Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Huawei Technologies
Zurich Research Centre

Master Thesis

Rethinking Sparse Tensor Storage: Incremental Formats as a Path Towards Maximizing Tensor-Vector Multiplication Efficiency

May 2nd, 2023

Supervised by:
Dr. Albert-Jan Yzelman
Prof. Dr. Olaf Schenk

Author:
Xiaohe Niu

Contents

Acknowledgements **i**

Abstract **iii**

1 Introduction **1**

2 Background and Related Work **3**
2.1 Background . . . 3
2.1.1 Notation . . . 3
2.1.2 Tensor-vector Multiplication (TVM) . . . 3
2.1.3 Increment . . . 3
2.1.4 Mode Dependency . . . 3
2.1.5 Traversal curves . . . 5
2.2 Related Work . . . 5

3 Format Options for Efficient Storage of Sparse Tensors **9**
3.1 MoSk-IF: Incremental Compressed Sparse Fibres with Mode Skipping 9
3.2 Ind-IF: Incremental Sparse Fibres with Index Identification . . . 10
3.3 Bit-IF: Incremental Sparse Fibres with Bit Encoding . . . 14

4 Tensor-Vector Multiplication **17**
4.1 MoSk-IF . . . 17
4.2 Ind-IF . . . 19
4.3 Bit-IF . . . 22

5 Cost Analysis and Efficiency: A Theoretical Study **25**
5.1 Tensor Storage: Cost Analysis . . . 25
5.2 Tensor-Vector Multiplication: Efficiency Factors . . . 26
5.2.1 Mode Dependency . . . 26
5.2.2 Traversal Curve . . . 27
5.2.3 Blocking . . . 27
5.3 Tensor-Vector Multiplication: Estimated Data Movement . . . 28

5.3.1 Lexicographical Tensor Traversal 29
5.3.2 Arbitrary Tensor Traversal 33
5.3.3 Tensor-Vector Multiplication: Blocking 36
5.4 Choosing the Optimal Storage Format 37

6 Implementation Details and Experiments **39**
6.1 Implementation Details 39
6.1.1 Construction of Bit-IF and Blocking Bit-IF 39
6.1.2 Computing the Z-Curve Order and Hilbert Order 40
6.1.3 Constructing $\mathcal{U}_B$ 40
6.1.4 Selecting Blocksizes β and β_B 41
6.2 Experiments 42
6.2.1 Setup 42
6.2.2 Results 42

7 Conclusion **47**

Bibliography **49**

A Bit-IF Run time Comparison of Different Block Sizes **49**

B Bit-IF and COO Run time Comparison **65**

Acknowledgements

I would like to express gratitude to everyone who has supported me throughout the process of my Master's thesis. First and foremost, I would like to thank my supervisor Dr. Albert-Jan Yzelman for his guidance throughout the entire project. His expertise, patience, and willingness to explain complex concepts were instrumental in the successful completion of this thesis. His support made this journey an enjoyable and enriching experience. I would also like to thank Professor Dr. Olaf Schenk for being my ETH supervisor. I am grateful for his trust in my abilities, and for allowing me to work freely with my supervisor without any complaints. Finally, I would like to thank Daniel Dorigatti for his invaluable contributions to this project. His expertise in code optimization and his suggestions and explanations of different optimization methods were extremely helpful in improving the efficiency of my algorithms. I would also like to acknowledge the support of my family and friends, who have always motivated and encouraged me throughout this journey. Once again, thank you all for your support and encouragement throughout my thesis.

Abstract

This thesis introduces three novel methods for storing sparse tensors, which aim to improve the efficiency of tensor operations by compressing the tensor and thereby reducing storage requirements. These methods, namely Incremental Compressed Sparse Fibers with Mode Skipping (MoSk-IF), Independent Incremental Sparse Fibers (Ind-IF), and Incremental Sparse Fibers with Bit Encoding (Bit-IF) build upon existing sparse tensor storage techniques and are constructed by compressing the Coordinate storage format (COO). The efficiency of the tensor-vector multiplication (TVM) algorithm is evaluated using each of these storage formats for various tensor traversal curves. A theoretical analysis involves estimating storage cost, efficiency, and data movement, which help to identify the most suitable storage format. A systematic comparison and analysis of the strengths and weaknesses of each method is conducted and based on this evaluation, the Bit-IF storage format is implemented. The Bit-IF structure is found to perform well in comparison to the COO storage format in terms of TVM. This suggests significant potential for further optimization of the algorithm and for the application of Bit-IF to other tensor operations. Overall, this thesis contributes to the field of sparse tensor storage by providing novel storage optimization methods and fundamental concepts that can be used to improve the efficiency of tensor operations.

Chapter 1

Introduction

Sparse tensors are commonly utilised in various scientific and engineering domains, including scientific computing [**?**], data science [**? ?**], natural language processing [**?**], computer vision [**?**], social network analysis [**?**], and machine learning [**? ?**]. Nonetheless, the storage requirements for these tensors can be significantly high, especially when the data size increases.

To address this issue, this thesis proposes three storage methods for sparse tensors that aim to decrease the storage requirements and improve the efficiency of tensor operations. The presented storage techniques draw inspiration from existing methods [**? ? ? ?**].

The subsequent chapter provides background information and notation relevant to this thesis and discusses related work. The third chapter introduces the three storage formats that compress the Coordinate (COO) [**? ?**] storage format. The first storage format, called Incremental Compressed Sparse Fibres with Mode Skipping (MoSk-IF), employs an incremental approach with mode skipping and utilises separate arrays to store index increments for each mode. It switches to the relevant incremental array when an increment triggers an overflow, indicating an index change in the increment array of the next mode. The second storage format, Independent Incremental Sparse Fibres (Ind-IF), uses index increments and identification arrays to locate COO format increments. The third storage format, Incremental Sparse Fibres with Bit Encoding (Bit-IF), also stores the index increments, but unlike Ind-IF, there are no identification arrays. Instead, it uses a bit encoding array that indicates whether an increment should be applied for the current mode.

We conducted a theoretical analysis to evaluate the tensor-vector multiplication (TVM) algorithm using each storage format for different tensor traversal curves. This analysis explored the strengths and weaknesses of the three storage formats and described the TVM algorithm for each method. Furthermore, we examined the impact of varying tensor traversal order and blocking on the efficiency of the TVM. We analyzed and

compared the estimated data movement and complexity of each TVM algorithm to determine the most suitable storage format for implementation.

Following the theoretical analysis of each storage method, the Bit-IF storage format was implemented based on the derived algorithm to test its real storage and TVM efficiency on existing tensor data. For the Bit-IF storage format, we implemented a non-blocked and blocked variant, where the blocked Bit-IF is sorted according to the Z-curve [**?**] or Hilbert curve [**?**] on an inter-block level.

The experiments indicate that while there remains room for improvement in the compression and TVM algorithm, the implemented Bit-IF storage format compresses the COO by an average of 27% across all tested tensors. Furthermore, the Bit-IF storage format, in both blocked and non-blocked variants, generally achieves a faster run time for tensor-vector multiplication than the COO format. However, in certain instances, the blocked TVM exhibits inferior performance compared to the non-blocking TVM, and in rare cases, even underperforms the TVM with the COO format. The concepts introduced in this thesis could be expanded by further optimizing the TVM or incorporating supplementary operations such as tensor-matrix multiplication (TMM), the Matricised tensor times Khatri-Rao product (MTTKRP), or various tensor decomposition algorithms [**?**]. In conclusion, this thesis contributes a fundamental idea that can be optimised and expanded upon in future research, representing an advancement in the domain of sparse tensor storage.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Notation

Table 2.1 provides an overview of the notation that will be employed throughout the thesis.

2.1.2 Tensor-vector Multiplication (TVM)

We adopt the notation from Kolda et al. [**?**] and Pawlowski et al. [**?**] to denote tensors and vectors. Specifically, tensors are represented using capital letters in a calligraphic font, while vectors are denoted by lowercase bold letters. The kth mode **tensor-vector multiplication (TVM)** is denoted as $\mathcal{A} \times_k \mathbf{v} = \mathcal{B}$ where $\mathcal{A}$ is an input tensor of order d with dimensions $n_0 \times n_1 ... \times n_{d-1}$, $\mathbf{v}$ is an input vector with dimensions n_k and $\mathcal{B}$ is the resulting output tensor with dimensions $n_0 \times ... \times n_{k-1} \times 1 \times n_{k+1} ... \times n_{d-1}$.

2.1.3 Increment

In this thesis, the term **increment** is used to refer to any change in the value of an index: $\Delta i_j = i_{i+1,j} - i_{i,j}$. This definition is consistent with the broader usage of the term in mathematics and includes both increases and decreases in the value of the index.

2.1.4 Mode Dependency

We define a storage format for sparse tensors as **mode independent** if the iteration over the modes does not depend on a particular ordering. In other words, we can access the storage array of each mode in any order or reorder the mode arrays arbitrarily

Notations and variables	Description
$\mathcal{A} \in \mathbb{R}^{n_0 \times n_1 \times \ldots \times n_{d-1}}$	Input tensor
$\mathbf{v} \in \mathbb{R}^{n_k}$	Input vector
$\mathcal{B} \in \mathbb{R}^{n_0 \times \ldots \times n_{k-1} \times 1 \times n_{k+1} \ldots \times n_{d-1}}$	Output tensor
$d \in \mathbb{N}$	Order of the tensor
n_j	Size of dimension/mode j
ΔA	Increment arrays of $\mathcal{A}$
ΔB	Increment arrays of $\mathcal{B}$
Δi	Current increments
$val_{\mathcal{A}}$	Non zero values of of $\mathcal{A}$
$val_{\mathcal{B}}$	Non zero values of of $\mathcal{B}$
$id_{\mathcal{A}}$	Identification arrays of $\mathcal{A}$
$id_{\mathcal{B}}$	Identification arrays of $\mathcal{B}$
$b_{\mathcal{A}}$	Bit encoding array of $\mathcal{A}$
$b_{\mathcal{B}}$	Bit encoding array of $\mathcal{B}$
β	Block size
$\mathcal{U}_B$	Map storing the values of $\mathcal{B}$ for the non-zero indices
$nnz_{\mathcal{A}} \in \mathbb{N}$	Number of non-zero entries of $\mathcal{A}$
$nnz_{\mathcal{B}} \in \mathbb{N}$	Number of non zero entries of $\mathcal{B}$
$p_j \in [0, 1]$	Ratio of overflows at mode j: $\frac{1}{nnz_{\mathcal{A}}} \sum \mathbb{1}_{i_j + \Delta i_j \geq n_j}$
$q_j \in [0, 1]$	Ratio of index changes at mode j: $\frac{1}{nnz_{\mathcal{A}}} \sum \mathbb{1}_{\Delta i_j \neq 0}$
w_{short}	Size of a shot (unsigned) integer in bits
w_{int}	Size of an (unsigned) integer in bits
w_{long}	Size of a long integer in bits
w_{val}	Size of the tensor value type in bits
$w_{inc,l}$	$w_{inc,l} = \mathbb{1}_{\Delta i + n_i < 2^{32}} \cdot w_{int} + \mathbb{1}_{\Delta i + n_i \geq 2^{32}} \cdot w_{long}$
$w_{inc,s}$	$w_{inc,s} = \mathbb{1}_{\Delta i < 2^{16}} \cdot w_{short} + \mathbb{1}_{\Delta i \geq 2^{16}} \cdot w_{int}$
$x_{inc,l}$	$x_{inc,l} = \mathbb{1}_{\Delta i + n_i < 2^{32}} + 2 \cdot \mathbb{1}_{\Delta i + n_i \geq 2^{32}}$
$x_{inc,s}$	$x_{inc,s} = \frac{1}{2} \mathbb{1}_{\Delta i < 2^{16}} + \mathbb{1}_{\Delta i \geq 2^{16}}$

Table 2.1: List of notations

without affecting the sparse tensor structure. For instance, in a third-order tensor, a mode independent format would allow manipulation of columns before rows and tubes, or tubes before any other fibres. For instance, the Coordinate format (COO) is mode independent, as the indices for each dimension can be accessed in any order for every non-zero entry. In contrast, a **mode dependent** format cannot be accessed or reordered in an arbitrary manner and is typically constrained by a specific mode ordering. The Compressed Sparse Fibre (CSF) format from Smith et al. [**?**] is an example of a storage method that is not mode independent, as the indices pointers are hierarchically linked to the next mode given an orientation. Other concepts related to the storage and manipulation of sparse tensors include mode-generic and mode-

specific sparse tensors, as defined in Baskaran et al. [**?**] and mode orientation, as defined by Li et al. [**?**]. It is important to note that mode independence should not be confused with mode obliviousness, which refers to tensor kernels whose performance does not depend on the mode in which they are applied [**?**]. In the case of the tensor-vector multiplication, mode obliviousness is achieved if the performance is roughly similar for every mode we apply it to [**?**].

2.1.5 Traversal curves

For the computation of the tensor-vector multiplication (TVM), we consider three tensor traversal curves:

- Lexicographical curve
- Z-curve traversal
- Hilbert traversal

The **lexicographical curve** involves traversing the tensor in a specific order, where we begin with mode $d-1$, then move to mode $d-2$ and so on until we reach mode 0. To accomplish this, we first sort the tensor, which is in COO storage format, by the last column corresponding to mode $d-1$ in ascending order, while keeping the other indices from mode 0 to $d-2$ constant. Next, we sort the tensor by the second to last column corresponding to mode $d-2$, while keeping the last column sorted in ascending order. We repeat this process until we have sorted the tensor by all columns except the last one, which is already sorted in ascending order. The **Z-curve**, also known as the Morton curve or Morton order, was first introduced by Morton [**?**]. The Morton order is constructed through a series of repeated subdivisions, where the space under consideration is divided into two equal parts along each dimension during each iteration. For example, in a 2D space, the division results in four equal parts during each iteration, while in a 3D space, it yields eight equal parts. The divided parts are then traversed in a specific order that resembles the letter "Z" leading to the name "Z-curve" [**?**]. The **Hilbert curve** [**?**], is also a space-filling curve that partitions the space into two equal parts along each dimension in an iterative manner. However, unlike the Morton curve, the order of the partitions is chosen such that every following partition is connected to the prior one, resulting in a continuous path.

2.2 Related Work

In this related work section, we will discuss various approaches for tensor storage optimization and tensor-vector multiplication. The field of sparse tensor storage optimization has seen significant advancements in recent years, with various techniques being proposed to improve the efficiency and performance of tensor operations.

For matrices, which are tensors or order two, Koster introduced the Incremental Compressed Row Storage (ICRS) [**?**], which is an incremental version of the Compressed Row Storage (CRS) [**?**] that stores differences between consecutive column indices. A row change is signaled by overflowing the column index and the row array can be replaced with an incremental row array. The primary goal of ICRS was to further compress the storage format and reduce the memory requirements for sparse matrices, particularly for those with a small number of non-zeros per row. This compression is more effective when the differences between consecutive column indices are small, as smaller integer values require fewer bits to represent. However, ICRS is not capable of storing nonzeroes in any ordering other than the CRS ordering.

Another improvement to the CRS was introduced by Buluç et al. [**?**], called the Compressed Sparse Blocks (CSB), by partitioning the sparse matrix into equal-sized square blocks and only storing the non-zero elements in a compact manner. The partitioning is done in such a way that many or most of the individual blocks have a low density of non-zeros, i.e. a low ratio of non-zeros to matrix dimension. The row and column indices are relative to the block containing the particular element and hence require fewer bits compared to the entire matrix indices. Within each block, CSB uses a Z-curve ordering to store the non-zero elements. This ordering provides good parallelism within a block and spatial locality. The ordering among blocks can be flexible, with a Z-curve or recursive ordering potentially offering better performance. The limitation of CSB lies in the assumptions about the matrix dimension, as it assumes that the block size is an exact power of two and that it divides the matrix dimension. Furthermore, CSB's performance benefits are more pronounced for specific types of sparse matrices and access patterns.

Yzelman and Bisseling [**?**] proposed an extension to the ICRS by allowing negative increments, leading to the Bi-directional Incremental Compressed Row Storage (BICRS). This enables the traversal of non-zero elements from the row array to the column array, i.e. in a bi-directional and non-CRS order. While this increases the number of memory accesses required to traverse the matrix, it is an improvement over the triplet structure used in other methods. By storing the non-zero elements of a sparse matrix in Hilbert order, BICRS works well on large unstructured matrices, exploiting spatial locality and enhancing cache efficiency during matrix-vector multiplication. However, it does not outperform the standard CRS if the matrix is already favorably structured.

