Dynasor: A Dynamic Memory Layout for Accelerating Sparse MTTKRP for Tensor Decomposition on Multi-core CPU

Sasindu Wijeratne*, Rajgopal Kannan†, Viktor Prasanna*
* University of Southern California, Los Angeles, USA
† DEVCOM Army Research Lab, Los Angeles, USA
Email: {kangaram, prasanna}@usc.edu, rajgopal.kannan.civ@army.mil

***Abstract*—Sparse Matricized Tensor Times Khatri-Rao Product (spMTTKRP) is the most time-consuming compute kernel in sparse tensor decomposition. In this paper, we introduce a novel algorithm to minimize the execution time of spMTTKRP across all modes of an input tensor on multi-core CPU platform. The proposed algorithm leverages the FLYCOO tensor format to exploit data locality in external memory accesses. It effectively utilizes computational resources by enabling lock-free concurrent processing of independent partitions of the input tensor. The proposed partitioning ensures load balancing among CPU threads. Our dynamic tensor remapping technique leads to reduced communication overhead along all the modes. On widely used real-world tensors, our work achieves 2.12× - 9.01× speedup in total execution time across all modes compared with the state-of-the-art CPU implementations.**

***Index Terms*—Tensor Decomposition, spMTTKRP, CPU**

I. Introduction

Tensor Decomposition (TD) enables the transformation of high-dimensional tensors into a lower-dimensional latent space, facilitating the identification of important features in the data distribution. TD finds applications in various fields such as machine learning [1], [2], [3], signal processing [4], and network analysis [5]. Canonical Polyadic Decomposition (CPD) via alternating least squares (CP-ALS) is a widely used TD algorithm where Matricized Tensor Times Khatri-Rao Product (MTTKRP) is the most time-consuming computation.

Numerous tensor formats have been proposed in the literature to tackle the challenges posed by sparse tensors encountered in real-world scenarios [6], [7], [8], [9], [10], [11]. These formats employ multiple tensor copies (i.e., mode-specific tensor formats) or additional memory to store intermediate results of computations to support irregular data access patterns in each tensor mode. However, these approaches have the drawback of increasing the overall memory requirements of the algorithm. Notably, mode-specific tensor formats necessitate several replicas of the original tensor, each arranged based on different permutations of nonzero tensor elements. This replication grows linearly with the number of modes, rendering it impractical as the number of modes increases. Additionally, relying on memory to store intermediate values poses scalability challenges, limiting its usefulness to small datasets. Furthermore, as the size of the tensor grows, there is a potential for memory explosion, further exacerbating the scalability issue.

One desirable solution to reduce memory traffic is to reduce the number of accesses to the data by increasing their reusability. Tensor formats such as HiCOO [8] and ALTO [7], which use variations of Morton ordering [8] to bring the tensor elements with neighboring coordinates closer, exhibit an increase of data reusability. However, these tensor formats still generate a significant number of intermediate values that usually lead to increased memory access time and storage requirements.

Wijeratne et al. [6] introduced FLYCOO, a novel tensor format aimed at accelerating spMTTKRP on Field Programmable Gate Arrays (FPGAs). FLYCOO enhances data locality across all tensor modes while accessing the input tensor and the factor matrices in the FPGA external memory.

We employ a dynamic memory layout to achieve load balancing and communication efficiency, enabling a straightforward static schedule for computing spMTTKRP in each mode. This dynamic approach significantly reduces the generation of intermediate values during computation, which would otherwise need to be communicated to the CPU external memory. To facilitate the memory layout, we propose a parallel algorithm, Dynasor, which performs elementwise computations and dynamic tensor remapping.

The key contributions of this work are:

- We adapt the FLYCOO data format to support spMTTKRP computation on multi-core CPUs by introducing, Dynasor: a parallel algorithm for spMTTKRP with dynamic tensor remapping. We implement Dynasor using C++ and openMP directives.
- **Dynamic memory layout:** By dynamically remapping the input tensor between successive executions of spMTTKRP, we increase the data locality of external memory accesses while avoiding the intermediate value communication to the external memory. Our empirical results demonstrate that our approach can perform under a memory-constrained environment with minimum external memory during runtime compared to state-of-the-art implementations when applied to large tensors.
- **Scheduling and Load balancing:** Our proposed Super-shard-based load balancing technique enables lock-free spMTTKRP computation. Compared to the single-thread implementation, the paper introduces a static scheduling

scheme that achieves a speedup of 8.5× to 21× on 56 CPU threads.

- On widely used real-world tensors, our work achieves 2.12× - 9.01× speedup in total execution time across all modes compared to the state-of-the-art CPU implementations.

II. Background and Related Work

A. Notations

Table I summarizes the notations used in this paper.

TABLE I: Notations

Symbol	Details
$\mathcal{X}$	sparse tensor
$\mathcal{X}_{(n)}$	mode-n matricization of $\mathcal{X}$
$\mathbf{M}$	matrix
$\mathbf{v}$	vector
a	scalar
$\circ$	vector outer product
$\otimes$	Kronecker product
$\odot$	Khatri-Rao product

B. Tensor Decomposition

A tensor is a generalization of an array in multiple dimensions. In TD, the number of dimensions of an input tensor is commonly called the number of tensor modes. A N-mode, real-valued tensor is denoted by $\mathcal{X} \in \mathbb{R}^{I_0 \times \cdots \times I_{N-1}}$. Further, $\mathcal{X}_{(n)}$ denotes the mode-n matricization or matrix unfolding [12] of $\mathcal{X}$. $\mathcal{X}_{(n)}$ is defined as the matrix $\mathcal{X}_{(n)} \in \mathbb{R}^{I_n \times (I_0 \cdots I_{n-1} I_{n+1} \cdots I_{N-1})}$ where the parenthetical ordering indicates, the mode-n column vectors are arranged by sweeping all the other mode indices through their ranges.

Canonical Polyadic Decomposition (CPD) decomposes $\mathcal{X}$ into a sum of single-mode tensors (i.e., arrays), which best approximates $\mathcal{X}$. For example, given 3-mode tensor $\mathcal{X} \in \mathbb{R}^{I_0 \times I_1 \times I_2}$, our goal is to approximate the original tensor as

$$\mathcal{X} \approx \sum_{r=0}^{R-1} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \tag{1}$$

where R is a positive integer and $\mathbf{a}_r \in \mathbb{R}^{I_0}$, $\mathbf{b}_r \in \mathbb{R}^{I_1}$, and $\mathbf{c}_r \in \mathbb{R}^{I_2}$. For a thorough review of CPD, refer to [13].

In the rest of Section II, we assume that the number of modes is three for illustration purposes.

