# Accelerating Sparse MTTKRP for Tensor Decomposition on FPGA

Sasindu Wijeratne
University of Southern California, USA
kangaram@usc.edu

Ta-Yang Wang
University of Southern California, USA
tayangwa@usc.edu

Rajgopal Kannan
DEVCOM US Army Research Lab, USA
rajgopal.kannan.civ@army.mil

Viktor Prasanna
University of Southern California, USA
prasanna@usc.edu

## ABSTRACT

Sparse Matricized Tensor Times Khatri-Rao Product (spMTTKRP) is the most computationally intensive kernel in sparse tensor decomposition. In this paper, we propose a hardware-algorithm co-design on FPGA to minimize the execution time of spMTTKRP along all modes of an input tensor. We introduce FLYCOO, a novel tensor format that eliminates the communication of intermediate values to the FPGA external memory during the computation of spMTTKRP along all the modes. Our remapping of the tensor using FLYCOO also balances the workload among multiple Processing Engines (PEs). We propose a parallel algorithm that can concurrently process multiple partitions of the input tensor independent of each other. The proposed algorithm also orders the tensor dynamically during runtime to increase the data locality of the external memory accesses. We develop a custom FPGA accelerator design with (1) PEs consisting of a collection of pipelines that can concurrently process multiple elements of the input tensor and (2) memory controllers to exploit the spatial and temporal locality of the external memory accesses of the computation. Our work achieves a geometric mean of 8.8× and 3.8× speedup in execution time compared with the state-of-the-art CPU and GPU implementations on widely-used real-world sparse tensor datasets.

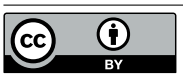
## CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; **Special purpose systems**.

## KEYWORDS

Tensor Decomposition, Sparse MTTKRP, FPGA, Hardware Accelerators

**ACM Reference Format:**
Sasindu Wijeratne, Ta-Yang Wang, Rajgopal Kannan, and Viktor Prasanna. 2023. Accelerating Sparse MTTKRP for Tensor Decomposition on FPGA. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543622.3573179





*FPGA '23, February 12–14, 2023, Monterey, CA, USA*

ACM ISBN 978-1-4503-9417-8/23/02.
https://doi.org/10.1145/3543622.3573179

## 1 INTRODUCTION

Tensor Decomposition (TD) transforms input tensors into a reduced latent space which can then be leveraged to learn salient features of the underlying data distribution. TD has been successfully employed in many fields including machine learning [2, 15, 18], signal processing [22], and network analysis [3]. One popular TD algorithm is Canonical Polyadic Decomposition via alternating least squares (CP-ALS) [6], which can be accelerated by optimizing spMTTKRP, its most expensive computation task. Since real-world tensors are generally sparse, keeping only nonzero values in memory is a natural way to reduce the memory footprint. However, there is a concomitant need to develop optimized sparse tensor formats that support highly irregular data access patterns of spMTTKRP [23].

Several tensor formats have been proposed for real-world sparse tensors [5, 12, 14, 16, 20]. Many of the proposed formats seek to alleviate the problem of irregular data access by either using the number of tensor copies proportional to the number of modes or through additional memory to save intermediate by-products of the computation. However, as the number of modes and the sparsity of the tensor increases, these approaches might introduce significant amount of memory overhead. One desirable solution to reduce the memory traffic to the external memory is to reduce the number of accesses to the external memory by increasing the data reuse. A cache can be used as an intermediate memory to achieve this goal. In addition, reordering the tensor based on space-filling curves has shown promising results. Tensor formats such as HiCOO [12] and ALTO [5] use variations of Z-Morton data ordering [12]. It brings the tensor elements with neighboring coordinates closer and increases the data reuse. However, these tensor formats still communicate a significant amount of intermediate values to the external memory. In this paper, we propose FLYCOO, a tensor format that eliminates the communication of intermediate values to the FPGA external memory during the execution time of spMTTKRP along all the modes of the input tensor. It also increases the locality of data used in spMTTKRP. Our work hides the cost of on-the-fly tensor remapping time by overlapping the spMTTKRP computations with the tensor remapping.