The generalisation of CRS for tensors was introduced by Smith et al. [**?**], leading to the Compressed Sparse Fiber (CSF) storage format. It utilises a tree-based data structure, where each level of the tree corresponds to a specific mode of the tensor and recursively compresses a sparse tensor along its modes. The non-zero elements are stored in a depth-first traversal order. Similar to the CRS, the CSF uses an array of pointers to store the starting positions of the non-zero elements of a fibre in the next level for each level of the tree. The non-zero tensor values are stored in a separate array. Compressing

the tensor indices along multiple modes, CSF reduces the storage overhead compared to the COO format. The CSF format has demonstrated better performance in tensor-matrix products than other storage formats like COO. The shortcomings of CSF include its dependency on the tree height, overhead of tree construction and complexity. The depth of the tree in the CSF storage format is determined by the order of the tensor. As the order increases, the depth of the tree also increases, potentially leading to increased storage overhead and complexity in traversing the tree. Furthermore, the CSF format is more complex than other simpler formats like COO. Implementing and maintaining algorithms based on CSF can be more difficult due to the hierarchical and mode dependent structure of the format.

Li et al. [**?**] then proposed the Hierarchical Coordinate (HiCOO) storage format, which exploits the hierarchical structure of sparse tensors and can be considered as an extension to the CSB storage. However, HiCOO uses smaller blocks, which are more suitable for sparse tensors. To convert a COO tensor to an HiCOO tensor, the non-zeros of a COO tensor are first sorted according to the Z-curve using a variation of quick sort. The sorted tensor is subsequently partitioned into sparse tensor blocks according to the given block size while recording the block pointers simultaneously. The COO indices are lastly compressed into sparse blocks. The coordinates of the elements within a block are relative to the block coordinates, while the blocks store the actual coordinates of the non-zero values in the tensor. Using the relative coordinates of a respective block to store each element, reduces the storage overhead, as the coordinates can be represented with short integers. With the usage of blocks, tensor operations such as tensor-matrix multiplications (TMM) or the matricised tensor times Khatri-Rao product (MMTTKRP) [**?**] can be parallelised and it improves data locality for tensor algorithms and reduces memory bandwidth for tensor access. This approach provides considerable improvements in storage and computational efficiency compared to CSF. As HiCOO is a derivation of COO and stores all non-zero indices, it may not be suitable for tensors with index changes predominantly for a single mode, as it does not have compression for fibres with a small number of non-zeros, leading to potential memory overhead.

Chapter 3

Format Options for Efficient Storage of Sparse Tensors

3.1 MoSk-IF: Incremental Compressed Sparse Fibres with Mode Skipping

The incremental component of MoSk-IF functions similarly to the incremental compressed row storage for matrices introduced by Koster [**?**], while the mode skipping feature is derived from the bi-directional incremental compressed sparse rows by Yzelman and Bisseling [**?**]. The non-zero values of the tensor are stored in a separate array, with each mode possessing an array that stores the increments. If a change triggers an overflow, signifying an index change in the next mode, the algorithm transitions to the corresponding incremental array.

The sign of the increment determines whether to switch to the next mode or skip it. If the sum of all previous increments exceeds the dimension size and the current increment has a positive sign, indicating a positive overflow, the algorithm examines the incremental array of the subsequent mode. Conversely, a negative sign in the current increment indicates a negative overflow and prompts the algorithm to skip the

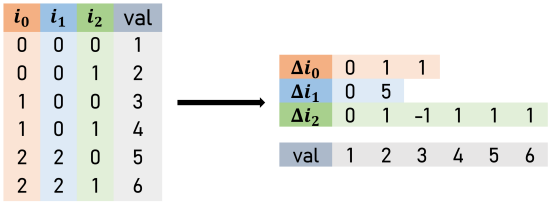

Figure 3.1: Visual representation of converting a $3 \times 3 \times 2$ tensor from COO format to MoSk-IF.

next mode. For instance, in a fourth-order tensor with a conventional mode sequence, a negative overflow in the fourth mode's incremental array would necessitate a lookup in the incremental array of the second mode.

To encode the indices of a tensor from the COO storage to the MoSk-IF storage, Algorithm 1 is employed. The inputs are the indices of the sparse tensor in COO format, typically stored as a two-dimensional array. The algorithm initializes $\Delta A \in \mathbb{Z}^d$ with the indices of the first non-zero entry. # For each row in the COO storage, the difference between the current and previous non-zero indices $i_r - i_{r-1}$ is calculated, producing the increment Δi.# Given that the input COO only stores the non-zero indices of the tensor and the non-zero values are stored in a separate array, the algorithm starts from the mode stored in the last column and works its way to the mode stored in the first column of the COO format. For each mode, the algorithm checks whether there exists an increment and simultaneously checks for any increments in the second next mode. If there is only an increment at the current mode j, it is added to the corresponding increment array and the algorithm continues to the next non-zero entry. If there is an increment at the next mode as well, we add the increment plus dimension size n_j to the respective increment array. If there is no increment in mode $j+1$ but in mode $j+2$, a negative sign is added to $\Delta i_j + n_j$ and stored in the increment array of mode j. For the modes corresponding to the first two columns in the COO format, the increments and overflow are computed separately. Figure 3.1 illustrates an example of MoSk-IF applied to a third-order tensor of size $3 \times 3 \times 2$.

To decode the indices from the MoSk-IF storage, the algorithm starts at the lowest level mode and computes the sum of the index and increment. If the sign of the increment is positive and the sum does not exceed the dimension size n_j, the increment is simply added to the index. If the sum exceeds the dimension size of the respective mode, the algorithm checks the increment array of mode $j+1$ and adds $\Delta i_j - n_j$ to the index. If the sign of the increment is negative, $\Delta i_j - n_j$ is added to the index, and the increment at the second next increment array is checked. Due to the similarity to the CSF [**?**] and ICRS [**?**] storage format and MoSk-IF is mode dependent. The overflow increments of each mode array indicate an increment in the array of the following modes. The only method to alter the access order of the modes is to switch their index arrays before constructing the storage format.

3.2 Ind-IF: Incremental Sparse Fibres with Index Identification

The Ind-IF storage format employs index increments along with index identification to determine the location of increments. Since identification arrays may lead to memory overhead, they are utilized only for tensor modes with a ratio of index changes q_j less than or equal to 0.5. For modes with a ratio exceeding 0.5, the increments Δi are

Algorithm 1 COO to MoSk-IF

Input: Input indices in COO format
Output: ΔA

initialise ΔA_j for $j = 0, ..., d-1$
store indices of COO $i_{0,j}$, $j = 0, ..., d-1$ in ΔA
for $r = 0, 1, ..., nnz_{\mathcal{A}}$ **do**
  compute increments $\Delta i \leftarrow i_r - i_{r-1}$
  $dim \leftarrow d - 1$
  while $dim > 1$ **do**
    if $\Delta i_j \neq 0$ and $\Delta i_l = 0, l \neq j$ **then**
      add Δi_j to ΔA_j
      $dim \leftarrow dim - 1$
    end if
    if $\Delta i_{j+2} \neq 0$ and $\Delta i_l = 0, l \neq j+2$ **then**
      add negative overflow $-(\Delta i_j + n_j)$ to ΔA_j
      $dim \leftarrow dim - 2$
    end if
    if $\Delta i_{j+1} \neq 0$ and $\Delta i_l = 0, l \neq j+1$ **then**
      add overflow $\Delta i_j + n_j$ to ΔA_j
      $dim \leftarrow dim - 1$
    end if
  end while
  if $\Delta i_1 \neq 0$ or $\Delta i_0 \neq 0$ **then**
    add increment or overflow to ΔA_j
  end if
end for

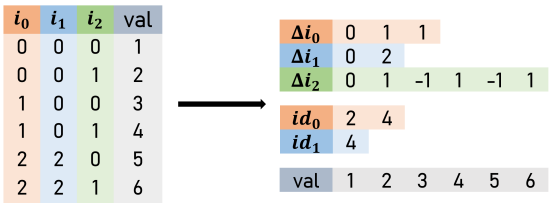

Figure 3.2: Visual representation of converting a $3 \times 3 \times 2$ tensor from COO format to Ind-IF.

stored directly, including $\Delta i = 0$. As a result, not all modes have an additional array for storing the position of increments. This approach ensures that the storage of a mode's increments does not surpass $nnz_{\mathcal{A}}$. The tensor values are stored in a separate array, as illustrated in Figure 3.2 for a $3 \times 3 \times 2$ tensor.

The conversion of a tensor from COO to Ind-IF format is outlined in Algorithm 2. After initialising the increment arrays, the ratio of index changes $q_j 1/nnz_{\mathcal{A}} \sum_{j=0, j\neq k-1}^{d-1} \mathbb{1}_{\Delta i_j \neq 0}$ is calculated for each mode. If $q_j \leq 0.5$, an identification array $id_{\mathcal{A},j}$ is initialised, along with a flag indicating the presence of an identification array for mode j. Initially, the indices of the first non-zero element are added to ΔA, but not to $id_{\mathcal{A}}$, as the first entries represent the offset from which indices should begin.

Subsequently, the algorithm iterates over each non-zero entry in the COO storage format and the increments Δi. Each dimension is checked for the flag for $id_{\mathcal{A},j}$. If mode j has an identification array $id_{\mathcal{A},j}$ and $\Delta i_j \neq 0$, the increment is appended to the end of ΔA_j, and the position of the current non-zero entry is added to the end of $id_{\mathcal{A},j}$. Conversely, if mode j does not have an identification array, we simply add the increment to ΔA_j and continue to the next mode.

To decode the indices of a non-zero element from the Ind-IF format, it is first determined whether the current mode has an identification array. If it does, the position of the increment Δi_j is retrieved from the identification array $id_{\mathcal{A},j}$ and the increment at that position in ΔA_j is added to the index. If a mode does not have an identification array, we directly add ΔA_j to the index. In contrast to MoSk-IF, the Ind-IF storage does not maintain any overflows indicating an index change in the subsequent mode, allowing access to the increment arrays and identification arrays in a different order without modifying the sparse tensor structure. The only requirement is that if the increment arrays are reordered, the flags of the corresponding increment identification arrays must also be updated accordingly. Consequently, the Ind-IF storage format is mode-independent.

Algorithm 2 From COO to Ind-IF

Input: Input indices in COO format
Output: $\Delta A, id_{\mathcal{A}}$

initialise ΔA_j for $j = 0, ..., d-1$

for each dimension $j = 0, ..., d-1$ **do**
  $q_j \leftarrow \frac{1}{nnz_{\mathcal{A}}} \sum \mathbb{1}_{\Delta i_j \neq 0}$ //explain that in Ind-IF section
  if $q_j \leq 0.5$ **then**
    add $id_{\mathcal{A},j}$ to id
    add flag for $id_{\mathcal{A},j}$
  end if
end for

store indices of COO $i_{0,j}$ in ΔA_j and set $id_{\mathcal{A},j} = 0$, for $j = 0, ..., d-1$
for $r = 0, 1, ..., nnz_{\mathcal{A}}$ **do**
  compute increments $\Delta i = i_r - i_{r-1}$
  add Δi_{d-1} to ΔA_{d-1}
  for each mode $j = 0, ..., d-1$ **do**
    if flag for $id_{\mathcal{A},j}$ exists **then**
      if $\Delta i_j \neq 0$ **then**
        add Δi_j to ΔA_j
        add row to $id_{\mathcal{A},j}$
      end if
    else
      add Δi_j to ΔA_j
    end if
  end for
end for

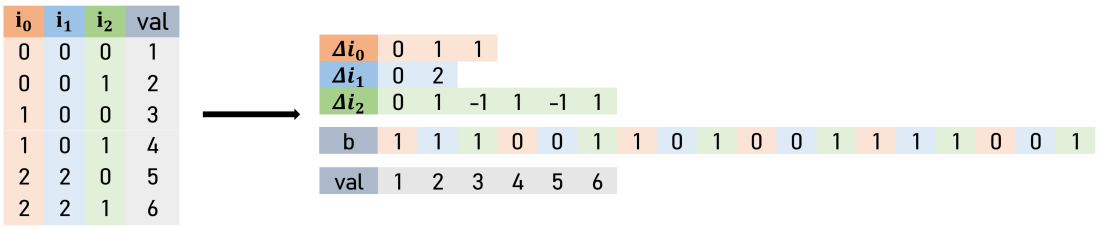

Figure 3.3: Visual representation of converting a $3 \times 3 \times 2$ tensor from COO format to Bit-IF.

3.3 Bit-IF: Incremental Sparse Fibres with Bit Encoding

Instead of adding an identification array to the increment array of modes with $q_j \leq 0.5$, we introduce a single additional array b of size $d \times nnz_{\mathcal{A}}$ that stores zeros and ones. In this context, a zero implies no changes in the current mode, while a one prompts an increment. This array b is called the bit encoding array, where each non-zero value is associated with a sequence of bits of size d. Figure 3.3 illustrates an example of this method applied to a $3 \times 3 \times 2$ tensor.

Algorithm 3 From COO to Bit-IF

Input: Input indices in COO format
Output: $\Delta A, b_{\mathcal{A}}$

initialise ΔA_j for $j = 0, ..., d-1$
store indices of COO $i_{0,j}$ in ΔA_j and set $b_j = 1, j = 0, ..., d-1$
for $r = 0, 1, ..., nnz_{\mathcal{A}}$ **do**
  compute increments $\Delta i = i_r - i_{r-1}$
  for each mode $j = 0, ..., d-1$ **do**
    if $\Delta i_j \neq 0$ **then**
      add Δi_j to ΔA_j
      add 1 to $b_{\mathcal{A}}$
    else
      add 0 to $b_{\mathcal{A}}$
    end if
  end for
end for

The transformation process from COO to Bit-IF is detailed in Algorithm 3 and bears resemblance to Algorithm 2. Upon initializing ΔA and b, the indices of the first non-zero element are added to ΔA and a sequence of ones is incorporated into b. For each

non-zero entry from COO, the increment Δi is calculated. Each mode j is examined to identify if an increment exists. If so, Δi_j is appended to ΔA_j, and a one is added to the end of b. Otherwise, a zero is inserted at the end of b, and the iteration proceeds to the next step.

To decode the indices of a non-zero element from the Bit-IF format, each bit in the corresponding bit sequence is examined. If the bit of a mode is one, the respective increment is added to the index. If it is zero, we continue to the next bit in the sequence. Analogous to the Ind-IF format, the Bit-IF format also maintains index increments. To access the modes of the tensor in a different order, one can directly access the increment arrays in the desired order. This alteration does not impact the tensor structure, provided that the corresponding bits are accessed in the same order. Consequently, the Bit-IF format remains mode-independent.

Chapter 4

Tensor-Vector Multiplication

In this section, the tensor-vector multiplication (TVM) algorithm is discussed with respect to both lexicographical tensor traversal and arbitrary tensor traversal methods. While the Z-Curve and Hilbert Curve are the primary focus of this thesis, it should be noted that they are merely two examples of arbitrary traversal curves, and the algorithms presented here are applicable to any other traversal curve as well.

4.1 MoSk-IF

The kth mode TVM algorithm for the MoSk-IF storage format sorted in lexicographical order is described in Algorithm 4.# In the following algorithms concerning the TVM, we apply the indexting starting from 0, i.e. the kth mode and kth index is referred to as $k-1$ in the algorithms.# Given that the increment arrays are mode dependent, it is assumed that the increment arrays are organized w.r.t. kth order, where the corresponding increment array is both the final and the largest.

The current indices are determined by decoding the increment arrays (refer to Section 3.1). Starting from the final mode, specifically the kth mode, the increment is added to the sum. If the sum does not surpass the dimension size, the kth index is updated with the increment and the loop is terminated. Otherwise, an overflow occurs and the remaining modes are checked using the sign of the increment. When the increment is positive, the dimension is reduced by one following the update of the sum, increment counter, and index. Conversely, if the increment is negative, the procedure remains the same, but the dimension is decreased by two instead of one. For each case, the sum of increments and the increment counter are updated, and the indices are returned upon exiting the loop. Subsequent to exiting the loop, the kth index is employed to calculate the tensor-vector product of the current non-zero value, which is then applied to val_B. The increments are subsequently recalculated using the indices from the preceding outer iteration and these are utilized to update ΔB.