Algorithm 1: CP-ALS for 3-mode tensors

```
1 Input: A tensor $\mathcal{X} \in \mathbb{R}^{I_0 \times I_1 \times I_2}$, the rank $R \in \mathbb{Z}^+$
2 Output: CP decomposition $[\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$, $\mathbf{A} \in \mathbb{R}^{I_0 \times R}$,
    $\mathbf{B} \in \mathbb{R}^{I_1 \times R}$, $\mathbf{C} \in \mathbb{R}^{I_2 \times R}$
3 while stopping criterion not met do
4     // Matricization of $\mathcal{X}$ is different for each factor
        matrix computation
5     $\mathbf{A} \leftarrow \mathbf{spMTTKRP}(\mathcal{X}_{(0)}, \mathbf{B}, \mathbf{C})$
6     $\mathbf{B} \leftarrow \mathbf{spMTTKRP}(\mathcal{X}_{(1)}, \mathbf{A}, \mathbf{C})$
7     $\mathbf{C} \leftarrow \mathbf{spMTTKRP}(\mathcal{X}_{(2)}, \mathbf{A}, \mathbf{B})$
8     Normalize $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$
```

The alternating least squares (ALS) method is used to compute CPD. Algorithm 1 shows the ALS method for CPD (i.e., CP-ALS) where Matricized Tensor-Times Khatri-Rao product (MTTKRP) is iteratively performed on all the Matricizations of $\mathcal{X}$, iteratively. In this paper, performing MTTKRP on all the Matricizations of an input tensor is called computing MTTKRP along all the modes. The outputs $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are the factor matrices that approximate $\mathcal{X}$. $\mathbf{a}_r$, $\mathbf{b}_r$, and $\mathbf{c}_r$ in Equation 1 refers to the r^{th} column of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively.

In this paper, we focus on MTTKRP on sparse tensors (spMTTKRP), which means the tensor is sparse, and the factor matrices are dense.

C. Elementwise computation of spMTTKRP

The objective of this paper is to reduce the total execution time of spMTTKRP along all the modes of the tensor. The efficient execution of elementwise computation of spMTTKRP in all the tensor modes is the key to reducing the total execution time.

Figure 1 summarizes the elementwise computation of a nonzero tensor element in mode 0 of a 3-mode tensor. Here, we use the same notations as Algorithm 1.

In Figure 1, the elementwise computation is carried out on a nonzero tensor element, denoted as $\mathcal{X}_{(0)}(i, j, k)$. In sparse tensors, $\mathcal{X}_{(0)}(i, j, k)$ is typically represented in formats such as COOrdinate (COO) or Compressed Sparse Fiber (CSF). These formats store the indices (i, j, and k) or pointers to these indices along with the element value (i.e., $val(\mathcal{X}_{(0)}(i, j, k))$).

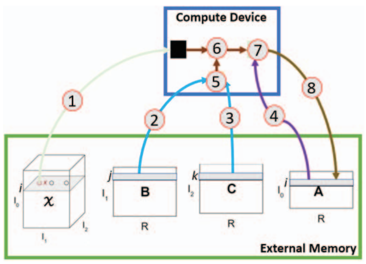


Fig. 1: Elementwise computation of spMTTKRP

To perform the computation, $\mathcal{X}_{(0)}(i, j, k)$ is first loaded onto the Compute device from the external memory (step $\textcircled{1}$). The Compute device retrieves the rows $\mathbf{A}(i,:)$, $\mathbf{B}(j,:)$, and $\mathbf{C}(k,:)$ from the factor matrices using the index values extracted from $\mathcal{X}_{(0)}(i, j, k)$ (step $\textcircled{2}$, step $\textcircled{3}$, and step $\textcircled{4}$). Then the Compute device performs the following computation:

$$\mathbf{A}(i, r) = \mathbf{A}(i, r) + val(\mathcal{X}_{(0)}(i, j, k)) \cdot \mathbf{B}(j, r) \cdot \mathbf{C}(k, r)$$

Here, r refers to the column index of a factor matrix row ($r < R$). The operation involves performing a Hadamard product between row $\mathbf{B}(j,:)$ and row $\mathbf{C}(j,:)$ (step $\textcircled{5}$), and then multiplying each element of the resulting product by $val(\mathcal{X}_{(0)}(i, j, k))$ (step $\textcircled{6}$). After updating $\mathbf{A}(i,:)$ (step $\textcircled{7}$), the updated value is stored back in the external memory (step $\textcircled{8}$).

D. Related Work

Wijeratne et al. [6] develop a customized accelerator on Field Programmable Gate Array (FPGA) for performing spMTTKRP on sparse tensors along with a specific tensor format labeled FLYCOO that supports the FPGA optimizations. In this paper, we focus on optimizing spMTTKRP for multi-core CPUs, which pose substantially different challenges from FPGAs, and adapt the FLYCOO format to support our multi-core CPU optimizations.

Helal et al. [7] propose ALTO, a tensor ordering method based on space-filling curves designed to effectively encode irregularly shaped spaces. ALTO requires minimum external memory to store tensors. However, this tensor format entails storing a large number of intermediate values generated during the computation in external memory during runtime, increasing memory access time. In contrast to ALTO, we use a dynamic memory layout to mitigate the communication time between the CPU and external memory.

J. Li et al. [8] propose HiCOO, a block-based format that utilizes compression techniques to handle sparse tensors by leveraging the Z-Morton curve [14] for efficient storage and retrieval. However, HiCOO encounters workload imbalance issues among blocks due to the irregular spatial distribution of sparse data. Despite both FLYCOO and HiCOO [8] employing similar tensor ordering strategies (i.e., Z-Morton ordering) during format generation, they demonstrate notable distinctions in terms of reduced intermediate value communication, tensor partitioning scheme, and distribution of nonzero elements. An in-depth comparison can be found in [6].

Kurt et al. [15] propose the STeF format to explore the impact of saving partial spMTTKRP results and reusing them during the spMTTKRP computation along all the tensor modes. They present a load-balancing approach at a fine-grained level to enable higher levels of parallelization. In contrast, our work employs a dynamic tensor remapping technique to optimize data locality during elementwise computation. Our proposed parallel algorithm also eliminates the need for additional storage to store partial spMTTKRP results, as required in the algorithm suggested by Kurt et al. [15].

III. FLYCOO Tensor Format

The FLYCOO tensor format is introduced in [6] to perform spMTTKRP for tensor decomposition on FPGAs. Our work adapts the FLYCOO format to accelerate spMTTKRP on multi-core CPU. Section III-A provides a brief overview of the FLYCOO format. More details on FLYCOO format can be found in [6].

In tensor decomposition, spMTTKRP is computed along each mode sequentially as described in Section II-B. When computing spMTTKRP for mode n of the input tensor, mode n is referred to as the output mode and its corresponding factor matrix as the output factor matrix. Meanwhile, the rest of the modes are called input modes, and their factor matrices become input factor matrices.

The FLYCOO format assigns each nonzero tensor element to a tensor partition for each mode and embeds partition IDs to each tensor element. The tensor is divided into multiple partitions, called *super-shards* with an equal number of output mode indices.

A. FLYCOO Tensor Format

For each output mode n ($0 \leq n < N$), consider m_n rows of the output factor matrix. Here, we are considering an input tensor with N-modes. Let I_n denote the indices in mode n. FLYCOO reorganizes the data in the following manner: First, the indices of mode n are partitioned into $k_n = \frac{|I_n|}{m_n}$ equal-sized sets $I_{n,0}, I_{n,1}, \ldots, I_{n,k_{n-1}}$, where$|I_n|$ is the number of indices in mode n. Next, the nonzero tensor elements incident on $I_{n,j}$ are collected into a *super-shard* $SS_{n,j}$. Finally, to support dynamic tensor remapping, each super-shard is further divided into equal-sized sets of size g called shards, where g is a tensor partitioning parameter that is tuned depending on the cache size of the CPU platform (see Section III-C). The q^{th} such shard is denoted as $shard_{n,j,q}$ and the total number of shards for mode n is equal to $\sum_{j=0}^{k_n-1} \lceil |SS_{n,j}|/g \rceil$.