The key contributions of this paper are:

- We introduce FLYCOO, a novel tensor format that eliminates the communication of intermediate values generated during spMTTKRP computation to the FPGA external memory. FLYCOO balances the workload among multiple Processing Engines along all the output modes of the input tensor.
- We develop a novel FPGA accelerator for spMTTKRP, which consists of (1) PEs with multiple pipelines to concurrently

process input tensors in a streaming fashion, (2) cache subsystems to exploit the locality of input factor matrices, (3) Direct Memory Access (DMA) buffers to load the input tensor partitions and store the output factor matrices in a streaming fashion, and (4) custom hardware to support on-the-fly tensor remapping.

- The memory controller design introduced in this paper reduces the total execution time by 4.5× compared with the traditional Direct Memory Access (DMA) buffer-based memory controllers. The memory controller further reduces 67% of the pipeline stalls caused by FPGA external memory accesses.
- On widely used real-world large tensor datasets, our FPGA accelerator achieves a geometric mean of 8.8× and 3.8× improvement in total execution time compared with the state-of-the-art CPU and GPU implementations, respectively.

# 2 BACKGROUND

## 2.1 Notations

An $N$-dimensional, real-valued sparse tensor is denoted by $\mathcal{X} \in \mathbb{R}^{I_0 \times \cdots \times I_{N-1}}$. For a thorough review of tensors and tensor algebra, please refer to [10]. Table 1 summarizes the list of symbols used in this paper.

**Table 1: Notations**

| Symbol | Description |
|---|---|
| $a$ | scalar (lowercase letter) |
| $\mathbf{v}$ | vector (bold lowercase letter) |
| $\mathbf{M}$ | matrix (bold capital letter) |
| $\mathcal{X}$ | sparse tensor (Euler script letter) |
| $N$ | number of modes |
| $\mathcal{X}_{(n)}$ | mode-$n$ matricization of the tensor $\mathcal{X}$ |
| $\circ$ | vector outer product |
| $\otimes$ | Kronecker product |
| $\odot$ | Khatri-Rao product |

## 2.2 Tensor Decomposition

Canonical Polyadic Decomposition (CPD) [10] on sparse tensors decomposes a sparse tensor $\mathcal{X}$ into a sum of dense matrices for each mode which best approximates the original sparse tensor. For example, given a sparse tensor $\mathcal{X} \in \mathbb{R}^{I_0 \times I_1 \times I_2}$, our goal is to express it as $\mathcal{X} \approx \sum_{r=0}^{R-1} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$, where $R \in \mathbb{Z}^+$ refers to the rank of $\mathcal{X}$, which is defined as the smallest sum of rank-one tensors required to generate $\mathcal{X}$, $\mathbf{a}_r \in \mathbb{R}^{I_0}$, $\mathbf{b}_r \in \mathbb{R}^{I_1}$, and $\mathbf{c}_r \in \mathbb{R}^{I_2}$ for $r = 0, \ldots, R-1$. For illustration purposes, we assume the number of modes of tensors to be three in the rest of the section. The components of the above summation can be expressed as input factor matrices, i.e., $\mathbf{A} = [\mathbf{a}_0, \ldots, \mathbf{a}_{R-1}]$ and likewise for $\mathbf{B}$ and $\mathbf{C}$. It is often useful to constrain the components to unit length, factoring the weights into the vector $\lambda = [\lambda_0, \ldots, \lambda_{R-1}] \in \mathbb{R}^R$, which allows to concisely express the model as $\mathcal{X} \approx [[\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]] = \sum_{r=0}^{R-1} \lambda_r \cdot \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$.

Since the problem is non-convex and has no closed-form solution, existing methods for this optimization problem rely on iterative schemes. The alternating least squares algorithm for computing the CP decomposition (CP-ALS) is the most popular method due to its simplicity and efficiency. Algorithm 1 shows a common formulation of CP-ALS for 3-mode tensors. It consists of several iterations of the sparse Matricized Tensor-Times Khatri-Rao Product (spMTTKRP) operation on each mode.

As illustrated in Algorithm 1, executing spMTTKRP in each mode is the most expensive operation of CP-ALS. spMTTKRP involves the mode-$d$ matricization $\mathcal{X}_{(d)}$ and the Khatri-Rao product [5] — given two matrices $\mathbf{B} \in \mathbb{R}^{I_1 \times R}$ and $\mathbf{C} \in \mathbb{R}^{I_2 \times R}$, their Khatri-Rao product $\mathbf{B} \odot \mathbf{C} = [\mathbf{b}_1 \otimes \mathbf{c}_1 \quad \mathbf{b}_2 \otimes \mathbf{c}_2 \cdots \mathbf{b}_R \otimes \mathbf{c}_R]$. spMTTKRP can be expressed as

$$\textbf{spMTTKRP}(\mathcal{X}_{(r)}, \mathbf{B}, \mathbf{C}) = \mathcal{X}_{(r)}(\mathbf{B} \odot \mathbf{C}).$$