Algorithm 4 TVM algorithm for MoSk-IF

Input: $\Delta A, val_{\mathcal{A}}, \mathbf{v}$
Output: $\Delta B, val_{\mathcal{B}}$

for each Δi_{k-1} in ΔA_{k-1} **do**
  if $i_{k-1} + abs(\Delta i_{k-1}) < n_{k-1}$ **then**
    $i_{k-1} += \Delta i_{k-1}$
    continue
  end if
  for $j = d-1, ..., 0$ and $j \neq k-1$ **do**
    if $\Delta i_j > 0$ **then**
      $i_j += \Delta i_j - n_j$
      $j -= 1$
    else if $\Delta i_j < 0$ **then**
      $i_j += \Delta i_j - n_j$
      $j -= 2$
    end if
    if $j = 0$ **then**
      $i_0 += \Delta A_0 - n_0$
    end if
  end for
  if $\Delta i_{j \neq k-1} \neq 0$ **then**
    $\Delta i = i_i - i_{i-1}$
    update ΔB with Δi according to Alg. 1
    $val_{\mathcal{B}}[i_{k-2} + \Delta i_{k-2}] = \mathbf{v}(i_{k-1}) \cdot val_{\mathcal{A}}(i_{k-1})$
  else
    $val_{\mathcal{B}}[i_{k-2}] += \mathbf{v}(i_{k-1}) \cdot val_{\mathcal{A}}(i_{k-1})$
  end if
end for

A notable constraint of the MoSk-IF storage format is its incapacity to accommodate arbitrary tensor traversal. This limitation arises due to the utilization of negative increments to signify mode skipping rather than reverting to lower indices, rendering the storage of decreasing indices unattainable.

4.2 Ind-IF

The Ind-IF tensor-vector multiplication (TVM) in Algorithm 5 assumes that the increment arrays are ordered for a lexicographical traversal with respect to the kth mode for $\mathcal{A} \times_k \mathbf{v}$. To ensure that, the COO format must be sorted initially, prior to constructing the Ind-IF tensor storage.

A temporary variable is employed to store the temporary sum of the tensor-vector element product, denoted as T, in cases where only the kth index changes.

To determine whether there exists an identification array for one of the modes in $\mathcal{B}$, we compute the number of increments q_j using ΔA_j and $nnz_{\mathcal{B}}$: $q_j = 1/nnz_{\mathcal{B}} \sum_{r=0}^{nnz_{\mathcal{B}}-1} \mathbb{1}_{\Delta i_j \neq 0}$ #. Upon initializing the temporary variable T, a loop iterates through every increment of ΔA_{k-1} as it shares the same length as $nnz_{\mathcal{A}}$. The kth index is updated by adding the increment to i_{k-1}. Following the update of the temporary variable, the remaining modes, excluding the kth mode, are investigated to ascertain the existence of an identification array in conjunction with the increment array. If an identification array exists, it is used to extract the corresponding increment Δi_j from ΔA_j. In the absence of an identification array for mode j, Δi_j is obtained solely from ΔA_j if it is not equal to zero. For every mode j other than the kth mode with an increment $\Delta i_j \neq 0$, the increment is applied to update ΔB_j and $id_{\mathcal{B}}$ if present. Upon examining increments in every mode, the temporary variable T is added to $val_{\mathcal{B}}$ if any of the increments are non-zero.

For arbitrary tensor traversal, a mapping, denoted as $\mathcal{U}_B$, is utilized to associate the indices of the resulting tensor with its TVM value. Relying solely on a temporary value and appending it to $val_{\mathcal{B}}$, whenever an index $i_{j\neq k-1}$ changes, would yield incorrect results. In the case of tensor traversal based on the Z-curve or Hilbert curve, Algorithm 6 employs a map that incorporates the tensor-vector element product into the correct index. The distinction from Algorithm 5 lies in the computation of the new indices using the increments Δi_j. These indices are subsequently used to locate the appropriate position in the map where the TVM value is stored, denoted as i_{key}. The value for i_{key} is then utilised to update $val_{\mathcal{B}}$ with T and to add the indices to $\mathcal{U}_B$ if they have not yet been included. Finally, the map $\mathcal{U}_B$ is used to construct the increment arrays identification arrays using the same approach as Algorithm 2.

Algorithm 5 TVM algorithm for Ind-IF

Input: $\Delta A, id_{\mathcal{A}}, val_{\mathcal{A}}, \mathbf{v}$
Output: $\Delta B, id_{\mathcal{B}}, val_{\mathcal{B}}$

for $j = 0, ..., d-1, j \neq k-1$ **do**
  $q_j \leftarrow \frac{1}{nnz_{\mathcal{B}}} \sum \mathbb{1}_{\Delta i_j \neq 0}$
  if $q_j \leq 0.5$ **then**
    add $id_{\mathcal{B},j}$ to $id_{\mathcal{B}}$
    add flag for $id_{\neq,j}$
  end if
end for

initialise temporary value T
initialise positions ← array(d-1) = 0
for $r = 0, 1, ..., nnz_{\mathcal{A}}$ **do**
  $\Delta i_{k-1} = \Delta A_{r,k-1}$
  $i_{k-1} += \Delta i_{k-1}$
  update $T += \mathbf{v}(i_{k-1}) \cdot val_{\mathcal{A}}(i_{k-1})$
  for $j = 0, ..., d-1, j \neq k-1$ **do**
    if $id_{\mathcal{A}_j}$ exists **then**
      position = positions_j
      if $id_{\mathcal{A}_{j,position}} = r$ **then**
        $\Delta i_j = \Delta A_{j,r}$
        add Δi_j to ΔB_j
        $\text{position}_j += 1$
        if $id_{\mathcal{B}_j}$ exists **then**
          add r to $id_{\mathcal{B},j}$
        end if
      end if
    else
      if $\Delta i_j \neq 0$ **then**
        $\Delta i_j = \Delta A_{j,r}$
        add Δi_j to ΔB_j
        if $id_{\mathcal{B},j}$ exists **then**
          r to $id_{\mathcal{B},j}$
        end if
      else
        $j += 1$
      end if
    end if
  end for
  if any $\Delta i_{j \neq k-1} \neq 0$ **then**
    add T to $val_{\mathcal{B}}$
  end if
end for

Algorithm 6 TVM Ind-IF for arbitrary traversal orders

Input: $\Delta A, id_{\mathcal{A}}, val_{\mathcal{A}}, \mathbf{v}$
Output: $\Delta B, id_{\mathcal{B}}, val_{\mathcal{B}}$

for $j = 0, ..., d-1, j \neq k-1$ **do**
  $q_j \leftarrow \frac{1}{nnz_{\mathcal{B}}} \sum \mathbb{1}_{\Delta i_j \neq 0}$
  if $q_j \leq 0.5$ **then**
    add $id_{\mathcal{B},j}$ to $id_{\mathcal{B}}$
    add flag for $id_{\neq,j}$
  end if
end for

initialise temporary value T
initialise positions $\leftarrow$ array(d-1) = 0
for $r = 0, 1, ..., nnz_{\mathcal{A}}$ **do**
  $\Delta i_{k-1} \leftarrow \Delta A_{r,k-1}$
  $i_{k-1} += \Delta i_{k-1}$
  update $T += \mathbf{v}(i_{k-1}) \cdot val_{\mathcal{A}}(i_{k-1})$
  for $j = 0, ..., d-1, j \neq k-1$ **do**
    if $id_{\mathcal{A},j}$ exists **then**
      position = $positions_j$
      get Δi_j from $id_{\mathcal{A},j}$ and ΔA_j
      $\Delta i_j = \Delta A_{j,r}$
      add Δi_j to ΔB_j
      $position_j += 1$
      if $id_{\mathcal{B}_j}$ exists **then**
        add r to $id_{\mathcal{B},j}$
      end if
    else
      if $\Delta i_j \neq 0$ **then**
        $\Delta i_j = \Delta A_{j,r}$
        add Δi_j to ΔB_j
        if $id_{\mathcal{B},j}$ exists **then**
          r to $id_{\mathcal{B},j}$
        end if
      else
        $j += 1$
      end if
    end if
  end for
  if any $\Delta i_{j \neq k-1} \neq 0$ **then**
    get i_{key} of $i_0, ..., i_{d-2}$
    update $\mathcal{U}_B(i_{key})$ with $i_0, ..., i_{d-2}$ if not contained
    $val_{\mathcal{B}}(i_{key}) = T$
  end if
end for
compute $\Delta B, id_{\mathcal{B}}$ with $\mathcal{U}_B$ according to Alg. 2

4.3 Bit-IF

For the kth mode tensor-vector multiplication (TVM) with the Bit-IF storage format (Algorithm 7), each bit sequence corresponding to a non-zero value is iterated. If solely the kth bit is active, i.e. equal to one, the kth index is updated with the corresponding increment, and the temporary variable is updated with the tensor-vector element product. In cases where the kth bit is zero or other bits are activated, the remaining modes are examined. If the bit of the current mode j is one, the increment associated with this mode is added to the corresponding increment array ΔB_j and a one is appended to the resulting bit encoding array $b_{\mathcal{B}}$. Conversely, if the mode's bit is zero, a zero is added to $b_{\mathcal{B}}$, and the next mode is processed. Following the processing of all modes in the inner iteration, the temporary variable T is incorporated into $val_{\mathcal{B}}$, if any bits of a mode were one.

Algorithm 7 TVM algorithm for Bit-IF with lexicographical traversal

Input: $\Delta A, b_{\mathcal{A}}, val_{\mathcal{A}}, \mathbf{v}$
Output: $\Delta B, b_{\mathcal{B}}, val_{\mathcal{B}}$

initialise temporary value T
for each set b in $b_{\mathcal{A}}$ **do**
  if only $b_{k-1} = 1$ **then**
    $i_{k-1} += \Delta i_{k-1}$
    update $T += \mathbf{v}(i_{k-1}) \cdot val_{\mathcal{A}}(i_{k-1})$
  else if $b_j = 1, j \neq k-1$ or $b_{k-1} = 0$ **then**
    for $j = 0, ..., d-1, j \neq k-1$ **do**
      if $b_j = 1$ **then**
        get Δi_j from and ΔA_j
        update ΔB_j with Δi_j
        add 1 to $b_{\mathcal{B}}$
      else
        add 0 to $b_{\mathcal{B}}$
        continue to next mode
      end if
    end for
    if any $b_{j \neq k-1} = 1$ **then**
      add T to $val_{\mathcal{B}}$
    end if
  end if
end for

The TVM algorithm for arbitrary traversal curves (Algorithm 8) adopts an approach, that is similar to Algorithm 6. The map $\mathcal{U}_B$ is also employed to associate the indices of the resulting tensor with its TVM value. Moreover, if a bit of a mode other than the

kth mode equals one, the index i_j is updated with the increment Δi_j instead of directly adding it to ΔB_j. After obtaining the indices from the inner loop, we compute i_{key} using the updated indices and utilise it to update $val_{\mathcal{B}}$ with T. The set of indices is added to $\mathcal{U}_B$ if not already present. Finally, the Bit-IF storage format for $\mathcal{B}$ is constructed utilising the methods from Algorithm 3.

Algorithm 8 TVM Bit-IF for abitrary traversal orders

Input: $\Delta A, b_{\mathcal{A}}, val_{\mathcal{A}}, \mathbf{v}$
Output: $\Delta B, b_{\mathcal{B}}, val_{\mathcal{B}}$

initialise temporary value T
for each set b in $b_{\mathcal{A}}$ **do**
  if only $b_{i-k} = 1$ **then**
    $i_{k-1} += \Delta i_{k-1}$
    update $T += \mathbf{v}(i_{k-1}) \cdot val_{\mathcal{A}}(i_{k-1})$
  else if $b_j = 1, j \neq k-1$ or $b_{k-1} = 0$ **then**
    for remaining modes j **do**
      if $b_j = 1$ **then**
        get Δi_j from and ΔA_j
        $i_j += \Delta i_j$
      else
        j += 1
      end if
    end for
    if any $b_{j \neq k-1} = 1$ **then**
      get i_{key} of $i_0, ..., i_{d-2}$
      update $\mathcal{U}_B(i_{key})$ with $i_0, ..., i_{d-2}$ if not contained
      $val_{\mathcal{B}}(i_{key}) = T$
    end if
  end if
end for
compute $\Delta B, b_{\mathcal{B}}$ with $\mathcal{U}_B$ according to Alg. 3

Chapter 5

Cost Analysis and Efficiency: A Theoretical Study

5.1 Tensor Storage: Cost Analysis

The analytical costs in Table 5.1 provide an estimate of how much storage each format requires. All three methods use less space than the COO format. Depending on the index changes in the sparse tensor, one increment at a lower dimension could trigger unnecessary overflow entries for the higher dimensions in MoSk-IF, therefore increasing the value of # the ratio of overflowing increments p_j # . Furthermore, these overflows are $\Delta i_j + n_j$, thus MoSk-IF may need long integers to store them if $\Delta i_j + n_j \geq 2^{32}$, exceeding the range of unsigned integers. In such cases, w_{int} in Table 5.1 would be replaced with w_{long} which in turn increases the storage size and makes it less storage efficient. This poses a lesser problem for Ind-IF and Bit-IF, as they store increments and not overflows. In fact, they can be stored with short integers w_{short}, if $\Delta i < 2^{16}$, which reduces the storage size. Ideally, the required storage for w_{bit} would be only one bit, which results in $d \cdot w_{bit} < w_{inc,s}$ and $d \cdot w_{bit} < w_{inc,l}$. If the ratio of index changes q_j and overflows p_j would be larger than 0.5 for all modes, then MoSk-IF may require more storage as COO, as $w_{inc,l} \geq w_{int}$ The similar holds for Ind-IF, as it would eliminate the term $q_j(w_{inc,s} + w_{int})$, resulting in a storage requirement of $nnz_{\mathcal{A}} \cdot (w_{val} + \sum_{j=0}^{d-1} w_{inc,s})$. This storage requirement is close to that of COO, as $w_{inc,s} \leq w_{int}$. In such cases, the Bit-IF storage format would require the least amount of storage.

By quantifying w_{int}, w_{long} and w_{bit}, we can express the storage cost with log_2 statements, which is listed in Table 5.2. The purpose of a log_2 statement is to provide the # number of bits required # to store the increments in a given range. As the tensor values need the same amount of storage for every format, we omitted it from the total cost. Storing increments Δi_j require $log_2(n_j)$ bits in general, where n_j refers to the size of the current dimension. Storing overflows in MoSk-IF requires $log_2(n_j) + 1$ since

$n_j \leq \Delta i_j + n_j < 2 \cdot n_j$. Similar to Table 5.1 the log_2 statements in MoSk-IF are replaced with 32 or 64, depending on the size of the overflow. If the ratio of index changes q_j and overflows p_j are low for each mode, for example between 0.05 and 0.1, then the storage for MoSk-IF would be at least $nnz_{\mathcal{A}}(w_{val} + w_{inc,l})$, while for Ind-IF, it would be $nnz_{\mathcal{A}}(w_{val} + \sum_{j=0}^{d-1} q_j(w_{inc,s} + w_{int}))$. This would mean that Ind-IF is the optimal storage format, followed by Bit-IF and MoSk-IF.

Format	Space
MoSk-IF	$nnz_{\mathcal{A}} \cdot \left(w_{val} + w_{inc,l} + \sum_{j=0}^{d-2} (q_j \cdot w_{inc,s} + p_j \cdot w_{inc,l}) \right)$
Ind-IF	$nnz_{\mathcal{A}} \cdot \left(w_{val} + \sum_{j=0}^{d-1} [\mathbb{1}_{q_j > 0.5} w_{inc,s} + \mathbb{1}_{q_j \leq 0.5} q_j (w_{inc,s} + w_{int})] \right)$
Bit-IF	$nnz_{\mathcal{A}} \cdot \left(w_{val} + d \cdot w_{bit} + \sum_{j=0}^{d-1} q_j \cdot w_{inc,s} \right)$
COO	$nnz_{\mathcal{A}} \cdot (w_{val} + d \cdot w_{int})$

Table 5.1: Analytical storage size of MoSk-IF, Ind-IF, Bit-IF and COO.

Format	Space in bits
MoSk-IF	$nnz_{\mathcal{A}} \left(x_{inc,l} \cdot \left\lceil log_2(n_j) \right\rceil + \sum_{j=0}^{d-2} (x_{inc,s} \cdot q_j + x_{inc,l} \cdot p_j) \cdot \left\lceil log_2(n_j) \right\rceil \right)$
Ind-IF	$nnz_{\mathcal{A}} \left(\sum_{j=0}^{d-1} \mathbb{1}_{q_j > 0.5} (x_{inc,s} \cdot \left\lceil log_2(n_j) \right\rceil) + \mathbb{1}_{q_j \leq 0.5} (x_{inc,s} \cdot q_j \cdot \left\lceil log_2(n_j) \right\rceil + q_j)) \right)$
Bit-IF	$nnz_{\mathcal{A}} \left(d + \sum_{i=0}^{d-1} x_{inc,s} \cdot q_i \cdot \left\lceil log_2(n_j) \right\rceil \right)$
COO	$nnz_{\mathcal{A}} \cdot \sum_{j=0}^{d-1} \left\lceil log_2(n_j) \right\rceil$

Table 5.2: Analytical storage size of MoSk-IF, Ind-IF, Bit-IF and COO using $log_2(n)$ statements, excluding non-zero entry values.