FLYCOO maps each nonzero tensor element to a shard in each mode. A tensor of size $|T|$ with N modes in the FLYCOO format is a sequence $x_0, \ldots, x_{|T|-1}$, where each element x_i is a tuple $\langle s_i, p_i, val_i \rangle$, $s_i = (b_0, \ldots, b_{N-1})$ is a shard ID vector where each shard ID corresponds to a mode of the tensor. Here, $b_n = (j,q)$ if and only if $x_i \in shard_{n,j,q}$. This is used to locate the shards where each nonzero tensor element belongs in each mode. $p_i = (c_0, \ldots, c_{N-1})$ is the original indices of the nonzero tensor element in each dimension. val_i is the value of the nonzero tensor elements of the tensor at p_i. Following the notation used in Section III-B, a single nonzero element in the FLYCOO format requires approximately $N \times \log_2\left(\frac{|T|}{g}\right) + \sum_{h=0}^{N-1} \log_2 |I_h| + \beta_{\text{float}}$ bits, where β_{float} is the number of bits needed to store the floating-point value of the nonzero tensor element. Here, $|s_i| \approx N \times \log_2\left(\frac{|T|}{g}\right)$, $|p_i| = \sum_{h=0}^{N-1} \log_2 |I_h|$, and $|val_i| = \beta_{\text{float}}$.

B. Dynamic Tensor Remapping

During preprocessing, for each mode, nonzero elements of the tensor are assigned to a shard following Section III-A. When performing spMTTKRP mode by mode, the tensor elements are dynamically remapped based on the shard IDs associated with the mode to be executed next. Initially, the tensor is ordered according to the shard IDs of mode 0. During the spMTTKRP computation for mode 0, the tensor is reordered based on the shard IDs of mode 1. Therefore, by the time the computation for mode 1 begins, the tensor is already ordered according to mode 1. This reordering holds true for any mode during the computation process. As proven in [6], the dynamic tensor remapping requires $2 \times |T|$ external memory. This eliminates the need to create additional tensor copies equal to the number of tensor modes, which was previously necessary to facilitate mode-specific optimizations.

C. FLYCOO for multi-core CPU

We adapt the FLYCOO tensor format without modifying the format. We introduce several key contributions. (1) We propose Dynasor, a novel thread-level parallel algorithm that supports multi-core CPU-based spMTTKRP. This algorithm leverages super-shard-wise partition distribution among threads, ensuring load balancing across the workload. (2) We demonstrate that dynamic tensor remapping can be efficiently performed on a CPU-based hierarchical cache memory system, eliminating the need for a specialized memory system as proposed in [6].

(3) We enable elementwise computation with dynamic tensor remapping in a single thread, with maximum parallelization possible on a CPU. (4) We show that FLYCOO can be adapted to general CPU platforms without specific hardware, like custom memory controllers.

In our work, we optimize the tensor partitioning parameters of the FLYCOO format for a given CPU platform. Consider a multi-core CPU with ν threads and a total cache size of Γ: Our goal is to select the tensor partitioning parameters of FLYCOO to (1) utilize all the threads optimally while executing Dynasor and (2) optimally share the cache among inputs and outputs of Dynasor. It can be achieved by satisfying the constraints shown in Equations 2 and 3. Equations 2 and 3 follow the same notations as Section III-A.

$$\forall n;\ \frac{|I_n|}{m_n} = q \times \nu; q \in \mathbb{Z}^+ \tag{2}$$

$$\forall n;\ \Gamma = \theta \times ((\alpha \times m_n \times R + \beta \times g) \times \nu + \sigma \times \sum_{j=0}^{k_n-1} \left\lceil \frac{|SS_{n,j}|}{g} \right\rceil \quad 0 < \theta < 1 \tag{3}$$

The objective is to determine the optimal values for the variables g and m_n for each mode n to efficiently utilize CPU threads and the CPU cache. All of these variables have inter-dependencies. For example, if m_n of mode n is too small, we are not able to choose larger g values for very sparse tensors as the number of nonzero tensor elements of super-shards of mode n can be significantly small.

Equation 2 ensures that the number of super-shards is sufficiently large enough to be distributed among the available CPU threads for all the modes ($\forall n$). Equation 3 describes sharing the cache among different inputs to the proposed algorithm, Dynasor. $(\alpha \times m_n + \beta \times g) \times \nu$ guarantees the intervals that correspond to the super-shards currently being executed on the CPU threads at a given time and the input tensor elements correspond to the super-shards executing on the CPU threads fit inside the total CPU cache. Meanwhile, $\sigma \times \sum_{j=0}^{k_n-1} \left\lceil \frac{|SS_{n,j}|}{g} \right\rceil$ make sure the memory address pointers in dynamic tensor remapping fit inside the total CPU cache. Here, α, β, and σ represent the size of a factor matrix row, nonzero tensor element, and address pointer for dynamic tensor remapping, respectively. We introduce θ (< 1) to share the cache with input factor matrices. In our experiments, we set $\theta = 0.5$. Finally, we select a set of tensor partitioning parameters for a given input tensor that satisfies these requirements.

IV. Parallel Algorithm

In Algorithm 2, tensor ordered according to mode 0 shards ($\mathcal{H}_0$), factor matrices ($\mathbf{Y} = \{Y_0, Y_1, ..., Y_{N-1}\}$), and super-shard to CPU thread map ($\mathcal{SS}\_List$) are used as the inputs. The preprocessing involves converting the input tensor to FLYCOO format, during which the super-shard-to-shard mapping for each mode is generated as metadata. This mapping is essentially a dictionary that indicates which shard belongs to which super-shard. $\mathcal{SS}\_List$ is used as a static scheduling policy of the super-shards among the available threads. More comprehensive details regarding the scheduling process can be found in Section IV-A.

Algorithm 2: Dynasor: Algorithm for spMTTKRP with Dynamic Tensor Remapping on multi-core CPU

```
1  Input: Input tensor ordered according to mode 0
     shards, $\mathcal{H}_0 = \{S_{0,j} : \forall j\}$
2  Super-shard to CPU thread map, $\{\mathcal{SS}\_List_n : \forall n\}$
3  Randomly initialized factor matrices
     $\mathbf{Y} = \{Y_0, Y_1, ..., Y_{N-1}\}$
4  Output: Updated factor matrices-set
     $\hat{\mathbf{Y}} = \{\hat{Y}_0, \hat{Y}_1, ..., \hat{Y}_{N-1}\}$
5  for each mode $n = 0, \ldots, N-1$ do
6      Initialize $\hat{Y}_n$ as a zero matrix
7      for each $S\_Map_{n,j}$ in $\mathcal{SS}\_List_n$ parallel do
8          if $thread_j$ is idle then
9              // $len(\mathcal{SS}\_List_n) \leq$ number of threads
10             for each $S\_pointer \in S\_Map_{n,j}$ do
11                 $S \leftarrow$ Load($S\_pointer$)
12                 for each element, $x_i = \langle s_i, p_i, val_i \rangle \in S$
                    do
                       // -MTTKRP Computation-
13
14                     $value \leftarrow val_i$
15                     $p_i = (c_0, \ldots, c_{N-1})$
16                     $shard\_ids,\ s_i = (b_0, \ldots, b_{N-1})$
17                     $z \leftarrow b_y$
18                     // $\ell$ is a vector of size $R$
19                     $\ell \leftarrow \{1\}$
20                     for input mode
                        $w \in \{0, \ldots, N-1\} \setminus \{n\}$ do
21                         $vec \leftarrow$ Load(row $c_w$ from $w$th
                             factor matrix)
22                         for each rank $r$ in $R$ parallel do
23                             $\ell(r) \leftarrow \ell(r) \times vec(r)$
24                     for each rank $r$ in $R$ parallel do
25                         $\hat{Y}_n(c_n, r) \leftarrow$
                             $\hat{Y}_n(c_n, r) + value \times \ell(r)$
                       // -Dynamic Tensor
                          Remapping-
26
27                     Store($x_i$ at shard$_{b_{(n+1) mod N}}$)
```

