization's pivotal role in the performance of read and write sparse tensor. Our key observations include: (1) Linear addressing strikes the best balance between storage size and access time; (2) Transforming high-dimensional tensor data into lower-dimensional formats enhances storage and access efficiency; (3) Tree-structured organizations excel in low-dimensional tensors and perform exceptionally well in high-dimensional tensors without dimension transformation. These insights contribute to a nuanced understanding of sparse tensor storage formats, guiding informed choices in practical applications. In future, we plan to explore automatic strategies for selecting different organization for applications based on the characterization of sparsity in their data.

Acknowledgment

This effort was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility.

References

[1] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *MICRO '52*. New York, NY, USA: ACM, 2019, p. 151–165.

[2] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *MICRO '52*. New York, NY, USA: ACM, 2019, p. 319–333.

[3] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, "Sparseloop: An analytical approach to sparse tensor accelerator modeling," 2023.

[4] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133901

[5] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 36–43.

[6] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *HPCA*, 2020.

[7] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," in *IPDPS*. IEEE, 2017, pp. 1058–1067.

[8] PyData. (2023 Sept 22) Sparse multi-dimensional arrays for the pydata ecosystem. GitHub repository. [Online]. Available: https://github.com/pydata/sparse

[9] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 36–43.

[10] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The tiledb array data storage manager," *Proc. VLDB Endow.*, vol. 10, no. 4, p. 349–360, nov 2016. [Online]. Available: https://doi.org/10.14778/3025111.3025117

[11] J. Mainzer, N. Fortner, G. Heber, E. Pourmal, Q. Koziol, S. Byna, and M. Paterno, "Sparse data management in hdf5," in *XLOOP*. IEEE, 2019, pp. 20–25.

[12] M. A. H. Shaikh and K. A. Hasan, "Efficient storage scheme for n-dimensional sparse array: Gcrs/gccs," in *HPCS*, 2015, pp. 137–142.

[13] PyData, "Sparse: A fast, memory-efficient sparse matrix library for python," https://github.com/pydata/sparse, 2023 Sept 7th.

[14] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 61–70.

[15] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," ser. IA'15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833179.2833183

[16] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Trans. Graph.*, vol. 25, no. 3, p. 579–588, jul 2006.

[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *ACM SIGMOD*, 1990, pp. 322–331.

[18] S. Byna, M. S. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "Exahdf5: delivering efficient parallel i/o on exascale computing systems," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 145–160, 2020.

[19] W. Zhang, S. Byna, C. Niu, and Y. Chen, "Exploring metadata search essentials for scientific data management," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 83–92.

[20] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 47–57.

[21] J. Li, J. Sun, and R. Vuduc, "Hicoo: Hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 238–252.

[22] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: Scaling tensor analysis up by 100 times - algorithms and discoveries," in *SIGKDD*, ser. KDD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 316–324. [Online]. Available: https://doi.org/10.1145/2339530.2339583

[23] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH*, vol. 23, no. 1, pp. 20–24, 1995.

[24] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[25] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The suitesparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019. [Online]. Available: https://doi.org/10.21105/joss.01244

[26] J. Brownlee, *Data Preparation for Machine Learning*, 2023.

[27] J. Dongarra, M. Heroux, and P. Luszczek, "Hpcg benchmark: a new metric for ranking high performance computing systems. knoxville, tennessee: lectrical engineering and computer sciente department, knoxville, tennessee; 2015."

[28] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.

[29] J. Mainzer, N. Fortner, G. Heber, E. Pourmal, Q. Koziol, S. Byna, and M. Paterno, "Sparse data management in hdf5," in *XLOOP*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2019, pp. 20–25. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/XLOOP49562.2019.00009

[30] A. Buluç, J. Gilbert, and V. B. Shah, *13. Implementing Sparse Matrices for Graph Algorithms*, pp. 287–313. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719918.ch13

[31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for Large-Scale machine learning," in *OSDI 16*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[32] A. Shoshani and D. Rotem, *Scientific data management: challenges, technology, and deployment*. CRC Press, 2009.