5.2 Tensor-Vector Multiplication: Efficiency Factors

5.2.1 Mode Dependency

Mode dependency considerably impacts efficiency in operations where multiple tensor modes are accessed, such as tensor-vector multiplications applied to multiple modes. In mode-dependent storage formats like MoSk-IF, the index arrays of the modes must be sorted while still in the COO format before constructing the storage format, as the indices decoding start from the last increment array. To obtain indices from a higher increment array, the increments from a lower array must first be decoded. As a result, only the last increment array stores increments that can be directly used to obtain

the corresponding index. Consequently, when performing TVM on multiple modes, the COO format must be resorted multiple times, leading to increased storage costs due to the need for multiple copies of the tensor in the MoSk-IF format with different orderings.

In contrast, mode-independent sparse tensor storage formats, such as Ind-IF and Bit-IF, do not face this issue. Ind-IF stores index increments without any overflows that indicate an index change in the next mode, enabling access to the increment arrays and identification arrays in a different order. Similarly, the Bit-IF format stores index increments along with a bit encoding array, allowing each increment array and its respective bit encodings to be accessed in an arbitrary order. If the tensor modes are accessed in a different order, the index arrays from the COO format do not need to be sorted before constructing the storage format in mode-independent formats. In terms of TVM, it is possible to apply the operation on an arbitrary increment array, not just the last one, as all increment arrays can directly be used to obtain the corresponding indices. This also implies that only one copy of the tensor storage format is needed when applying the TVM on multiple modes.

5.2.2 Traversal Curve

The choice of a traversal curve is an important factor that affects the efficiency and performance of the tensor-vector multiplication (TVM). Sorting the COO entries based on the lexicographical traversal order can improve performance by reducing data movement for index-dependent tensor operations or reducing storage size for certain tensor storage methods. However, depending on the size of the tensor, the whole tensor does not fit in the cache and accessing tensor elements may lead to a high number of cache misses if the non-zero values are placed unfavourably. In such cases, alternative traversal methods may be more suitable. The Z-curve or Hilbert curve are space-filling curves that exploit spatial locality which will boost the efficiency of the TVM.

5.2.3 Blocking

For large tensors that cannot fit entirely within the cache, blocking can be employed during the execution of the TVM. This approach subdivides the tensor into hyper-square blocks and implements MoSk-IF, Ind-IF, or Bit-IF at an intra-block level. In the COO storage format, tensor entries are organized into blocks and traversed according to the Z-curve or Hilbert curve. Within blocks, the traversal is lexicographical and follows a first-order sequence. The total number of blocks is calculated by multiplying the ceiling of each quotient of the dimension size and block size. The initial block order is based on a last-fiber-major arrangement, meaning that blocks are accessed row-wise, then column-wise, and subsequently tube-wise, as illustrated in Figure 5.1. This ordering facilitates the assignment of COO entries to their corresponding blocks,

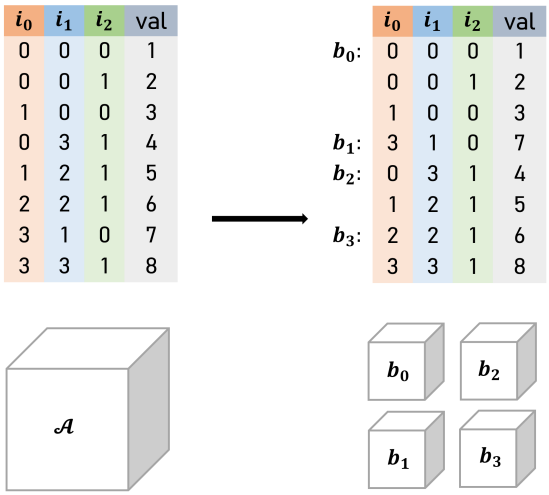

Figure 5.1: Block organization of a $4 \times 4 \times 2$ tensor in Coordinate Format (COO) storage. The block size is set to $\beta = 2$ and the initial traversal curve follows a tube-major ordering. The block number of each entry is computed using the equation: $block_n = pow(2,0)\lfloor i/2 \rfloor + pow(2,1)\lfloor j/2 \rfloor + pow(2,2) \cdot \lfloor k/2 \rfloor$

which can be done by utilising the power of the block size, the mode of the index, and the floor of the quotient of the index and block size to compute the block number for each entry. After each entry of the COO is assigned to its corresponding block, the blocks are sorted according to the chosen traversal curve.

This begs the question of how the block sizes should be chosen, to fully exploit the cache without causing any computational overload. Under the assumption, that every block is non-empty and only holds a few non-zeros in the worst case, iterating over the blocked structure would require a cost of $\Theta(n^d/\beta^d)$, where n is a dimension size, under the assumption that the tensor is hypercubical. If the iteration over all blocks should be scaleable and hence proportional to the number of non-zeros, i.e. $\Theta(\frac{n^d}{\beta^d}) \sim nnz_{\mathcal{A}}$, then the block size would be:

$$\beta = \left(\frac{n_0 \times \ldots \times n_{d-1}}{nnz_{\mathcal{A}}} \right)^{\frac{1}{d}} \tag{5.2.1}$$

5.3 Tensor-Vector Multiplication: Estimated Data Movement

In the subsequent calculations for estimating data movement across the entire tensor, we designate p as the average ratio of overflows for all modes. Additionally, we define q_{k-1} as the ratio of index changes at mode k, and q as the average ratio of index changes encompassing all other modes, with the exclusion of the kth mode. Moreover, we

define a non-major index change as an index alteration that transpires in any mode other than the kth mode.

Prior to examining the data movement, it is essential to first consider the asymptotic complexity of the tensor-vector multiplication (TVM) algorithms. Table 5.3 presents the asymptotic complexity, estimated work, and worst-case upper bounds for each TVM algorithm. The analysis reveals that the worst-case upper bound for all formats is $\mathcal{O}(d \cdot nnz_{\mathcal{A}})$. However, the estimated work exhibits variations across the different formats. MoSk-IF has a more complex expression for the estimated work due to the additional parameters q and p. This complexity may lead to higher computational costs in cases where there is a significant amount of overlapping overflows or increments. The complexity of MoSk-IF is lower than that of COO under specific conditions; for instance, when p and q are both less than 0.5. Conversely, Ind-IF and Bit-IF have identical expressions for estimated work, which suggests that their performance should be comparable under most conditions. The comparison between Ind-IF or Bit-IF and COO will depend on the value of q and the dimension d of the tensor. When $nnz_{\mathcal{A}} + q \cdot nnz_{\mathcal{A}}(d-1)$ is larger $nnz_{\mathcal{A}}$, the complexity of Ind-IF or Bit-IF will surpass that of COO. In contrast, if $nnz_{\mathcal{A}} + q \cdot nnz_{\mathcal{A}}(d-1)$ is less than $nnz_{\mathcal{A}}$, Ind-IF or Bit-IF will have a lower complexity than COO.

Format	Estimated work	Worst case upper bound
MoSk-IF	$\mathcal{O}\big(nnz_{\mathcal{A}}[1 + p \cdot d + q \cdot (d-1)]\big)$	$\mathcal{O}(2 \cdot d \cdot nnz_{\mathcal{A}}) = \mathcal{O}(d \cdot nnz_{\mathcal{A}})$
Ind-IF	$\mathcal{O}\big(nnz_{\mathcal{A}}[1 + q \cdot (d-1)]\big)$	$\mathcal{O}(d \cdot nnz_{\mathcal{A}})$
Bit-IF	$\mathcal{O}\big(nnz_{\mathcal{A}}[1 + q \cdot (d-1)]\big)$	$\mathcal{O}(d \cdot nnz_{\mathcal{A}})$
COO	$\mathcal{O}\big(2 \cdot d \cdot nnz_{\mathcal{A}}\big)$	$\mathcal{O}(d \cdot nnz_{\mathcal{A}})$

Table 5.3: Asymptotic complexity of the TVM algorithm of MoSk-IF, Ind-IF and Bit-IF.

5.3.1 Lexicographical Tensor Traversal

An estimation of the data movements during TVM computation for each storage format is shown in Table 5.4. The following sections analyze the data movement under the assumption that the tensor is traversed lexicographically and the storage formats are sorted accordingly. The COO format is sorted w.r.t. the kth mode, such that the mode on which the TVM is applied is arranged as the last column before the tensor values. The rows of the COO format follow a lexicographical order. These assumptions are necessary for the algorithm to produce the correct resulting tensor.

COO

The easiest way to compute a tensor-vector multiplication with the COO format is to sum tensor-vector element products until a non-major index change occurs. To determine if there is such a change, one could compare the previous $d-1$ indices

Format	Estimated cost for data movement
MoSk-IF	$nnz_{\mathcal{A}}\big(3w_{val} + w_{int} + d \cdot w_{int} + (d-1)w_{int} + (p(d-1)(w_{inc,l} + 2w_{int}) + 2(1-p)w_{int}) + q(5(d-1)w_{int} + 2(d-1)w_{inc,l})\big)$
Ind-IF	$nnz_{\mathcal{A}}\big(3w_{val} + 2w_{int} + q(9(d-1)w_{inc,s} + w_{val})\big)$
Bit-IF	$nnz_{\mathcal{A}}\big(3w_{val} + d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}) + q(2 \cdot w_{inc,s} + w_{val} + (d-1)w_{bit})\big)$
COO	$nnz_{\mathcal{A}}\big(3w_{val} + 2(d-1)w_{int} + w_{int} + q((d-1)w_{int} + w_{val})\big)$

Table 5.4: Data movement estimates for the TVM algorithm using row-major tensor traversal for MoSk-IF, Ind-IF, and Bit-IF. The non-zero entries are sorted according to lexicographical tensor traversal, with the kth mode as the largest increment array.

with the current ones, which results in $2(d-1)$ data element accesses. Computing the product of the current values and adding them to the sum of the TVM element would require two element accesses and one value overwrite for every non-zero entry. The COO format of the resulting tensor can be obtained by copying the columns 0 to $d-3$ of the original COO array and removing any duplicates, given that column $d-2$ is used to store the kth index.

MoSk-IF

Upon entering the *for*-loop in Algorithm 4, the indices are updated, which is a single variable access for each mode. Updating the tensor-vector product values require three element accesses and updates combined. Accessing the increments requires two movements if there is no overflow. Otherwise, with a ratio of p, we would need at most $d-1$ movements for the sum of increments and to check for additional overflows. Similar to the analytic storage cost, long integers may be needed depending on the size of the overflow, which increases the data movement. Comparing the old and new indices require additional reads for $d-1$ elements for each iteration and updating the resulting increment arrays would require at most $d-1$ updates if a non-major index change occurs.

Ind-IF

The Ind-IF algorithm does not require dependent increment arrays, which simplifies the data movement. For each non-zero element, the kth index is updated along with the tensor-vector element product, requiring three movements to access and update the array elements. During a non-major index change with ratio q_j, Algorithm 5 requires one overwrite to add the temporary TVM value to the resulting value array, three movements to update the temporary value, and two accesses to the identification array. For each non-major index change, one element access is needed to retrieve the dimension of the increment, two overwrites to add the new increments and identifications, and three movements to update the counters and current identifications

for at most $d-1$ times. Lastly, to modify the identification of the resulting tensor, one element overwrite is required.

Bit-IF

For each non-zero entry, storing the current bit sequence requires d overwrites and applying the tensor-vector element product to the temporary value requires two accesses and one access and overwrite. In the case of an index change in the k-th mode, one movement occurs to access i_{k-1} and one movement to update i_{k-1}. If a non-major index change occurs, it requires one movement to add the temporary value to the resulting TVM value array $val_{\mathcal{B}}$ and two movements to apply the increment Δi_j to ΔB_j. The temporary value is then updated, which again requires two accesses and one overwrite. Finally, the bits are added to the new bit encoding array, which at most requires $d-1$ overwrites.

Comparison

In terms of data movement during the TVM, the COO format may not necessarily result in the most data movement compared to other storage formats. This depends on the ratio of overflows and non-major index changes, which determine how much data is accessed and overwritten in MoSk-IF, Ind-IF or Bit-IF. Two cases are considered: a very small ratio of overflows and non-major index changes $p, q \to 0$, and a very large ratio of overflows and non-major index changes $p, q \to 1$. Additionally, the contribution of the non-zero tensor values can be neglected, as they are the same for every format in Table 5.4. In the first case, the data movement would reduce to:

- MoSk-IF : $nnz_{\mathcal{A}} \cdot 2(d+1)w_{int}$
- Ind-IF : $nnz_{\mathcal{A}} \cdot 3w_{int}$
- Bit-IF : $nnz_{\mathcal{A}}(w_{int} + d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}))$
- COO : $nnz_{\mathcal{A}} \cdot (2d-1)w_{int}$

and Ind-If would be the most efficient storage format in terms of data movement of the TVM. For the second case, the data movement would reduce to:

- MoSk-IF : $nnz_{\mathcal{A}}\big((9d-8)w_{int} + 3(d-1)w_{inc,l}\big)$
- Ind-IF : $nnz_{\mathcal{A}}\big(2w_{int} + (9(d-1)w_{inc,s} + w_{val})\big)$
- Bit-IF : $nnz_{\mathcal{A}}\big(w_{int} + d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}) + (2w_{inc,s} + w_{val} + (d-1)w_{bit})\big)$
- COO : $nnz_{\mathcal{A}}\big(2(d-1)w_{int} + w_{int} + ((d-1)w_{int} + w_{val})\big)$

for which the cost of data movement would heavily rely on the size of the increments and overflows. For large overflows and small increments, MoSk-IF would require the most data movement to compute the TVM. For high order tensors, Bit-IF would be the most efficient storage format. In both cases the presented storage formats are more efficient than the COO format.

Next, we will determine the values of p and q for which one tensor storage format performs better than the other. Specifically, we will begin by investigating the conditions under which MoSk-IF is superior to Ind-IF. To do this, we can rewrite the equations from Table 5.4 as follows:

$$\begin{aligned} &w_{int} + d \cdot w_{int} + (d-1) \cdot w_{int} + [p(d-1)(w_{inc,l} + 2 \cdot w_{int}) + 2(1-p)w_{int}] + \\ &q(2(d-1)w_{int} + (d-1)(2w_{inc,l} + 3w_{int})) < 2 \cdot w_{int} + q(9(d-1)w_{inc,s} + w_{val}) \end{aligned} \tag{5.3.1}$$

By eliminating the term $nnz_{\mathcal{A}} \cdot 3w_{val}$ and the factor $nnz_{\mathcal{A}}$ from the equation, which are not relevant to the comparison of p and q, and using the approximation $w_{inc,s} \approx w_{inc,l} \approx w_{int}$ and $w_{val} \approx 2 \cdot w_{int}$, we can further simplify Equation 5.3.1.

$$\begin{aligned} & 1 + d + (d-1) + 3p(d-1) + < 2 + q(9(d-1) + 2) \\ & \qquad 2(1-p) + 7q(d-1) \\ \Leftrightarrow \quad & 2d + 3pd + 2 - 5p + 7qd - 7q < 9qd - 7q \\ & \qquad \Leftrightarrow \quad 3pd - 5p < 9qd - 7q - 7q(d-1) - 2d - 2 \\ & \qquad \Leftrightarrow \quad p(3d-5) < 2qd - 2d - 2 \\ & \qquad \Leftrightarrow \quad p < \frac{2(d(q-1)-1)}{3d-5} \end{aligned} \tag{5.3.2}$$

To determine the conditions under which Ind-IF is more efficient than MoSk-IF, we can modify Equation 5.3.2 by replacing < with >. Next, we can look at the values for p and q, where MoSk-IF outperforms Bit-IF. We again rewrite the equation to:

$$\begin{aligned} &w_{int} + d \cdot w_{int} + (d-1) \cdot w_{int} + [p(d-1)(w_{inc,l} + 2 \cdot w_{int}) + (1-p) \cdot 2 \cdot w_{int}] + \\ &q(2(d-1)w_{int} + (d-1)(2w_{inc,l} + 3w_{int})) \\ &< d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}) + q \cdot (2w_{inc,s} + w_{val} + (d-1) \cdot w_{bit}) \end{aligned} \tag{5.3.3}$$

In contrast to the comparison of MoSk-IF and Ind-IF, we cannot eliminate w_{int}, $w_{inc,s}$, or w_{val} as their sizes are significantly larger than w_{bit}. Therefore, we introduce the variable $\epsilon = \frac{w_{bit}}{w_{int}}$ as well as using the approximation $w_{inc,s} \approx w_{inc,l} \approx w_{int}$ and $w_{val} \approx 2 \cdot w_{int}$ again. This results in the following equation:

$$
\begin{aligned}
& 1 + d + (d-1) + 3p(d-1) + < \epsilon d + 2q_{k-1} + q(\epsilon(d-1) + 4) \\
& \qquad 2(1-p) + 7q(d-1) \\
\Rightarrow \quad & 2d + 3pd - 5p - 2 - 7qd - 7q < \epsilon d + 2q_{k-1} + \epsilon qd - \epsilon q + 4q \\
& \Rightarrow \quad 3pd - 5p < \epsilon d + 2q_{k-1} + \epsilon qd - \epsilon q + 4q - 2d + 7qd + 7q + 2 \\
& \Rightarrow \quad p(3d-5) < 2q_{k-1} + qd(7+\epsilon) + q(11-\epsilon) + d(\epsilon - 2) + 2 \\
& \Rightarrow \quad p < \frac{2q_{k-1} + qd(7+\epsilon) + q(11-\epsilon) + d(\epsilon-2) + 2}{(3d-5)}
\end{aligned}
\tag{5.3.4}
$$

By replacing < with >, the condition becomes that Bit-IF is more efficient in terms of data movement than MoSk-IF.