In Algorithm 2, a CPU with ν threads can simultaneously process ν super-shards. All threads are synchronized after each mode. Once a super-shard is allocated to a CPU thread, each thread undertakes two distinct operations on every nonzero tensor element within the super-shard. These operations consist of the following: (1) elementwise spMTTKRP computation (see Section II-C), and (2) dynamic tensor remapping.

Using $\mathcal{SS}\_List$, each thread sequentially loads the shards of the assigned super-shards into the CPU cache (Algorithm 2, line 11). Firstly, the index values of all the modes are extracted from the tensor element (Algorithm 2, line 15). Next, the corresponding rows of the input factor matrices are loaded from the external memory (Algorithm 2, lines 20-21). Subsequently, the current CPU thread performs elementwise operations between the tensor element and each element of the rows of the input factor matrices (Algorithm 2, lines 22-25). The elementwise computation can be summarized as Equation 4. Equation 4 uses the same notation as the Algorithm 2. Here, elementwise computation is performed for the r^{th} column element of c_n^{th} row of Y_n factor matrix.

$$\hat{Y}_{n,c_n,r} = \hat{Y}_{n,c_n,r} + value \times \prod_{w=0; w\neq n}^{N-1} \hat{Y}_{w,c_w,r}; 0 \leq r < R \quad (4)$$

After computing spMTTKRP for mode n, the CPU executes spMTTKRP for the subsequent mode, which is given by $(n+1) \mod N$. To support the proposed parallel algorithm, nonzero tensor elements in the input tensor must be ordered according to the output mode shard ids. Therefore, the tensor is remapped based on the shard IDs of mode $(n+1) \mod N$ to compute the spMTTKRP for the upcoming mode. As a result, while the current mode is being executed, the tensor is remapped in parallel according to the shard IDs of the upcoming mode, facilitating the sequential execution of all the modes (Algorithm 2, line 27). The algorithm maintains a record of the locations that need to be filled for each shard of the upcoming mode, determining the memory location to be filled in each shard and allowing for dynamic tensor remapping.

By following this process, the algorithm leverages the parallelism offered by the multiple threads to process multiple super-shards concurrently while ensuring synchronization and orderly execution of the required operations.

Initially, the input tensor is arranged in the CPU external memory based on the shard IDs of mode 0, denoted as $\mathcal{H}_0 = \{SS_{0,j} : \forall j\}$. Meta-data is used as inputs to establish the mapping between threads and shards, enabling efficient computation scheduling across multiple threads. This mapping is represented as $\mathcal{SS\_List}_{n,j} : \forall n, j$. It is worth noting that both $\mathcal{SS\_List}_n$ and $S\_Map_{n,j}$ (Algorithm 2, line 2 and line 10) are arrays are pointers that reference to tensor partitions.

Algorithm 2 is designed for a multi-thread CPU, where each thread can independently perform the elementwise computations on each assigned super-shard without interfering with the other threads (i.e., lock-free computation). Lock-free computation is enabled by collecting all the nonzero tensor elements of an output mode index to the same super-shard. It ensures that all the elementwise computations used to update a row of an output factor matrix are executed on the same CPU thread.

A. *Super-shard Scheduling Scheme*

We propose static super-shard scheduling to ensure balanced workload distribution among the CPU threads. During the preprocessing time, we map each super-shard and its shards to a CPU thread and generate $\mathcal{SS\_List}$. $\mathcal{SS\_List}$ is a 2-dimensional list that indicates the super-shards assigned to each CPU thread in each mode. We employ Algorithm 3, a greedy algorithm to distribute the super-shards among the CPU threads, which achieves balanced workload distribution for each mode. By distributing the super-shards appropriately, the proposed method aims to optimize the utilization of CPU resources and achieve efficient parallel execution of the spMTTKRP computation for each mode.

Algorithm 3: Super-shard scheduling among threads

```
1 Input: A sorted list of super-shards, $SS$ depending on
    the number of shards in the super-shard. Here, $SS_{n,j}$
    indicates the $j^{th}$ super-shard in mode $n$. $|SS_{n,j}|$
    indicates the number of shards belongs to the
    super-shard $SS_{n,j}$
2 $\mathcal{SS\_List}$ with $N \times \nu$ empty bins. Here, $\mathcal{SS\_List}_{n,j}$
    indicates the $j^{th}$ bin of mode $n$. $j^{th}$ bin correspond
    to the $j^{th}$ CPU thread
3 Output: $\mathcal{SS\_List}$, where each super-shards, $SS$
    mapped to a bin inside $\mathcal{SS\_List}$
4 for each mode $n = 0, \ldots, N-1$ do
5     for each super-shard id $j = 0, \ldots, k_{n-1}$ do
6         // identify the least filled bin in mode $n$ of
            $\mathcal{SS\_List}$
7         $ind$ = Index($min(|\mathcal{SS\_List}_{n,i}|)$); $\forall i$
8         $\mathcal{SS\_List}_{n,ind}.append(\mathcal{SS}_{n,j})$
9 return $\mathcal{SS\_List}$
```

In this context, the total number of computations associated with a super-shard is directly proportional to the number of nonzero tensor elements it contains. Since each super-shard is partitioned into shards with an equal number of tensor elements, the total number of computations is also proportional to the number of shards within a super-shard. It is worth noting that the number of shards in a super-shard can vary based on the sparsity of the tensor.

Suppose a tensor of size T is partitioned into super-shards $\{SS_{n,j} : \forall j\}$ for a mode n. Let $|SS_{n,j}|$ be the number of shards in the super-shard $SS_{n,j}$. We reorder the indices of super-shards such that $|SS_{n,j}| \geq |SS_{n,j'}|$ if $j < j'$ so that they are sorted in descending order of the number of shards. We use the sorted list SS as the input to the Algorithm 3. Each super-shard ($SS_{n,j}$) is iteratively assigned to the CPU thread currently the least heavily loaded (i.e., with the least number of shards assigned). We perform the above operation for all the tensor modes as indicated in Algorithm 3. For a given output mode, let $|\overline{SS}|_{\max}$ be the largest number of shards assigned to a single thread, and $|\overline{SS}|^*_{\max}$ be the value of $|\overline{SS}|_{\max}$ in an optimal shard distribution among the threads. Then our proposed greedy approach above guarantees that $|\overline{SS}|_{\max} \leq 4/3 \cdot |\overline{SS}|^*_{\max}$ [16].