**Algorithm 1:** CP-ALS for the 3-mode tensors

```
Input: A tensor X ∈ R^{I0×I1×I2}, the rank R ∈ Z+
Output: CP decomposition [[λ; A, B, C]], λ ∈ R^R, A ∈ R^{I0×R},
  B ∈ R^{I1×R}, C ∈ R^{I2×R}
while stopping criterion not met do
    A ← spMTTKRP(X_(0), B, C)
    B ← spMTTKRP(X_(1), A, C)
    C ← spMTTKRP(X_(2), A, B)
    Normalize A, B, C and store the norms as λ
```

# 3 RELATED WORK

Srivastava et al. [21] propose a custom CGRA fabric to accelerate sparse computations, including spMTTKRP. The authors use a mode-specific tensor format to compute spMTTKRP. In our work, we propose a mode-agnostic tensor format that only requires one additional tensor copy regardless of the number of modes of the tensor.

Nisa et al. [17] optimize MTTKRP on GPUs. They propose a tensor slicing technique for the load balancing between GPU warps. We develop a simple load balancing scheme that fairly shares the total workload between ACCELs (see Section 4.6).

There are several CPU-based MTTKRP acceleration algorithms proposed in the literature. J. Li et al. propose HiCOO [12], a block-based format that compresses the sparse tensor. Helal et al. propose ALTO [5], a space-filling curve-based tensor ordering method that can efficiently encode spaces with irregular shapes. ALTO requires the least amount of external memory to store tensors. Unlike prior formats, we perform on-the-fly memory layout remapping to reduce external memory communication time.

# 4 ACCELERATOR DESIGN

## 4.1 Target Platform

We develop a hardware accelerator for a data center FPGA device directly connected to external DRAM memory.

## 4.2 Hypergraph Representation of Tensors

The hypergraph representation of a tensor has been used to describe the spMTTKRP operation in the literature [8, 13]. To describe our proposed tensor format based on the hypergraph, we will briefly introduce the hypergraph representation of a tensor in this section. For a given tensor $\mathcal{X}$ with $N$ modes, we build a hypergraph $\mathscr{H} = (\mathscr{V}, \mathscr{E})$ with the vertex set $\mathscr{V}$ and the hyperedge set $\mathscr{E}$ as follows: $\mathscr{V} = V_0 \cup V_1 \cup \cdots \cup V_{N-1}$, where $V_n$ is the set of all the tensor indices in mode $n$; $\mathscr{E}$ contains

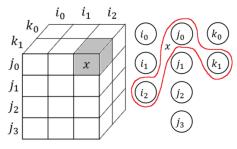



**Figure 1: Hypergraph of a sparse tensor**

hyperedges that represent the nonzero tensor elements in $\mathcal{X}$. For a 3-mode sparse tensor $\mathcal{X} \in \mathbb{R}^{|V_0| \times |V_1| \times |V_2|}$ with $M$ nonzero tensor elements, its hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ consists of $|\mathcal{V}| = |V_0| + |V_1| + |V_2|$ vertices and $|\mathcal{E}| = M$ hyperedges. A hyperedge $\mathcal{X}(i, j, k)$ connects the three vertices $i$, $j$, and $k$, which correspond to the row indices of the factor matrices. Figure 1 shows an example of the hypergraph for a sparse tensor.

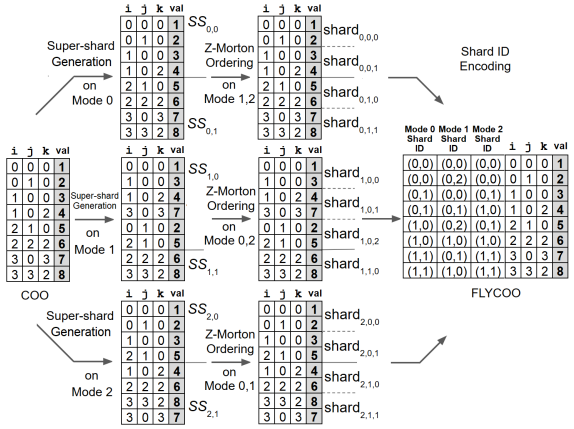



**Figure 2: Example FLYCOO Format Generation**

## 4.3 Tensor Format

During tensor decomposition, spMTTKRP is computed along each mode, one mode after the other (see Algorithm 1). For an $N$-mode tensor, when computing spMTTKRP for mode $n$, we refer to mode $n$ as the output mode and its corresponding factor matrix as the output factor matrix. Meanwhile, the rest of the modes becomes input modes, and their factor matrices become input factor matrices.