Finally, we compare Ind-IF with Bit-IF and find the value of q such that the TVM with Ind-IF is more efficient than with Bit-IF and vice versa. We approximate q_{k-1} with $q_{k-1} \approx 1$ for simplicity. To find a value of q such that Ind-IF is more efficient in terms of data movement, we rewrite the data movement cost as follows:

$$
\begin{aligned}
& 2 + q(9(d-1) + 2) < \epsilon d + 2q_{k-1} + q(\epsilon(d-1) + 4) \\
\Rightarrow \quad & 2 + 9qd - 9q + 2q < \epsilon d + 2 + \epsilon qd - \epsilon q + 4q \\
\Rightarrow \quad & q(d(9-\epsilon) + \epsilon - 11) < \epsilon d \\
& \Rightarrow \quad q < \frac{\epsilon d}{(d(9-\epsilon) + \epsilon - 11)}
\end{aligned}
\tag{5.3.5}
$$

To determine when Bit-IF is more efficient with regards to data movement, we can replace the < symbol with > in Equation 5.3.5.

5.3.2 Arbitrary Tensor Traversal

The estimated data movement for TVM with arbitrary tensor traversal is shown in Table 5.5. Since the MoSk-IF storage format cannot be applied for an arbitrary tensor traversal TVM, the Ind-IF and Bit-IF remain the sole storage formats available for analysis and comparison to COO. The addition of the current tensor-vector element product to the corresponding indices results in two element accesses and one element overwrite, with a cost of $3 \cdot nnz_{\mathcal{A}} \cdot w_{val}$ for all storage formats. Compared to algorithms with lexicographical traversal tensors, this approach requires the computation of all $d-1$ indices in order to calculate the TVM values. The estimated data movement only contains the computation of the TVM as the construction of Ind-IF or Bit-IF from the map (Section 4) is not part of the TVM.

Format	Estimated cost for data movement
Ind-IF	$nnz_{\mathcal{A}}\big(3w_{val} + 2w_{int} + 7q(d-1)w_{inc,s} + (d-1)w_{int}\big) + nnz_{\mathcal{B}} \cdot q(d-1)w_{int} + \mathcal{W}_{Ind-IF}$
Bit-IF	$nnz_{\mathcal{A}}\big(3w_{val} + d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}) + q(d-1)w_{int} + 2q(d-1) \cdot w_{inc,s}\big) + nnz_{\mathcal{B}} \cdot (d-1)w_{int} + \mathcal{W}_{Bit-IF}$
COO	$nnz_{\mathcal{A}}\big(3w_{val} + w_{int} + 2(d-1)w_{int} + q((d-1)w_{int} + w_{val})\big) + nnz_{\mathcal{B}} \cdot (d-1)w_{int}$

Table 5.5: Data movement estimates for the TVM algorithm using row-major tensor traversal for MoSk-IF, Ind-IF, and Bit-IF. The non-zero entries are sorted according to an arbitrary tensor traversal, with the kth mode as the largest increment array.

COO

The cost of data movement for the TVM with arbitrary tensor traversal is similar to the cost for lexicographical traversal. With the mapping, the entries from the COO can be directly used to construct the key and add the corresponding value, eliminating the need to compare previous and current indices.

Ind-IF

The kth index of the tensor-vector element product can be updated by overwriting one element and accessing another. In the case of a non-major index change, the algorithm requires two accesses to determine the number of modes whose indices are changing and to find their position. Additionally, up to $d-1$ indices can be updated with two element accesses and three element overwrites. Converting the indices to a key that can identify the corresponding tensor value requires $d-1$ element accesses. The average cost of transforming the map $\mathcal{U}_B$ to the Ind-IF storage format is indicated by $\mathcal{W}_{Ind-IF}$.

Bit-IF

To compute the indices for the current bit sequence, we employ the same data movement method as described for the lexicographical traversal in Section 5.3.1. However, unlike the data movement described in Table 5.4, we compute the indices instead of directly updating the bit encoding array and increment arrays. This computation requires one element access and one element overwrite in case of a non-major index change. We then use the computed indices to obtain the key values, which requires $d-1$ element accesses. The cost of transforming the map $\mathcal{U}_B$ to the Bit-IF storage format is indicated by $\mathcal{W}_{Bit-IF}$.

Comparison

As in Section 5.3.1, we can consider the cases of a very small ratio of overflows and non-major coordinate changes, and a very large ratio of overflows and non-major coordinate changes. As the cost $nnz_{\mathcal{B}} \cdot (d-1)w_{int}$ is identical over all storage methods, it can be omitted from the comparison. When $p, q \to 0$, the data movement can be described as follows:

- Ind-IF : $nnz_{\mathcal{A}}\big(2w_{int} + (d-1)w_{int}\big) + \mathcal{W}_{Ind-IF}$
- Bit-IF : $nnz_{\mathcal{A}}\big(d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}) + (d-1)w_{int}\big) + \mathcal{W}_{Bit-IF}$
- COO : $nnz_{\mathcal{A}}\big(w_{int} + 2(d-1)w_{int}\big)$

Based on the observations that $w_{bit} < w_{int}$ and $w_{inc,s} \leq w_{int}$, it seems that Bit-IF requires the least data movement.

- Ind-IF : $nnz_{\mathcal{A}}\big(7(d-1)w_{inc,s} + (d+1)w_{int}\big) + \mathcal{W}_{Ind-IF}$
- Bit-IF : $nnz_{\mathcal{A}}\big(d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}) + 2(d-1)w_{inc,s} + (d-1)w_{int}\big) + \mathcal{W}_{Bit-IF}$
- COO : $nnz_{\mathcal{A}}\big(w_{int} + 3(d-1)w_{int} + w_{val}\big)$

It is evident that the Tensor Virtual Machine (TVM) with the Coordinate Format (COO) storage is the most efficient in terms of data movement. This is due to the fact that the conversion from increment arrays to tensor indices, in the case of non-major index changes, is more costly than simply comparing two rows in the COO storage format.

For simplicity, we use the approximation $\mathcal{W}_{Ind-IF} \approx \mathcal{W}_{Bit-IF}$ and subsequently cancel them out in the following equation. To identify the conditions under which COO is less efficient than Ind-IF or Bit-IF in terms of data movement, we can employ the same analysis outlined in Section 5.3.1. Specifically, we start by identifying the conditions under which Ind-IF is more efficient than COO. By omitting the term $3 \cdot nnz_{\mathcal{A}} \cdot w_{val}$, and utilising the approximation $w_{inc,s} \approx w_{inc,l} \approx w_{int}$ we obtain:

$$\begin{aligned} 2 + 7q(d-1) + (d-1) &< 2(d-1) + 1 \\ \Rightarrow \quad 7q(d-1) &< d-2 \\ \Rightarrow \quad q &< \frac{d-2}{7(d-1)} \end{aligned} \tag{5.3.6}$$

This evaluation is repeated for the comparison between Bit-IF and COO and additionally use the approximation $q_{k-1} \approx 1$ as well as the variable $\epsilon = \frac{w_{bit}}{w_{int}}$:

$$\begin{aligned} \epsilon d + 2 + 2q(d-1) + (d-1) &< 2(d-1) + 1 \\ \Rightarrow \quad 2q(d-1) &< d - 2 - \epsilon d \\ \Rightarrow \quad q &< \frac{d(1-\epsilon) - 2}{2(d-1)} \end{aligned} \tag{5.3.7}$$

From the conditions for q in Equations 5.3.6 and 5.3.7, it can be observed that the numerators are of similar size. The main difference lies in the denominator, where q can be as much as twice as large in Bit-IF as in Ind-IF. This leads to the conclusion that Bit-IF should be preferred over Ind-IF in these situations.

5.3.3 Tensor-Vector Multiplication: Blocking

Similar to a TVM with arbitrary tensor traversal, only Ind-IF and Bit-IF can be employed for an arbitrary block-wise traversal, as there exists the possibility of revisiting previous indices. The estimated data movement required to perform a TVM with an arbitrary block traversal and lexicographical within-block traversal is provided in Table 5.6.

In the case of the COO storage, the data movement remains the same as for arbitrary traversal orders, as discussed in Section 5.3.2. The primary reason for this is that the indices $(i_0, ..., i_{k-2})$ must be compared with the indices in the new COO array to correctly add the tensor-vector element product to the appropriate entry.

Format	Estimated cost for data movement
Ind-IF	$nnz_{\mathcal{A}}\big(3w_{val} + 2w_{int} + 7q(d-1)w_{inc,s} + (d-1)w_{int}\big) + nnz_{\mathcal{B}} \cdot q(d-1)w_{int} + \mathcal{W}_{Ind-IF}$
Bit-IF	$nnz_{\mathcal{A}}\big(3w_{val} + d \cdot w_{bit} + q_{k-1}(w_{inc,s} + w_{int}) + 2q(d-1)w_{inc,s} + (d-1)w_{int}\big) + nnz_{\mathcal{B}} \cdot q(d-1)w_{int} + \mathcal{W}_{Bit-IF}$
COO	$nnz_{\mathcal{A}}\big(3w_{val} + w_{int} + 2(d-1)w_{int} + q((d-1)w_{int} + 2w_{val})\big) + nnz_{\mathcal{B}} \cdot q(d-1)w_{int}$

Table 5.6: Data movement estimates for the TVM algorithm using row-major tensor traversal for MoSk-IF, Ind-IF, and Bit-IF. The non-zero entries are sorted based on an arbitrary tensor traversal for inter-block and lexicographical tensor traversal for intra-block, with the kth mode as the largest increment array.

Ind-IF and Bit-IF

Given the blocked Ind-IF format as the input tensor, Algorithm 6 iterates over every block and the entries of the increment arrays within each block. The product of the number of blocks and the number of entries within a block is equivalent to the number

of non-zero entries. For each entry, the identification array is examined to determine if an index change occurs, and if so, the corresponding increments are added to the index. The temporary value is used to store the tensor-vector element product of entries with the same non-major indices. In case the non-major indices remain the same, only three element accesses are needed. In the event of a non-major index change, the same procedure as in Algorithm 6 is followed.

The same concept is applied to the Bit-IF storage format, which explains the lack of significant changes in the data movement estimation and its similarity to the data movement estimation for the TVM with arbitrary tensor traversal.

Comparison

We examine the conditions under which Bit-IF outperforms Ind-IF by determining the requisite values for q. By employing the approximations presented in Section 5.3.1 and 5.3.2 and removing the term $nnz_{\mathcal{B}} \cdot q(d-1)w_{int}$ from both cost estimations, the following equation is derived:

$$\begin{aligned} \epsilon d + 2 + 2q(d-1) + (d-1) &< 2 + 7q(d-1) + (d-1) \\ \Rightarrow \quad -5q(d-1) &< -\epsilon d \\ \Rightarrow \quad 5q(d-1) &> \epsilon d \\ \Rightarrow \quad q &> \frac{\epsilon d}{5(d-1)} \end{aligned} \tag{5.3.8}$$

Under these conditions, Bit-IF demonstrates greater efficiency than Ind-IF, provided that q exceeds a specific value.

5.4 Choosing the Optimal Storage Format

In terms of the required storage space, MoSk-IF and Bit-IF appear to require the least storage, depending on the size of w_{bit} and $w_{inc,l}$. The Ind-IF storage format requires additional storage to store the increment identifications. When considering the space required in bits, Bit-IF appears to be the best option. In terms of estimated work, all three storage formats have an upper cost of $\mathcal{O}(d \cdot nnz_{\mathcal{A}})$, making none of them inherently better or worse. When analyzing the estimated cost for data movement, MoSk-IF cannot be used for tensor-vector multiplication with arbitrary or blocked traversal, leaving Ind-IF and Bit-IF as possible options. For large vectors, Bit-IF may be more efficient as it requires less storage than integers. Ind-IF may be more efficient in terms of data movement but only if $q < \epsilon d/5(d-1)$. If we use a small ϵ for example, $\epsilon = 0.1$, the value of q needs to be smaller than $d/50(d-1)$ in order for Ind-IF to be more efficient than Bit-IF. This scenario would be true for sparse tensors with a low

order and size, and minimal non-major index changes. Based on these observations, Bit-IF appears to be the most efficient storage method for sparse tensors, at least theoretically.

Chapter 6

Implementation Details and Experiments

6.1 Implementation Details

6.1.1 Construction of Bit-IF and Blocking Bit-IF

The Bit-IF storage format and the Tensor-Vector Multiplication (TVM) have been implemented in C++. The transformation of the tensor from the COO storage to the Bit-IF storage format adheres to the process described in Algorithm 3. Tensor values are represented as doubles, while increments are stored as integers. For the bit encoding array, we employed the dynamic bitset from the Boost library [**?**]. The COO format utilises the vector container from the standard C++ library [**?**] for storage, whereas the Bit-IF format relies on pointer arrays. To facilitate tensor-vector multiplication across all modes, we implemented the arbitrary tensor traversal outlined in Section 4.3. This approach circumvents the need for sorting the COO format in lexicographical order for each mode, which would otherwise result in computational overhead. Subsequently, the output tensor is stored in the Bit-IF storage format, consistent with the implementation of Algorithm 3.

To store the blocking version of Bit-IF, we initially sort the entries of the COO into blocks. The total number of blocks is calculated by obtaining the product of the ceiling of each dimension size divided by the specified block size: $\lceil n_j/\beta \rceil$. The block number for each entry is computed by raising the block size to the power of the mode j, multiplying the result by $\lfloor i_j/\beta \rfloor$, and summing the obtained products. Once each entry is sorted into its corresponding block, these blocks are subsequently sorted concerning the Z-curve or Hilbert curve. Following this, for each block, we encode the non-zero entries within the block from the COO to the Bit-IF storage format according to Algorithm 3.

6.1.2 Computing the Z-Curve Order and Hilbert Order

The Z-curve order for each block is calculated using the bit representation of the block number. First, the indices of the block number are computed. Subsequently, the binary representation of these indices is determined [**?**]. The binary representations of the indices are then interleaved, beginning with the most significant bit, followed by the next most significant bits, and continuing until all bits are combined. This process results in a one-dimensional sequence of bits representing the Z-order of the coordinates. Finally, the binary representation of the Z-order is converted to an unsigned integer to obtain the Z-order of the block.

To obtain the Hilbert number of a spatial coordinate, this thesis employs the method introduced by John Skilling [**?**], which also involves using the binary representation of the indices. Given the coordinates of a block, the Gray code of each index is computed using their binary representation. The Gray codes are then interleaved bit by bit to create the combined Gray code, which represents the Hilbert order. The Gray code of the Hilbert order is then converted back to the binary representation, to obtain the final number of the Hilbert order.

6.1.3 Constructing $\mathcal{U}_B$

As the tensor is not in a lexicographical order for every mode, it is necessary to efficiently arrange the resulting tensor values according to their respective indices. To store the indices and values of $\mathcal{B}$, a two-dimensional integer vector and a one-dimensional double vector from the C++ standard library are employed [**?**]. The initial approach involved searching the indices container each time a new set of indices was computed. If the set of indices is found in the container, the temporary product value would be added to the corresponding position in the value vector. Otherwise, the set of indices and temporary product value would be appended to the end of their respective vectors.

However, the complexity of finding an element in the vector is linear, i.e., $\mathcal{O}(n)$. Consequently, if this process must be executed for every newly computed set of indices during the TVM, the complexity becomes $\mathcal{O}(q \cdot nnz_{\mathcal{A}} \cdot nnz_{\mathcal{B}})$, which is quadratic if $q \to 1$ and $nnz_{\mathcal{A}} \approx nnz_{\mathcal{B}}$. This resulted in a highly inefficient TVM computation. For instance, the run time exceeded two minutes for a fifth mode TVM with the LBNL-Network. Likewise, the third mode TVM of the NIPS tensor took nearly ten minutes to complete, and the TVM of larger tensors would not finish even after an hour.