B. *Memory Requirements*

In this Section, we use the same notations as Section III. Dynasor requires $2 \times |T|$ to store the tensor, including the additional memory required to perform dynamic tensor remapping.

Dynasor also keeps the total elements of factor matrices equal to $|\mathbf{I}| \times R$, where $|\mathbf{I}| = \sum_{n=0}^{N-1} |I_n|$. As the number of super-shards is constructed to be the same as the number of intervals (see Section III-A), the total number of super-shards equal to $\frac{|I_n|}{m_n}$ for mode n. Since $\mathcal{SS}\_List_n$ used for scheduling (see Section IV-A) requires keeping pointers for each super-shard, it requires $\sum_{n=0}^{N-1} \frac{|I_n|}{m_n}$ pointers. Dynamic tensor remapping required pointers for each shard (Algorithm 2, line 27) resulting in a total of $\sum_{j=0}^{k_n-1} \lceil |SS_{n,j}|/g \rceil$ shard pointers.

V. Experimental Results

A. Experimental Setup

1) Platforms: We conduct detailed experiments on Intel Xeon Gold 5120 CPU with Skylake microarchitecture. The platform consists of two sockets, each CPU consisting of 14 physical cores (28 threads) running at a fixed frequency of 2.2 GHz, sharing 128 GB of CPU external memory. Utilizing this platform, we systematically vary hardware parameters such as the total CPU external memory and the number of CPU threads. To quantify the improvements, we measure CPU thread utilization and CPU external memory utilization.

We also demonstrate superior performance in total execution time on AMD platforms. We use AMD Ryzen Threadripper 3990X CPU with Zen 2 microarchitecture. The platform consists of two sockets, each CPU consisting of 32 physical cores (64 threads) running at a fixed frequency of 2.2 GHz, sharing 128 GB of CPU external memory.

Both systems run Ubuntu 18.04.5 LTS Linux distribution as the Operating System.

2) Implementation: The code is built using g++ 7.5.0 C/C++ compiler (version 19.1.3) and OpenMP application programming interface. The experiments use all hardware threads on the target platforms unless otherwise stated.

We use the Linux perf [17] and Intel Advisor [18] for performance counter measurements, thread pinning, and roofline analysis on the Intel platform.

3) Datasets: Similar to state-of-the-art [6], [7], [8], we use tensors from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) dataset [19]. Table II shows a summary of the characteristics of the tensors.

TABLE II: Characteristics of the sparse tensors

Tensor	Shape	#NNZs	Density
Nell-1	$2.9M \times 2.1M \times 25.5M$	$143.6M$	9.1×10^{-13}
Nell-2	$12.1K \times 9.2K \times 28.8K$	$76.9M$	2.4×10^{-05}
Flickr	$319.6K \times 28.2M \times 1.6M$	$112.9M$	1.1×10^{-14}
Delicious	$532.9K \times 17.3M \times 2.5M \times 1.4K$	$140.1M$	4.3×10^{-15}
Vast	$165.4K \times 11.4K \times 2 \times 100 \times 89$	$26M$	7.8×10^{-07}

4) Baselines: We evaluate our approach against state-of-the-art CPU-based work ALTO [7], HiCOO [8], and STeF [15]. ALTO, HiCOO, and Dynasor (i.e., our work) are executed on an identical CPU environment. Additionally, we compare against the reported results of STeF in [15].

To obtain the best results with HiCOO, we follow the recommended configurations of the source code [20]. However, the HiCOO code [20] only supports tensors with up to 4 modes. Therefore, we use a benchmarking tool [21] provided by the authors of HiCOO [8] to estimate the execution time for tensors with more than 4 modes. In our experiments, we use open-source ALTO repository [22] compiled with GCC and BLAS library [23].

5) Tensor Partitioning Parameters of FLYCOO: When $|I_n|$ is less than ν, we set m_n equal to 1. Conversely, when $|I_n|$ is much larger than ν, we determine the value of q from Equation 2, such that $1000 < m_n < 16000$ holds true for both CPU platforms. Additionally, we ensure that the value of g falls within the range of 1024 to 32768 while satisfying Equation 3 for all datasets on both CPU platforms.

B. Implementing Dynamic Remapping

In Algorithm 2, we can perform the elementwise computation and dynamic tensor remapping integrated into the same thread or independently in different threads, overlapping the execution of elementwise computation and dynamic tensor remapping. Figure 2 illustrates the overall improvement in execution time achieved by performing both operations on the same thread compared to running them on separate threads. Throughout the paper, we use integrated remapping and elementwise computation as it shows better speedups.

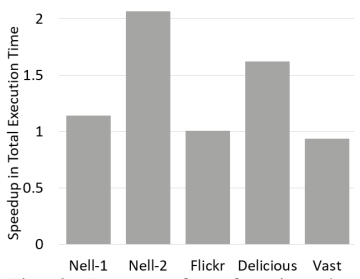


Fig. 2: Impact of performing dynamic tensor remapping and elementwise spMTTKRP computation in the same CPU thread vs. different CPU threads

C. Overall Performance

Figure 3 compares the total execution time of ALTO, HiCOO, and Dynasor for all the datasets on the Intel CPU. The factor matrix rank (R) in the experiments varies from 16 to 256.

For Flickr and Nell-1, ALTO and HiCOO ran out-of-memory while computing factor matrices for larger ranks (e.g., $R = 256$) since the intermediate values generated during the computation exceed the available CPU external memory (128 GB). In contrast, Dynasor is able to compute the factor matrices for all the ranks. Super-shards-wise computation in Dynasor avoids such memory explosion as it allows intermediate values to be fit within the CPU cache during runtime.

Dynasor exhibits superior overall performance in total computation time, except for a few specific cases. For example, Dynasor partitions the indices in each mode equally among the super-shards, which are then distributed across the CPU cores for computation. In mode 3 of the Vast dataset, there are only 2 indices that limit the distribution of the workload across all the threads. As a result, ALTO outperforms our approach in mode 3 of the Vast dataset. In Nell-2, each mode contains only thousands of indices (i.e., 12092, 9184, and 28818) which are smaller than other large datasets with millions of indices per

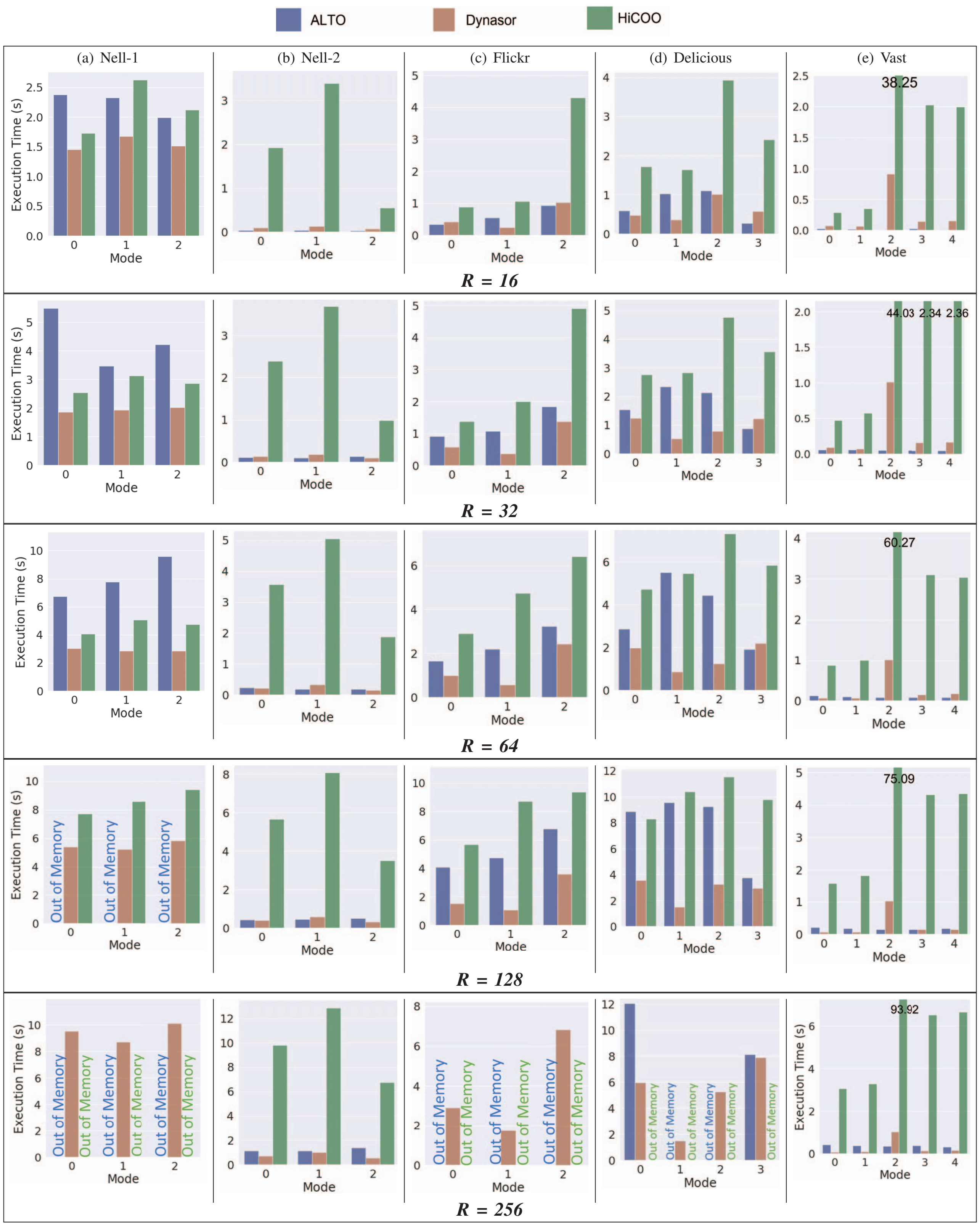


Fig. 3: Total execution time on Intel platform

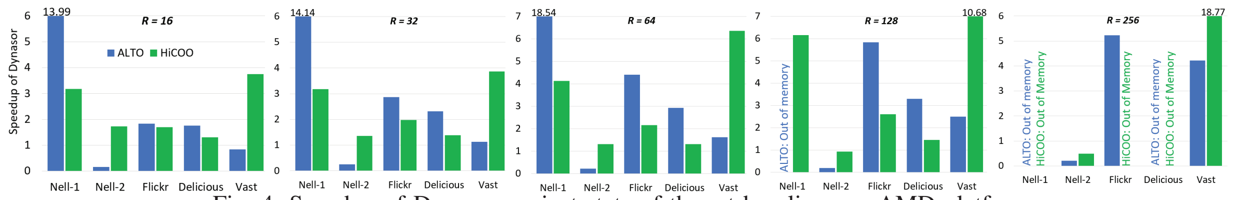


Fig. 4: Speedup of Dynasor against state-of-the-art baselines on AMD platform

TABLE III: Speedup of Dynasor over state-of-the-art

	$R = 16$	$R = 32$	$R = 64$	$R = 128$	$R = 256$	Overall Speedup
Speedup over ALTO [7] on Intel platform	0.88	1.47	1.92	2.25^{+}	2.36^{+}	**1.70^{+}**
Speedup over HiCOO [8] on Intel platform	9.02	9.19	11.51	14.34	37.55^{*}	**13.67^{*}**
Speedup over ALTO [7] on AMD platform	3.75	4.11	5.53	2.95^{+}	3.21^{+}	**3.22^{+}**
Speedup over HiCOO [8] on AMD platform	2.33	2.35	3.03	4.37	9.63^{*}	**4.34^{*}**
Speedup over STeF [15]	n/a	1.71	2.52	n/a	n/a	**2.12**
Overall Speedup	**4.00**	**3.77**	**4.91**	**5.98^{+}**	**13.19^{+*}**	**6.37^{+*}**

+: ALTO runs out of memory for some input tensors
$*$: HiCOO runs out of memory for some input tensors
n/a: Not reported in [15]

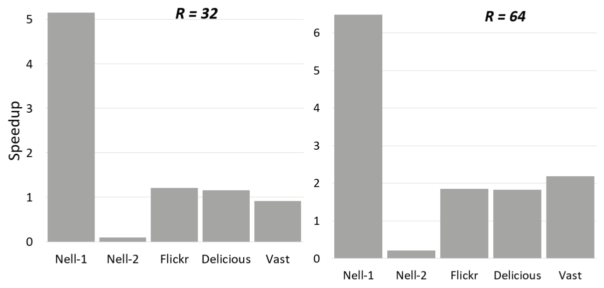


Fig. 5: Speedup compared with STeF

mode. Consequently, all the factor matrices of Nell-2 fit inside the CPU cache. Therefore, combining intermediate values in Adaptive Linearized Order (in ALTO) becomes an inexpensive operation compared to dynamic remapping use in Dynasor. Therefore, ALTO outperforms Dynasor for Nell-2.

To demonstrate platform independence, we conducted identical experiments on an AMD Ryzen Threadripper 3990X CPU platform. As shown in Figure 4, the total execution time exhibits a similar trend as the Intel platform. The overall speedup of Dynasor is 3.22× and 4.32× compared with the baselines (HiCOO and ALTO) executed on the same AMD platform.

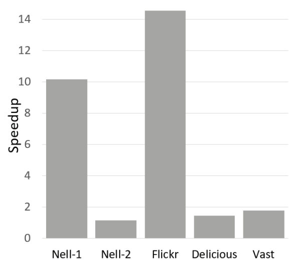


Fig. 6: Impact of proposed scheduling strategy

Based on the results reported in Kurt et al. [15], we compare our results and STeF. We employ the identical AMD platform as in [15]. Figure 5 illustrates the results, showcasing that our approach surpasses STeF in terms of overall performance. STeF incorporates a selective intermediate value storage technique during computation, while our approach leverages a dynamic tensor remapping technique, resulting 2.12× average speedup in total execution time. In datasets like Nell-2, since there are only a few thousand indices along all the modes, the selective intermediate value storage technique used in STeF outperforms the dynamic tensor remapping technique used in Dynasor.

Compared with the baselines, our approach achieves an average speedup of 6.37×. Table III summarizes the overall speedup achieved by our approach compared with the baselines for ranks between 16-256.

D. Impact of Super-shard Scheduling

Section IV-A presents the scheduling strategy we used in this work to distribute the super-shards among CPU threads. Here, the objective is to distribute the nonzero tensor elements equally among the threads for execution. Our scheduling strategy guarantees the maximum number of nonzero tensor elements scheduled for a thread does not exceed 4/3 times the optimal load in each mode (see Section IV-A).