To address this issue, an additional unordered map from the C++ standard library was incorporated [**?**], which stores both the hash key values for each set of indices and the corresponding tensor values. When a new set of indices is computed, the hash key for this set is calculated using a hashing function [**?**]. The unordered map is then inspected to determine whether it already contains the hash key. If it does, the

corresponding tensor-vector product is updated. Otherwise, a new pair is created, comprising the new hash key and the currently computed tensor-vector product. Then, the set of indices is appended to the 2D-vector containing the indices.

After processing all indices, we proceed to iterate through the 2D-vector and compute the hash key for each non-zero index in $\mathcal{B}$. The hash key is then used to extract the corresponding tensor value, which is subsequently appended to the vector of doubles. This approach is significantly more efficient due to the $\mathcal{O}(1)$ complexity of locating an element in the unordered map, resulting in a linear complexity of $\mathcal{O}(q \cdot nnz_{\mathcal{A}})$. The advantages of using an additional unordered map are evident, as the run time for the fifth mode TVM for the LBNL-Network was reduced from over two minutes to approximately 30 milliseconds on the same system. For the blocked representation of Bit-IF, a similar process is followed. We employ a nested version of $\mathcal{U}_B$, which consists of a nested unordered map that checks for each block, i.e., each outer map, if the hash key is contained within the block, i.e., the inner map.

6.1.4 Selecting Blocksizes β and $\beta_{\mathcal{B}}$

The selection of a block size β can profoundly influence the performance of TVM. If β is too small, we may encounter excessive computational overhead and require a larger number of computations to process the input tensor and vector. Conversely, if β is too large and accommodates too many non-zero values within a block, it may not fit into the cache, subsequently increasing overall data movement. Moreover, if the TVM operation is to be executed on multiple processors, a load imbalance between processes and machines could result. Therefore, an optimal β should be small enough to ensure all entries within a block fit into the cache, yet not so small as to cause computational overhead. The same principle applies to $\beta_{\mathcal{B}}$ when constructing the blocked variant of the Bif-IF format for $\mathcal{B}$.

In this thesis, we evaluated three distinct computations for the block sizes β and $\beta_{\mathcal{B}}$. The first method, employing the derivation for the block size from Section 5.2.3 and denoted as β_1, calculates the block size for the output tensor $\mathcal{B}$ as follows:

$$\beta_{1,\mathcal{B}} = \left(\frac{n_0 \times \ldots \times n_{k-1} \times n_{k+1} \ldots \times n_{d-1}}{nnz_{\mathcal{A}}} \right)^{\frac{1}{(d-1)}} \tag{6.1.1}$$

The second block size, denoted as β_2, modifies the first by multiplying β_1 by four and strictly rounding up to the nearest hundreds digit, with $\beta_{2,\mathcal{B}} = \beta_2/2$. Dividing the input block size by a factor for the output block size is motivated by the elimination of the $\beta_{\mathcal{B}}$ computation for each output block, as it would introduce additional computations to the TVM algorithm, potentially causing overhead.

The last block size, denoted as β_3, is determined by dividing the second-largest di-

mension by a scaling factor. For small tensors with less than 70 million non-zero elements, the scaling factor is 100, while for large tensors with more than 70 million non-zero elements, the scaling factor is 1000. The threshold is established based on the distribution of non-zero values among all tested tensors, which range from 1.6 million non-zeros to over 143 million non-zeros. The ratio is then rounded to the nearest 500 digits, with a minimum value of 1000 for β_3. The block size of the output tensor is chosen to be equivalent to the input block size, i.e., $\beta_{3,\mathcal{B}} = \beta_3$.

6.2 Experiments

6.2.1 Setup

To evaluate the effectiveness of each Bit-IF configuration, we utilised the tensors from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [**?**]. We conducted our tests on two distinct CPU architectures: the Intel Xeon Gold 6238T [**?**] and the AMD EPYC 7742 [**?**]. The Intel Xeon Gold 6238T is a high-performance CPU designed for server and workstation applications, featuring a maximum clock speed of 3.7 GHz and up to 22 cores per processor. On the other hand, the AMD EPYC 7742 is a server-class processor, providing 64 cores per socket and a maximum clock speed of 3.4 GHz. All tests were executed with a single thread and on a single core.

Tensor	Order	Dimensions	Non-Zeros
LBNL-Network	5	1 605 × 4 198 × 1 631 × 4 209 × 868 131	1 698 825
NIPS	4	2 482 × 2 862 × 14 036 × 17	3 101 609
Uber Pickups	4	183 × 24 × 1 140 × 1 717	3 309 490
Chicago Crime	4	6 186 × 24 × 77 × 32	5 330 673
VAST 2015	5	165 427 × 11 374 × 2 × 100 × 89	26 021 945
Enron Emails	4	6 066 × 5 699 × 244 268 × 1 176	54 202 099
NELL-2	3	12 092 × 9 184 × 28 818	76 879 419
Flickr	4	319 686 × 28 153 045 × 1 607 191 × 731	112 890 310
Delicious	4	532 924 × 17 262 471 × 2 480 308 × 1 443	140 126 181
NELL-1	3	2 902 330 × 2 143 368 × 25 495 389	143 599 552

Table 6.1: Description of the sparse tensors from FROSTT.

6.2.2 Results

Storage Comparison

The storage requirements for each tensor in the COO and Bit-IF storage formats are displayed in Table 6.2. The Bit-IF format successfully compresses the storage for all tensors. On average, Bit-IF requires 27% less storage than COO when using the same types to store the indices, increments, and non-zero tensor values. Moreover, the

storage can be further compressed by employing short integers for increments instead of regular integers. This adjustment will result in an even greater storage reduction for the Bit-IF format.

Tensor	COO	Bit-IF
LBNL-Network	0.0443	0.0271
NIPS	0.0693	0.0500
Uber Pickups	0.0740	0.0519
Chicago Crime	0.1192	0.0920
VAST	0.6786	0.5273
Enron Emails	1.2115	0.8666
NELL-2	1.4320	1.0753
Flickr	2.5233	1.7804
Delicious	3.1321	2.3754
NELL-1	2.6748	2.0816

Table 6.2: Comparison of storage requirements in Gbits for each tensor using COO and Bit-IF formats. The table shows the storage requirements for each tensor, with Bit-IF format requiring less storage than COO format for all tensors.

Run Time Comparison: β_1, β_2 and β_3

The performance of most tensors listed in Table 6.1 tends to improve with larger block sizes, as illustrated in Tables 6.2.2 and 6.2.2, where the run-time decreases with increasing block size. This trend can be attributed to an increase in the amount of data that can be processed simultaneously as each block can store more entries. As shown in Table 6.5, the smallest blocks contain only a single entry for $\beta_{1,\mathcal{B}}$ for all modes, while for $\beta_{3,\mathcal{B}}$, the smallest block stores 37 non-zero entries for the third mode tensor-vector multiplication (TVM). Consequently, the larger block sizes result in better cache utilisation and reduced cache misses. This improved cache performance can lead to a decrease in run time, as less time is spent waiting for data to be transferred between different levels of the memory hierarchy. Another contributing factor to the improved performance with larger block sizes could be the better exploitation of the processor's architecture, such as vectorisation or instruction-level parallelism. This allows for more operations to be performed simultaneously, resulting in better performance. The run time tables for β_1, β_2 and β_3, as well as the block size tables for the remaining tensors, can be found in Appendix A.

Run time Comparison: Z-Curve and Hilbert Curve

In general, using the Hilbert curve to traverse the blocks of the tensor resulted in superior performance compared to the Z-curve for the majority of tensors and modes during the TVM. A possible explanation for this observation is that the Hilbert curve

NELL-2						
Mode	$\beta_1 = 35$		$\beta_2 = 200$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	65587.0/ 77372.5	55023.4/ 64908.6	27197.9/ 28375.7	25818.7/ 26804.6	19190.6/ 20394.9	19086.5/ 20282.8
2	75347.0/ 90879.2	30857.9/ 41092.2	26067.8/ 27575.9	16744.4/ 17963.0	18278.4/ 19773.9	18123.2/ 19600.8
3	27956.1/ 28127.7	7611.5/ 7783.5	5874.8/ 5904.9	4921.2/ 4949.9	2106.3/ 2129.3	1792.3/ 1815.3

Table 6.3: NELL-2 tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

NELL-2						
Mode	$\beta_1 = 35$		$\beta_2 = 200$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	74208.2/ 91131.4	70295.3/ 86364.4	39958.7/ 42082.8	40239.6/ 42260.5	33301.5/ 35497.3	33434.2/ 35627.6
2	80191.4/ 100496.4	48122.7/ 65827.0	40065.9/ 42744.0	30006.0/ 32385.8	32844.6/ 35511.2	32169.1/ 34826.0
3	35088.1/ 35373.2	14629.6/ 14913.6	11272.8/ 11321.4	9771.2/ 9816.3	3785.2/ 3810.8	3547.2/ 3577.4

Table 6.4: NELL-2 tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

exhibits better spatial locality and more efficient memory access patterns compared to the Z-curve. The Hilbert curve clusters nearby points in the original space more closely together [**?**], while the Z-curve has more discontinuities and less spatial locality preservation due to its simple construction [**?**]. As a result, the Hilbert curve exhibits fewer cache misses and better cache utilisation, leading to a mainly better performance in the tensor-vector multiplication code. Conversely, the Z-curve experiences more cache misses and less efficient cache usage, resulting in inferior performance.

Run time Comparison: COO, Bit-IF and Blocked Bit-IF

In general, the Bit-IF storage format for tensor-vector multiplication outperforms the COO format. However, benchmark results indicate that the blocked implementation of Bit-IF may not always result in improved performance, depending on the specific characteristics of the tensor. There are a few factors that could contribute to the

NELL-2							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	$nnz_{\mathcal{B}}$	1	4	42	1067	9127	75316
	Bits	16	64	672	17072	146032	1205060
2	$nnz_{\mathcal{B}}$	1	9	45	1249	4337	69326
	Bits	16	144	720	19984	69392	1109220
3	$nnz_{\mathcal{B}}$	1	4	1	160	37	4368
	Bits	16	64	16	2560	592	69888

Table 6.5: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the NELL-2 tensor.

observed performance differences.

One factor is the relative size of the tensor dimensions. The blocked Bit-IF TVM tends to perform worse for modes with significantly larger dimension sizes compared to the other modes. For example, Figure 6.1 shows that the first mode TVM with blocking Bit-IF performed worse than with the COO storage for the Chicago Crime tensor and for the second mode TVM, the blocked Bit-IF performed worse than with the non-blocking Bit-IF for the Delicious-4D tensor. A significant difference in dimension size between modes can result in an imbalanced workload. Moreover, blocking may introduce additional overheads, such as extra loop iterations, conditional statements, and memory accesses, to manage the blocks. When dealing with tensors with a large dimension, this overhead can become more noticeable, leading to a larger run time.

Another factor to consider is the distribution of non-zeros within the tensor. If the non-zeros are evenly spread throughout the tensor, they will be distributed uniformly across all blocks, maintaining a balanced workload for all blocks. In contrast, if the non-zeros are concentrated in a few blocks, this concentration can result in load imbalances. In such cases, having a few blocks containing almost all non-zeros is functionally equivalent to using no blocks at all, rendering the blocks ineffective. The additional overhead caused by the blocks will consequently lead to a decline in performance. The run time comparison of the remaining tensors can be found in Appendix B.

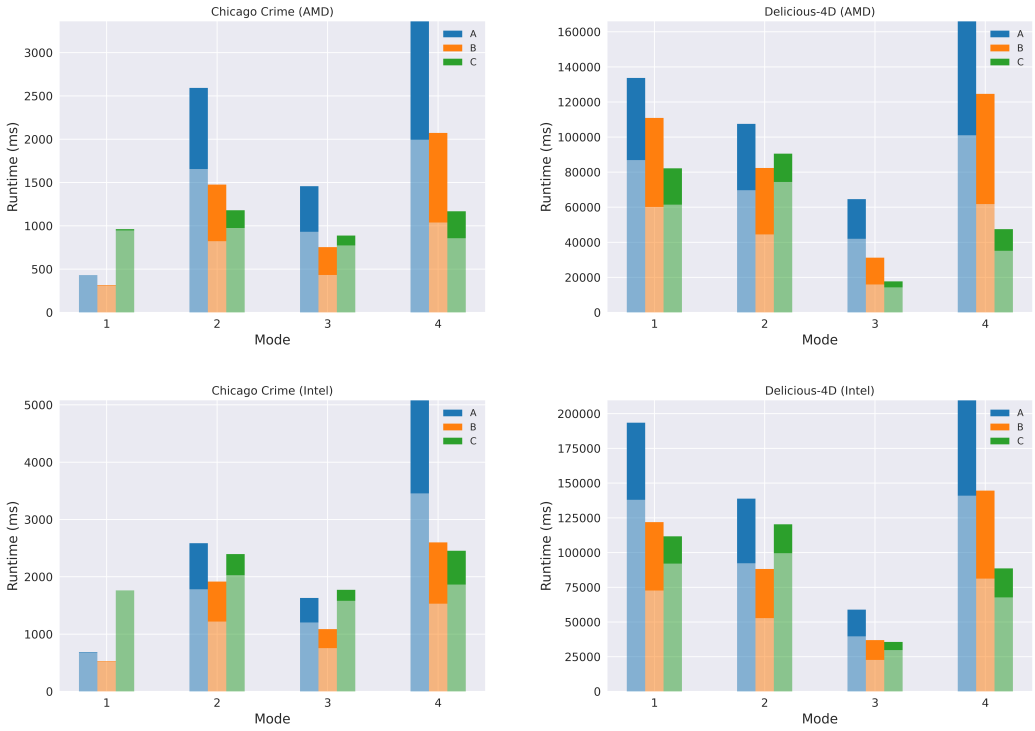

Figure 6.1: The plot illustrates the run times of the TVM with Bit-IF compared to COO for tensors Chicago Crime and Delicious-4D across all modes using the AMD processor (top plots) and Intel processors (bottom plots). The light shaded bar represents the run time for TVM alone, while the dark shaded bar denotes the combined run time of TVM and Bit-IF construction for $\mathcal{B}$. The legends indicate A=TVM with COO, B=non-blocking TVM with lexicographical traversal, and C=fastest blocked TVM computation across all configurations.

Chapter 7

Conclusion

In this thesis, we introduced three novel methods for compressing a sparse tensor from its coordinate (COO) storage format. For each method, we described the encoding and decoding of non-zero entries and proposed an approach to perform tensor-vector multiplication (TVM) using our presented storage formats. Following an extensive theoretical analysis, we implemented the Bit-IF storage format and evaluated its practical efficiency compared to the COO storage format using various configurations.

Multiple configurations of our implementation were tested, and initial concepts were revisited and modified to enhance the performance of our methods. Our experiments and benchmarks yielded promising results, confirming our assumptions that the Bit-IF storage format generally requires less storage space than the COO format and outperforms it in terms of efficiency during TVM. In a few instances, our method exhibited inferior performance, which is most likely attributed to the unfavorable distribution of the tensor's non-zero values. Moreover, the TVM conducted with a tensor in the blocked Bit-IF format typically outperformed the TVM with the non-blocking Bit-IF format, if the block size was optimal. We further observed that the Hilbert curve generally performs better than the Z-curve due to the enhanced locality of reference in memory access patterns provided by the Hilbert curve.

While our proposed methods have already demonstrated promising results, there is potential for improvement. Such improvements include further optimisation of the tensor-vector multiplication (TVM) algorithm, such as refining the process of constructing $\mathcal{U}_B$ or developing an alternative method that does not necessitate $\mathcal{U}_B$. Additional optimization strategies for the TVM involve parallelising the blocked TVM algorithm by employing message passing or other optimisation techniques. The compression from COO to Bit-IF can be further enhanced by storing increments with dynamic types, enabling the storage of small increments with short integers. Adding additional sparse tensor operations to the Bit-IF storage format, such as the CANDECOMP/PARAFAC (CP) or the Tucker decomposition, as well as matricised

tensor times Khatri-Rao Product (MTTKRP), is another possible extension. Finally, adding tensor-matrix multiplication (TMM) or tensor-tensor multiplication (TTM) algorithms for the Bit-IF structure would further extend its capabilities.

By providing a fundamental concept with potential for future work, our thesis aims to stimulate further research in this area. We hope that our idea will continue to be refined and improved upon, revolutionising the field of sparse tensor computations.