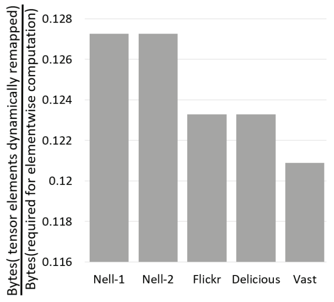


Fig. 8: Total data remapped dynamically vs. total data communicated during the elementwise computations of spMTTKRP

A state-of-the-art approach to scheduling super-shards among threads in each mode is to distribute the super-shards in block-cyclic manner [24].

E. Cost of Tensor Remapping

Figure 6 compares the state-of-the-art and our proposed method in Section IV-A. The experiments are conducted on the Intel Xeon platform. We also use the same FLYCOO tensor partitioning parameters and memory layout for both cases. Our proposed scheduling strategy achieves 1.1× to 14.2× speedups on sparse tensors w.r.t. total execution time.

As shown in Figure 8, the overall volume of dynamically remapped data during the computation (in all tensors)

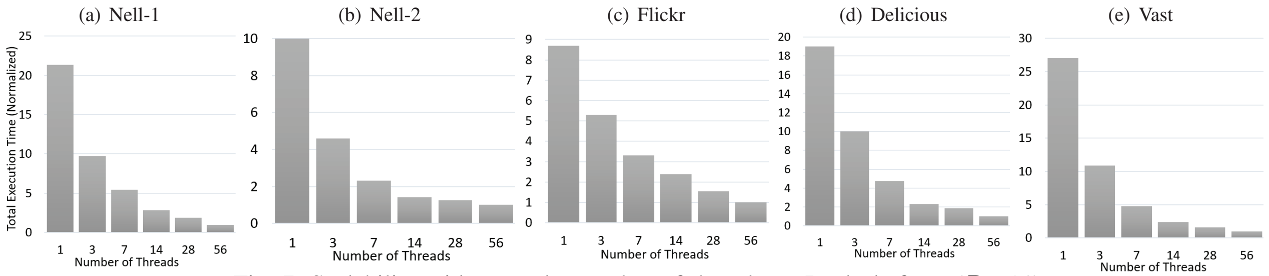


Fig. 7: Scalability with w.r.t. the number of threads on Intel platform ($R = 16$)

is consistently below 15% of the total data communicated during the elementwise computations of spMTTKRP. Hence the dynamic tensor remapping does not significantly contribute to the total memory traffic. We determine the volume of data transmitted in various data types, such as tensor elements and factor matrices, by incorporating data element counters into the source code. These counters allow us to measure and track the amount of data communicated during the execution of the program.

F. Scalability

Figure 7 shows the execution time of the tensors as the number of threads is varied for $R = 16$. Note that execution time is normalized, and the x-axis is in log scale. Utilizing the 56 CPU threads leads to an 8.5× - 21× reduction in the overall execution time compared with the single thread implementation. As shown in the roofline model (see Section V-G), the spMTTKRP operation is a memory-bound computation. Hence the speedup saturates as the number of CPU threads is increased. Similar scalability is observed for $R = 32, 64, 128$, and 256.

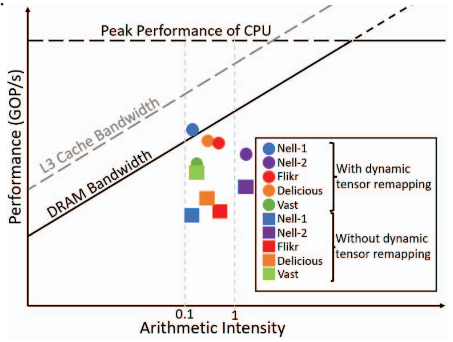


Fig. 9: Roofline analysis on Intel platform

G. Roofline Analysis

Figure 9 illustrates the roofline model [25], generated using Intel Advisor [18] for the datasets on the Intel CPU. The arithmetic intensity of the elementwise spMTTKRP computation (see Section II-C) is used as the x-axis.

We consider two distinct cases to evaluate the impact of dynamic tensor remapping: Case 1 involves executing Dynasor with dynamic tensor remapping across all modes, while Case 2 involves performing elementwise computation without employing dynamic tensor remapping. In Case 2, we organize the tensor based on shard ids in each mode and perform elementwise computation across all modes without dynamic remapping. For example, for a 3-mode input tensor, we organize the tensor-based shards of mode 0 and perform elementwise computation for modes 0, 1, and 2. This process is repeated after ordering the tensor for all the modes, resulting in multiple roofline values. The best performance value (y-axis) among those is reported in Figure 9 as "without dynamic tensor remapping".

As depicted in Figure 9, including dynamic tensor remapping significantly enhances the performance for all the tensors, compared to the scenario where only the elementwise computation is performed without dynamic tensor remapping.

H. Impact of Factor Matrix Rank

The total execution time as the rank of the factor matrices (R) is varied, depicted in Figure 10. Loading the input factor matrices dominates the external memory traffic generated in Algorithm 2. Using the same notation as Section III, the total number of the input factor matrix accesses is proportional to $N \times (N-1) \times |T| \times R$. Consequently, the total CPU memory traffic is proportional to R. Since spMTTKRP is a memory-bound computation for sparse tensors (as highlighted in Section V-G), the total execution time is proportional to R.

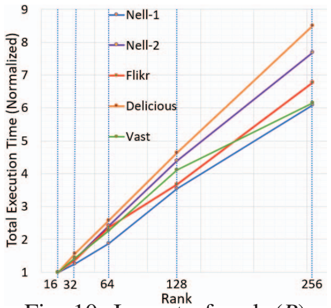


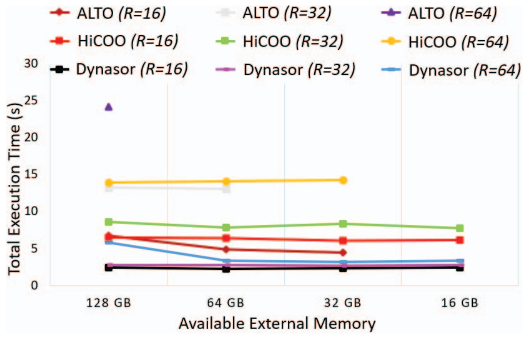
Fig. 10: Impact of rank (R)

I. Impact of External Memory Size

Despite the possibility of compressing the total size of the tensor using various tensor formats, excessive generation of intermediate values during runtime can lead to memory explosion. This can result in an out-of-memory error during execution. To illustrate the resilience of Dynasor to the memory explosion, we use 2 of the largest tensors, Nell-1 and Flickr (similar results have been observed in other datasets).