Appendix A

Bit-IF Run time Comparison of Different Block Sizes

LBNL-Network						
Mode	$\beta_1 = 473$		$\beta_2 = 1900$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	477.0/ 656.5	428.0/ 588.4	404.8/ 552.7	407.2/ 555.8	420.9/ 567.1	419.3/ 564.0
2	479.5/ 660.6	423.1/ 576.6	407.2/ 553.3	413.7/ 563.3	426.7/ 573.4	426.3/ 571.1
3	479.9/ 657.4	413.6/ 562.5	408.3/ 557.0	409.8/ 562.2	421.6/ 568.7	414.6/ 557.6
4	481.2/ 659.2	425.9/ 580.1	412.5/ 554.4	411.8/ 554.7	433.4/ 580.2	421.7/ 565.0
5	36.3/ 42.2	33.5/ 39.4	28.8/ 31.3	27.2/ 30.2	31.4/ 34.4	28.6/ 31.8

Table A.1: LBNL-Network tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

NIPS						
Mode	$\beta_1 = 27$		$\beta_2 = 200$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	2661.9/ 4054.9	2465.3/ 3867.8	1205.4/ 1459.4	1184.7/ 1437.1	902.1/ 1159.7	906.3/ 1165.7
2	2046.4/ 2654.2	1204.1/ 1804.6	879.0/ 976.7	638.5/ 726.1	539.2/ 637.9	545.7/ 644.7
3	284.1/ 286.2	248.7/ 251.2	135.2/ 136.0	131.5/ 132.4	207.3/ 207.7	208.2/ 208.6
4	2178.6/ 3488.5	1661.7/ 2561.9	886.0/ 1256.1	857.6/ 1180.4	714.3/ 961.1	717.8/ 963.6

Table A.2: NIPS tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Uber						
Mode	$\beta_1 = 7$		$\beta_2 = 100$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	800.9/ 901.8	959.3/ 1068.3	629.7/ 659.8	637.2/ 668.2	1554.4/ 1612.6	1544.6/ 1602.0
2	1003.7/ 1222.2	792.0/ 1006.2	604.5/ 703.4	602.7/ 698.6	1059.0/ 1253.8	1058.2/ 1253.9
3	1195.3/ 1409.1	721.3/ 928.1	551.8/ 603.8	553.9/ 604.4	679.4/ 779.5	677.0/ 777.4
4	1006.2/ 1231.4	531.5/ 723.8	301.1/ 361.7	298.8/ 360.1	389.1/ 504.6	390.7/ 506.8

Table A.3: Uber tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Chicago Crime						
Mode	$\beta_1 = 3$		$\beta_2 = 100$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	1087.2/ 1103.9	946.4/ 961.6	979.0/ 981.5	978.2/ 980.9	954.6/ 957.0	948.6/ 951.0
2	3177.0/ 4398.6	1703.4/ 2651.4	974.6/ 1179.7	982.5/ 1188.4	1571.5/ 1986.6	1549.2/ 1968.2
3	2228.4/ 3136.0	1730.8/ 2594.1	774.5/ 890.0	774.2/ 888.1	1092.5/ 1295.7	1093.1/ 1299.0
4	2446.0/ 3770.8	1896.3/ 2900.1	857.2/ 1171.0	856.5/ 1168.3	1877.9/ 2503.3	1885.7/ 2517.6

Table A.4: Chicago-Crime-Comm-Network tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

VAST 2015						
Mode	$\beta_1 = 17$		$\beta_2 = 100$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	25954.3/ 31574.0	23431.8/ 28628.1	16540.0/ 17488.9	16297.6/ 17238.0	15107.5/ 16706.3	14987.2/ 16571.0
2	20875.2/ 30206.0	10814.9/ 17856.0	7072.4/ 8327.8	4206.6/ 5341.0	5033.8/ 6478.5	5407.8/ 6955.1
3	18262.5/ 26335.0	10476.9/ 16767.9	8156.6/ 11944.2	8180.9/ 11884.6	6362.6/ 8715.7	6423.1/ 8809.4
4	21470.2/ 33789.2	14166.9/ 23664.0	7957.9/ 11155.4	7875.2/ 10963.2	6419.4/ 8695.4	6450.5/ 8756.8
5	21146.4/ 33419.9	14637.3/ 24469.7	7963.0/ 11161.6	7782.7/ 10850.9	6406.6/ 8678.2	6459.5/ 8762.6

Table A.5: VAST-2015-MC1-5D tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Enron						
Mode	$\beta_1 = 116$		$\beta_2 = 500$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	27837.5/ 38449.6	27194.7/ 37154.3	21289.0/ 26918.1	21624.7/ 27183.6	26923.4/ 34677.3	26962.1/ 34705.3
2	17785.4/ 20355.1	15445.1/ 17647.3	10823.2/ 11645.3	10821.9/ 11577.6	12259.9/ 13296.9	12334.2/ 13375.9
3	9211.5/ 9469.9	6596.9/ 6847.5	6869.2/ 6917.8	5831.5/ 5876.9	6618.0/ 6665.3	6269.8/ 6317.6
4	12846.8/ 19201.6	12251.6/ 18372.7	8840.4/ 12025.9	7925.0/ 10720.7	11201.4/ 14888.2	11113.4/ 14849.2

Table A.6: Enron Emails tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Flickr						
Mode	$\beta_1 = 3111$		$\beta_2 = 12500$		$\beta_3 = 16000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	66898.6/ 203484.4	63253.0/ 202766.2	47072.8/ 58939.3	44479.8/ 55219.2	41484.0/ 50938.1	49150.9/ 60220.9
2	32897.3/ 39940.7	23582.7/ 30241.4	26778.0/ 30476.2	21669.0/ 24938.2	25960.8/ 29805.2	25528.8/ 29840.0
3	24347.4/ 30595.3	21127.4/ 27553.4	15521.6/ 18044.0	12874.5/ 15145.0	14720.3/ 16858.5	12160.7/ 14911.7
4	43271.7/ 198562.5	37345.3/ 264324.8	32824.8/ 46924.2	31672.9/ 44542.9	26768.5/ 36636.4	27855.8/ 40799.8

Table A.7: Flickr tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Delicious						
Mode	$\beta_1 = 3915$		$\beta_2 = 15700$		$\beta_3 = 25000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	102272.5/ 127425.2	97929.6/ 121775.0	68234.5/ 116120.2	66899.3/ 120294.6	61442.5/ 82182.8	62716.5/ 82687.5
2	98722.7/ 118196.3	74405.4/ 90530.6	90211.7/ 107302.6	83428.0/ 100249.9	88183.7/ 107021.5	77708.6/ 94973.4
3	40652.4/ 58083.3	32642.2/ 47323.9	17937.3/ 21779.9	16896.8/ 20542.3	22236.3/ 26321.9	14298.1/ 17726.1
4	63351.0/ 92023.4	56908.6/ 83602.7	59666.2/ 87805.5	41811.6/ 59162.1	35163.3/ 47464.8	36617.1/ 54675.9

Table A.8: Delicious 4D tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

NELL-1						
Mode	$\beta_1 = 10337$		$\beta_2 = 41400$		$\beta_3 = 30000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	232530.2/ 336848.9	196802.1/ 298639.7	79917.7/ 126515.8	84555.8/ 150786.4	140122.7/ 253692.3	124540.3/ 220167.8
2	255713.9/ 360665.1	165480.1/ 241926.3	74360.2/ 170084.7	91638.9/ 286785.7	172132.3/ 219516.9	104813.5/ 240576.1
3	127578.9/ 139611.9	46794.0/ 57403.4	79694.6/ 80996.1	29433.3/ 30516.6	108923.1/ 110967.3	26587.8/ 27762.4

Table A.9: NELL-1 tensor benchmark results for the AMD EPYC 7742 processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

LBNL-Network						
Mode	$\beta_1 = 473$		$\beta_2 = 1900$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	829.5/ 1061.8	822.2/ 1051.5	784.7/ 985.9	786.0/ 988.6	771.6/ 957.3	769.5/ 951.3
2	832.7/ 1075.0	820.4/ 1058.7	785.7/ 979.1	786.7/ 980.2	775.6/ 959.8	778.5/ 958.8
3	838.8/ 1075.6	817.2/ 1042.8	781.5/ 981.9	781.7/ 980.8	774.4/ 960.4	768.2/ 950.2
4	848.2/ 1092.0	827.3/ 1062.2	797.9/ 1001.6	792.6/ 990.3	781.6/ 965.8	774.5/ 954.6
5	66.3/ 77.4	60.7/ 71.8	53.0/ 58.0	51.0/ 55.7	57.0/ 61.1	52.0/ 56.0

Table A.10: LBNL-Network tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

NIPS						
Mode	$\beta_1 = 27$		$\beta_2 = 200$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	3609.2/ 5921.9	3247.8/ 5393.3	2155.5/ 2608.2	2089.9/ 2504.0	1560.7/ 1981.6	1718.5/ 2142.1
2	2372.5/ 3301.8	1697.7/ 2518.1	1509.8/ 1668.9	1102.2/ 1241.7	970.8/ 1130.4	983.7/ 1144.9
3	482.2/ 485.9	435.4/ 439.6	251.2/ 252.2	244.0/ 245.1	403.0/ 403.2	402.0/ 402.8
4	3058.6/ 5203.0	2349.3/ 3969.6	1533.8/ 2156.0	1508.5/ 2076.6	1467.6/ 1881.4	1474.5/ 1887.8

Table A.11: NIPS tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Uber						
Mode	$\beta_1 = 7$		$\beta_2 = 100$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	1273.3/ 1408.8	1310.4/ 1446.0	1241.1/ 1297.6	1257.0/ 1311.7	1328.9/ 1372.9	1328.5/ 1372.7
2	1567.2/ 1885.4	1314.9/ 1604.8	1219.2/ 1403.5	1203.6/ 1385.0	1363.8/ 1567.7	1359.2/ 1563.0
3	1481.5/ 1772.1	1143.5/ 1424.9	1120.2/ 1221.7	1101.6/ 1197.7	1004.7/ 1096.4	1005.1/ 1096.8
4	1177.5/ 1484.2	829.1/ 1072.0	615.2/ 727.1	556.7/ 661.5	428.6/ 538.6	430.5/ 540.1

Table A.12: Uber Pickups, Intel Cascade run times are displayed in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Chicago Crime						
Mode	$\beta_1 = 3$		$\beta_2 = 100$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	1956.0/ 1984.7	1929.0/ 1955.8	1768.5/ 1773.6	1754.6/ 1759.4	1767.6/ 1771.7	1772.3/ 1776.5
2	3981.6/ 5778.8	2751.6/ 4328.4	2027.9/ 2396.4	2037.3/ 2409.0	2287.0/ 2699.5	2291.6/ 2705.0
3	3734.7/ 5528.8	2947.6/ 4601.2	1574.8/ 1767.9	1580.9/ 1773.0	1691.9/ 1905.8	1690.5/ 1904.9
4	3613.8/ 5760.9	2806.4/ 4352.6	1864.2/ 2453.5	1896.3/ 2522.4	2250.9/ 2895.2	2246.4/ 2892.2

Table A.13: Chicago Crime tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

VAST 2015						
Mode	$\beta_1 = 17$		$\beta_2 = 100$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	27044.5/ 33927.3	24553.7/ 31073.7	20222.2/ 21875.6	18342.1/ 19869.6	18010.4/ 19813.0	17995.4/ 19798.1
2	25123.0/ 37566.2	16113.5/ 26386.8	10608.4/ 12877.3	8137.9/ 10227.6	8837.0/ 11099.6	8833.6/ 11089.5
3	24433.0/ 35656.2	17022.8/ 26605.3	13804.0/ 20190.5	13686.2/ 19761.4	12931.3/ 17284.0	13093.0/ 17449.5
4	28862.9/ 46059.1	22969.2/ 37752.9	13457.1/ 18620.0	13337.9/ 18292.5	13111.6/ 17211.7	13141.8/ 17239.7
5	29586.2/ 47899.4	23240.2/ 38240.3	13487.3/ 18728.3	13361.7/ 18302.7	13105.1/ 17212.2	13109.6/ 17212.5

Table A.14: VAST-2015-MC1-5D tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Enron						
Mode	$\beta_1 = 116$		$\beta_2 = 500$		$\beta_3 = 1000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	40840.2/ 55961.3	40087.7/ 54485.7	34399.5/ 43417.2	33581.7/ 42200.1	35186.5/ 44502.1	35624.0/ 44950.1
2	24036.2/ 27028.9	20328.3/ 23138.8	19267.1/ 20500.0	18723.8/ 19895.3	19047.6/ 20295.0	19246.7/ 20492.7
3	14369.2/ 14719.4	11725.4/ 12047.2	12461.7/ 12530.9	11561.2/ 11628.4	12043.3/ 12110.5	11332.4/ 11401.6
4	20308.8/ 28992.2	19304.7/ 27458.5	16038.2/ 20680.9	15190.0/ 19630.3	17043.3/ 21698.8	16585.5/ 21221.1

Table A.15: Enron tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Flickr						
Mode	$\beta_1 = 3111$		$\beta_2 = 12500$		$\beta_3 = 16000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	94507.5/ 127527.2	91024.1/ 123786.1	72465.1/ 89826.7	71324.8/ 88125.3	67350.5/ 82867.0	67360.1/ 82760.6
2	46606.2/ 56264.8	34271.3/ 43477.3	37397.0/ 41750.6	32200.1/ 36283.7	36763.7/ 40938.8	32335.7/ 36368.4
3	40162.2/ 49170.7	33123.2/ 41228.1	26542.3/ 29510.3	24092.6/ 26919.5	24369.3/ 27163.5	23046.9/ 25807.5
4	70030.1/ 96173.4	60987.9/ 84130.9	57377.8/ 78236.4	56425.6/ 76127.1	51442.5/ 65996.4	51378.8/ 65727.3

Table A.16: Flickr tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

Delicious						
Mode	$\beta_1 = 3915$		$\beta_2 = 15700$		$\beta_3 = 25000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	154065.8/ 198679.7	141837.9/ 183532.7	100744.7/ 124359.4	98609.7/ 121511.7	92195.9/ 112093.8	92009.6/ 111675.0
2	155832.0/ 187420.5	103667.6/ 132532.7	115397.8/ 137275.9	99543.3/ 120349.8	110471.7/ 131546.3	100838.1/ 121621.5
3	63617.6/ 96040.7	52056.9/ 79637.8	33291.3/ 39182.6	30749.1/ 36257.4	34575.2/ 40577.9	29742.4/ 35661.3
4	109038.0/ 163367.0	98416.1/ 148704.9	76906.3/ 108806.3	75422.1/ 104744.7	67711.8/ 88553.8	67773.1/ 88241.2

Table A.17: Delicious 4D tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

NELL-1						
Mode	$\beta_1 = 10337$		$\beta_2 = 41400$		$\beta_3 = 30000$	
	Z-Curve	Hilbert	Z-Curve	Hilbert	Z-Curve	Hilbert
1	273464.1/ 407255.2	245426.9/ 374741.1	108432.2/ 123824.4	103855.8/ 118095.8	104659.1/ 117925.7	100211.1/ 113048.0
2	225600.4/ 255395.7	104759.3/ 128532.8	108852.4/ 124271.5	92905.1/ 106726.8	107508.0/ 121733.3	82884.5/ 96160.8
3	159816.1/ 177727.0	69794.1/ 85850.6	92345.6/ 94641.5	46393.6/ 48047.3	92538.5/ 94739.5	46344.3/ 47963.9

Table A.18: NELL-1 tensor benchmark results for the Intel Xeon Gold 6238T processor, showcasing run times in milliseconds. The first run time measures only the TVM, while the second run time includes both the TVM and Bit-IF construction of $\mathcal{B}$, $\mathcal{W}_{Bit-IF}$.

LBNL-Network							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	936	1	2546	1	2678
	Bits	24	22464	24	61104	24	64272
2	nnz	1	563	1	2574	1	2701
	Bits	24	13512	24	61776	24	64824
3	nnz	1	915	1	2546	1	2678
	Bits	24	21960	24	61104	24	64272
4	nnz	1	563	1	2579	1	2706
	Bits	24	13512	24	61896	3	64944
5	nnz	1	123	19	9626	3	10674
	Bits	24	2952	456	231024	72	256176

Table A.19: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the LBNL-Network tensor.

NIPS							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	44	23	2019	487	85845
	Bits	20	880	460	40380	9740	1716900
2	nnz	1	24	10	1024	66	45009
	Bits	20	480	200	20480	1320	900180
3	nnz	1	3	1	22	409	834
	Bits	20	60	20	440	8180	16680
4	nnz	1	38	1	168	46	37309
	Bits	20	760	20	3360	920	746180

Table A.20: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the NIPS tensor.

Uber							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	27	1	36177	136	354052
	Bits	20	540	20	723540	2720	7081040
2	nnz	1	125	1	66722	134	1105191
	Bits	20	1500	20	1334440	2680	22103800
3	nnz	1	8	1	55512	12205	607735
	Bits	20	160	20	1110240	244100	12154700
4	nnz	1	8	1	58786	504	692030
	Bits	20	160	20	1175720	10080	13840600

Table A.21: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the Uber tensor.

Chicago Crime							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	1	14650	27446	42095	42095
	Bits	20	20	293000	548920	841900	841900
2	nnz	1	8	93	18315	70219	507322
	Bits	20	160	1860	366300	1404400	10146400
3	nnz	1	1	286	14661	44236	279465
	Bits	20	20	5720	293220	884720	5589300
4	nnz	1	8	96	28116	97520	742620
	Bits	20	160	1920	562320	1950400	14852400

Table A.22: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the Chicago Crime tensor.

VAST 2015							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	10	1911	16793	228482	900738
	Bits	24	240	45864	403032	5483600	21617700
2	nnz	1	25	1	3698	1887	161466
	Bits	24	600	24	88752	45288	3875180
3	nnz	1	183	1	486	5	58533
	Bits	24	4392	24	11664	120	1404790
4	nnz	1	65	1	739	5	58532
	Bits	24	1560	24	17736	120	1404770
5	nnz	1	54	1	486	5	58534
	Bits	24	1296	24	11664	120	1404820

Table A.23: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the VAST 2015 tensor.

Enron							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	4405	1	36177	136	354052
	Bits	20	88100	20	723540	2720	7081040
2	nnz	1	1125	1	66722	134	1105191
	Bits	20	22500	20	1334440	2680	22103800
3	nnz	1	64	1	55512	12205	607735
	Bits	20	1280	20	1110240	244100	12154700
4	nnz	1	5099	1	58786	504	692030
	Bits	20	101980	20	1175720	10080	13840600

Table A.24: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the Enron tensor.

Flickr							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	16556	1	33801	1	65913
	Bits	20	331120	20	676020	20	1318260
2	nnz	1	8510	1	821642	52	2290967
	Bits	20	170200	20	16432800	1040	45819300
3	nnz	1	388	1	6250	48	16000
	Bits	20	7760	20	125000	960	320000
4	nnz	1	30552	1	27649	1	61468
	Bits	20	611040	20	552980	20	1229360

Table A.25: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the Flickr tensor.

Delicious-4D							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	121424	1	1405973	1	3496054
	Bits	20	2428480	20	28119500	20	69921100
2	nnz	1	21599	2	4825443	18	13639507
	Bits	20	431980	40	96508900	360	272790000
3	nnz	1	1930	1	56774	3	307011
	Bits	20	38600	20	1135480	60	6140220
4	nnz	1	30539	1	131819	1	750785
	Bits	20	610780	20	2636380	20	15015700

Table A.26: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the Delicious-4D tensor.

NELL-1							
Mode		$\beta_{1,\mathcal{B}}$		$\beta_{2,\mathcal{B}}$		$\beta_{3,\mathcal{B}}$	
		Min	Max	Min	Max	Min	Max
1	nnz	1	35	296	1710	535	3081
	Bits	16	560	4736	27360	8560	49296
2	nnz	1	31	149	1409	881	2957
	Bits	16	496	2384	22544	14096	47312
3	nnz	1	21	175	2165	716	4204
	Bits	16	336	2800	34640	11456	67264

Table A.27: Count of non-zero elements and the required bits (lower bound) in the smallest and largest blocks of $\mathcal{U}_B$ after the kth mode TVM in the NELL-1 tensor.

Tensor	Mode	Lower Bound	Upper Bound
LBNL-Network	1	0.0584	3.0796
	2	0.0585	2.4931
	3	0.0584	3.0592
	4	0.0584	2.5022
	5	0.0450	0.0827
NIPS	1	0.1241	0.9683
	2	0.0925	0.6589
	3	0.0695	0.2746
	4	0.1245	0.9770
Uber	1	0.0792	1.1810
	2	0.0873	2.7686
	3	0.0834	0.6426
	4	0.0838	0.5017
Chicago Crime	1	0.1205	0.6237
	2	0.1634	1.1712
	3	0.1679	1.0308
	4	0.1791	1.3555
VAST 2015	1	0.8624	5.5553
	2	1.0031	6.0988
	3	1.0872	9.9674
	4	1.1889	8.9602
	5	1.2091	9.0244

Table A.28: Comparison of Gbits moved during tensor-vector multiplication. The upper bound includes all touched variables and container elements, while lower bound considers only the touched elements in the Bit-IF storage format.

Tensor	Mode	Lower Bound	Upper Bound
Enron	1	1.7333	94.2846
	2	1.3175	40.6176
	3	1.2228	7.5011
	4	1.5324	60.9283
NELL-2	1	1.7696	11.9098
	2	1.8158	12.8475
	3	1.4399	7.3227
Flickr	1	3.5795	132.7210
	2	2.8747	71.2992
	3	2.8416	31.0421
	4	3.5477	286.2440
Delicious-4D	1	0.6168	4104.9800
	2	0.3670	2222.7100
	3	0.0795	31.5645
	4	0.9503	612.3070
NELL-1	1	0.5948	38.7505
	2	0.7484	39.8272
	3	3.0428	19.9763

Table A.29: Comparison of Gbits moved during tensor-vector multiplication. The upper bound includes all touched variables and container elements, while lower bound considers only the touched elements in the Bit-IF storage format.

Appendix B

Bit-IF and COO Run time Comparison

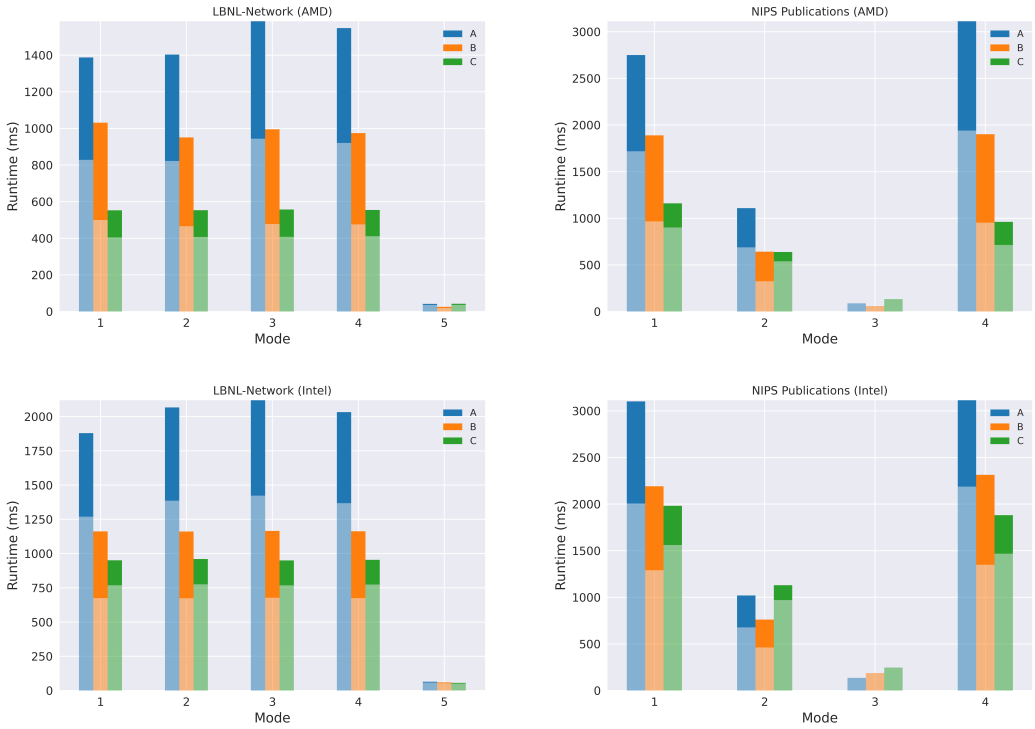

Figure B.1: run time comparison for tensors LBNL-Network and NIPS using the AMD processor (top images) and Intel processors (bottom image). The light shaded bar represents the run time for TVM alone, while the dark shaded bar denotes the combined run time of TVM and Bit-IF construction. Legends: A=TVM with COO, B=non-blocking TVM with lexicographic traversal, C=the fastest blocked TVM computation selected from all configurations.

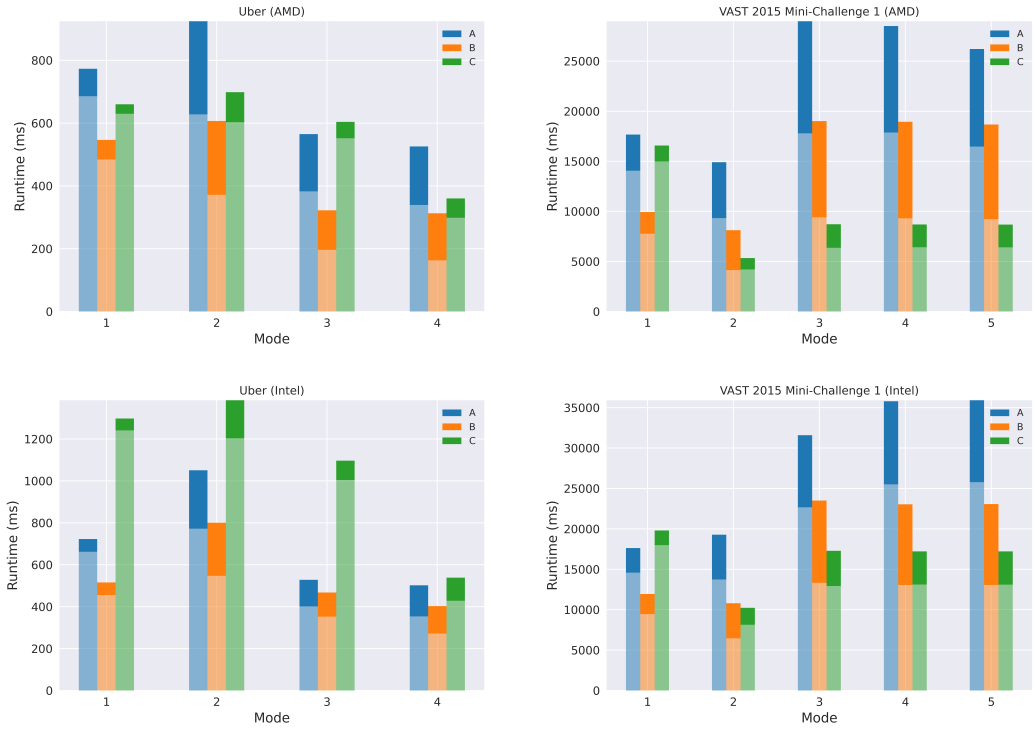

Figure B.2: Run time comparison for tensors Uber and VAST using the AMD processor (top images) and Intel processors (bottom image). The light shaded bar represents the run time for TVM alone, while the dark shaded bar denotes the combined run time of TVM and Bit-IF construction. Legends: A=TVM with COO, B=non-blocking TVM with lexicographic traversal, C=the fastest blocked TVM computation selected from all configurations.

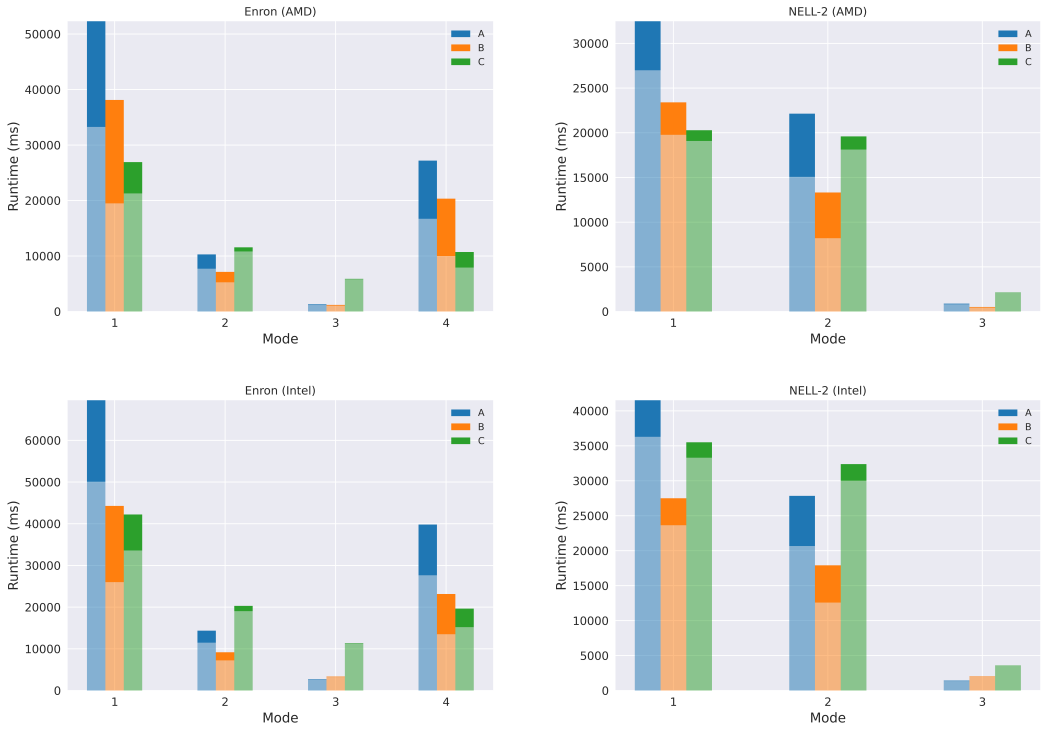

Figure B.3: Run time comparison for tensors Enron and NELL-2 using the AMD processor using the AMD processor. The light shaded bar represents the run time for TVM alone, while the dark shaded bar denotes the combined run time of TVM and Bit-IF construction. Legends: A=TVM with COO, B=non-blocking TVM with lexicographic traversal, C=the fastest blocked TVM computation selected from all configurations.

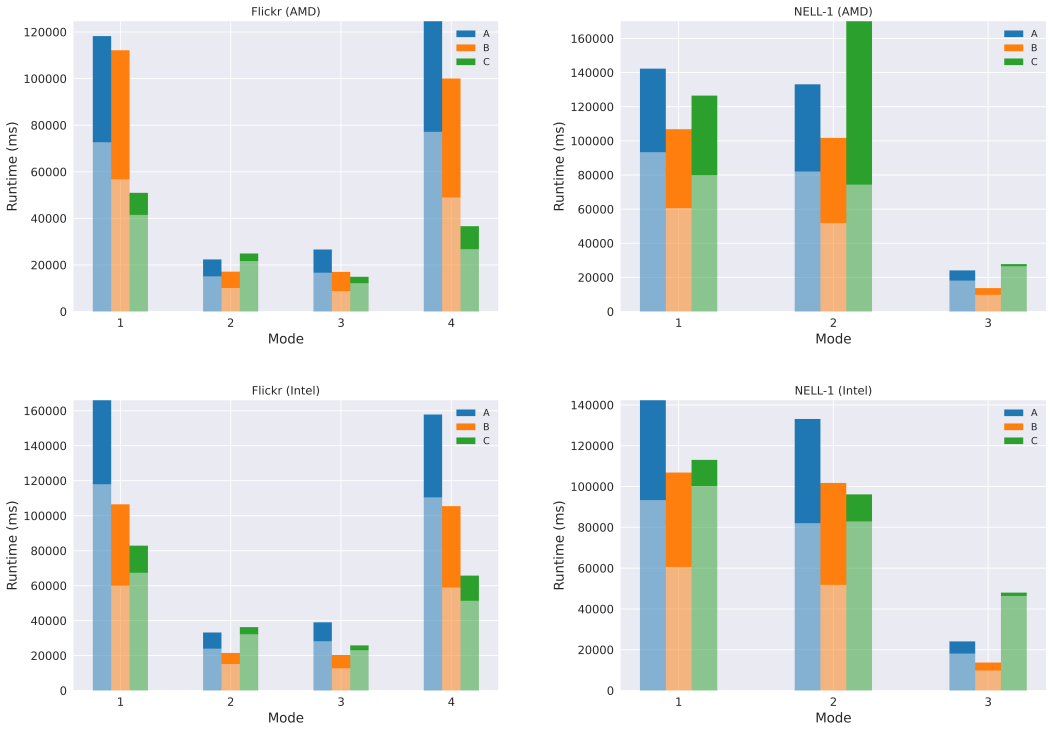

Figure B.4: Run time comparison for tensors Flickr and NELL-1 using the AMD processor (top images) and Intel processors (bottom image). The light shaded bar represents the run time for TVM alone, while the dark shaded bar denotes the combined run time of TVM and Bit-IF construction. Legends: A=TVM with COO, B=non-blocking TVM with lexicographic traversal, C=the fastest blocked TVM computation selected from all configurations.