Figure 11 (a) and Figure 11 (b) show the total execution time of Nell-1 and Flickr when executed on limited CPU external memory. In the experiments, we varied the rank of the factor matrices from 16 to 64. For Nell-1 and Flickr, the total external memory requirement to store the tensor memory layout (i.e., including additional space to store remapped tensor), factor matrices, and other metadata stays between 10 GB - 15.6 GB

(a) Nell-1



(b) Flickr

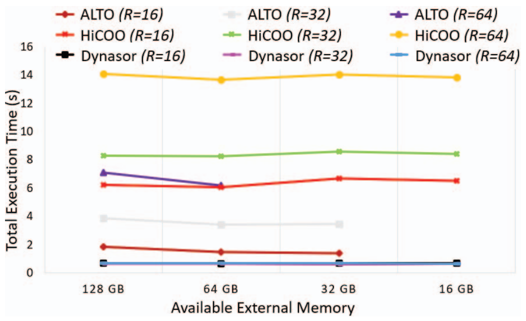


Fig. 11: Impact of external memory size. Missing data points indicates out-of-memory due to memory explosion

for R values of 16, 32, and 64. Consequently, we vary the size of the CPU external memory (of the Intel platform) between 128 GB (the total available memory) and 16 GB using the OS settings for memory management.

Despite using the low-rank factor matrices ($R = 16$), ALTO cannot perform spMTTKRP with 16 GB of external memory. ALTO also incurs memory explosion as the rank increases, making it impractical for higher ranks. The memory explosion occurs due to excessive intermediate values generated during the execution time. In contrast, HiCOO can be executed in limited memory for low ranks (e.g., $R = 16$). However, ALTO and HiCOO fail to complete the execution using limited external memory as the rank increases (e.g., $R = 64$).

We can perform spMTTKRP in all the cases while achieving minimum execution time. Furthermore, the execution time remains almost the same as the external memory size is decreased for both datasets.

J. Preprocessing Time

The preprocessing of an input tensor comprises three stages: (1) super-shard generation for each mode, (2) Z-Morton ordering of the super-shards, and (3) shard generation using the super-shards. We have parallelized the preprocessing using OpenMP and Boost library [26]. Although the focus of our paper is not on the preprocessing time, Figure 12 compares the preprocessing time across various baselines. We use the same Intel Xeon platform (see Section V-A1) for all the implementations. The preprocessing time of Dynasor is considerably shorter than that of HiCOO for datasets that contain a large number of indices along each mode. This distinction arises because the partitioning scheme employed by Dynasor focuses solely on the nonzero elements, whereas the HiCOO partitioning scheme operates across the entire index space, encompassing all potential combinations of index values across all the modes. For tensors with a small number of mode indices but a large number of nonzeros (e.g., Vast), the HiCOO format outperforms our approach in preprocessing time. Furthermore, the ALTO format generation performs faster than Dynasor due to implementation inefficiencies in tensor element ordering based on Z-Morton. We intend to address this issue in future work by further optimizing our preprocessing implementation.

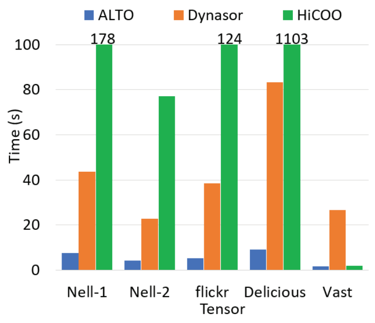


Fig. 12: Preprocessing time on Intel platform

VI. Conclusion and Future Work

This paper presented a novel parallel algorithm for spMTTKRP across all the modes of an input tensor on multi-core CPUs. We reduced the total execution time of spMTTKRP by employing a parallel algorithm that enables concurrent processing of independent partitions of the input tensor. Furthermore, our algorithm reduces the intermediate values being communicated to external memory. The experimental results demonstrate that our work achieves a geometric mean of 6.37× speedup in execution time compared with the state-of-the-art CPU implementations across widely-used real-world sparse tensor datasets.

Our future work focuses on adapting the parallel algorithm and FLYCOO tensor format for emerging heterogeneous systems and massively parallel computing platforms. We plan to utilize massively parallel computing platforms, including GPU, to perform spMTTKRP.

Acknowledgment

This work is supported by the National Science Foundation (NSF) under grants OAC-2209563, CNS-2009057 and in part by DEVCOM Army Research Lab under grant W911NF2220159.
We also acknowledge Jiajia Li (North Carolina State University), Ahmed E. Helal (Intel Labs), and Fabrizio Petrini (Intel Labs) for their support in setting up the baseline experiments. Distribution Statement A: Approved for public release. Distribution is unlimited.

References

[1] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[2] M. Mondelli and A. Montanari, "On the connection between learning two-layer neural networks and tensor decomposition," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 1051–1060.

[3] Z. Cheng, B. Li, Y. Fan, and Y. Bao, "A novel rank selection scheme in tensor ring decomposition based on reinforcement learning for deep neural networks," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 3292–3296.

[4] F. Wen, H. C. So, and H. Wymeersch, "Tensor decomposition-based beamspace esprit algorithm for multidimensional harmonic retrieval," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 4572–4576.

[5] S. Fernandes, H. Fanaee-T, and J. Gama, "Tensor decomposition for analysing time-evolving social networks: An overview," *Artificial Intelligence Review*, pp. 1–26, 2020.

[6] S. Wijeratne, T.-Y. Wang, R. Kannan, and V. Prasanna, "Accelerating sparse mttkrp for tensor decomposition on fpga," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 259–269. [Online]. Available: https://doi.org/10.1145/3543622.3573179

[7] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, "Alto: Adaptive linearized storage of sparse tensors," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 404–416. [Online]. Available: https://doi.org/10.1145/3447818.3461703

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

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

[10] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356216

[11] J. Li, B. Uçar, U. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 227–237. [Online]. Available: https://doi.org/10.1145/3330345.3330366

[12] G. Favier and A. L. de Almeida, "Overview of constrained parafac models," *EURASIP Journal on Advances in Signal Processing*, vol. 2014, no. 1, pp. 1–25, 2014.

[13] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[14] D. S. Wise, "Ahnentafel indexing into morton-ordered arrays, or matrix locality for free," in *Euro-Par 2000 Parallel Processing*, A. Bode, T. Ludwig, W. Karl, and R. Wismüller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 774–783.

[15] S. E. Kurt, S. Raje, A. Sukumaran-Rajam, and P. Sadayappan, "Sparsity-aware tensor decomposition," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 952–962.

[16] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[17] A. C. de Melo and R. Hat, "The new linux ' perf ' tools," 2010.

[18] K. O'Leary, I. Gazizov, A. Shinsel, R. Belenov, Z. Matveev, and D. Petunin, "Intel® advisor roofline analysis," *THE CHANGING HPC LANDSCAPE STILL LOOKS THE SAME*, p. 56, 2017.

[19] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[20] J. Li, B. Uçar, U. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," 2019. [Online]. Available: https://github.com/hpcgarage/ParTI

[21] J. Li, M. Lakshminarasimhan, X. Wu, A. Li, C. Olschanowsky, and K. Barker, "A parallel sparse tensor benchmark suite on CPUs and GPUs," 2020. [Online]. Available: https://gitlab.com/tensorworld/pasta

[22] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, "Alto: Adaptive linearized storage of sparse tensors," 2021. [Online]. Available: https://github.com/IntelLabs/ALTO

[23] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[24] L. Prylli and B. Tourancheau, "Efficient block cyclic data redistribution," in *Euro-Par'96 Parallel Processing: Second International Euro-Par Conference Lyon, France, August 26–29 1996 Proceedings, Volume I 2*. Springer, 1996, pp. 155–164.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[26] B. Schäling, *The boost C++ libraries*. XML press Laguna Hills, 2014, vol. 3.