For each mode, the FLYCOO format assigns each nonzero tensor element to a tensor partition. Then embed the partition ids to each tensor element. Figure 2 illustrates the tensor format generation process for an example tensor with 3 modes and 8 elements. For a given output mode, FLYCOO divides the tensor into multiple partitions with an equal number of output mode indices. We call these partitions *super-shards*. Each PE processes the super-shards one by one. The number of intermediate values generated while processing each super-shard is proportional to the number of output mode indices in the super-shard. It ensures the intermediate values are combined in the FPGA internal memory to generate the output factor matrices. Despite generating many intermediate values similar to other tensor formats, our method enables combining them into output factor matrix elements before communicating the results to the FPGA external memory.

Due to the sparsity of input tensors, each super-shard contains a different number of nonzero tensor elements. We further divide each super-shard into *shards*, where the shards have the same number of nonzero tensor elements residing inside. Having the same size shards throughout the execution enables streaming memory access to the input tensor stored in FPGA external memory. It also leads to a static load balancing scheme described in Section 4.7 that fairly distributes the workload among PEs in all the output mode computations. Before partitioning the super-shards into shards, each super-shard is ordered based on Z-Morton order [12] using input mode indices of each nonzero element (see Figure 2). As a result, the nonzero tensor elements with the same input mode indices reside within the same shard. Z-Morton order recursively partitions multidimensional data into one dimension while preserving the locality of the data [12]. We adopt the Z-Morton order that lays out the elements along a recursive $Z$-shaped curve. The Z-Morton vector of each nonzero element is computed from the indices of each input mode by interleaving their binary coordinate values. For each nonzero tensor element, the input factor matrices are accessed based on their indices of the input modes. Hence, the proposed ordering improves the data locality in each shard while accessing input factor matrices.

*4.3.1 Tensor Format Definition.* Following the hypergraph representation described in Section 4.2, we can define the tensor format as follows. Consider a FPGA internal memory with enough space to store $m$ rows of the output factor matrix. For each output mode $n$, we partition the vertex set $V_n$ which represent the indices of the input tensor in mode $n$ into equal-size vertex sets $V_{n,0}, V_{n,1}, \ldots, V_{n,k_n-1}$, where $k_n = \frac{|V_n|}{m}$. Here, $|V_n|$ refers to the size of vertex set $V_n$. For $j = 0, 1, \ldots, (k_n - 1)$, each vertex set $V_{n,j}$ of size $m$ is defined as a subset of vertex set $V_n$. We call each vertex set $V_{n,j}$ as an *interval*. Then, we collect all the hyperedges incident on the vertices (i.e., nonzero tensor elements) in $V_{n,j}$ together as a super-shard, denoted by $SS_{n,j}$. Since real-world sparse tensors have high variance in the distribution of nonzero tensor elements, each super-shard contains a different number of hyperedges. This leads to load imbalance during the spMTTKRP computation. To address this, we further divide each super-shard into equal-sized sets called shards. Each super-shard $SS_{n,j}$ is further divided into $t_{n,j} = \lceil |SS_{n,j}|/g \rceil$ shards to fit in the FPGA buffers of size $g$. Here, we denote the $q$-th shard in $SS_{n,j}$ as $shard_{n,j,q}$. The total number of shards for mode $n$ is $\tau_n = \sum_{h=0}^{k_n-1} t_{n,h} \approx |T|/g$ for a tensor with $|T|$ nonzero tensor elements.

FLYCOO format maps each nonzero element in the tensor 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 4.3.1, a single nonzero element in the FLYCOO format requires approximately $N \times \log_2\left(\frac{|T|}{g}\right) + \sum_{h=0}^{N-1} \log_2 |V_h| + \beta_{\text{float}}$ bits, where $\beta_{\text{float}}$ is the number of bits needed to store the floating-point value of the nonzero tensor element. We encode $|s_i| \approx N \times \log_2\left(\frac{|T|}{g}\right)$, $|p_i| = \sum_{h=0}^{N-1} \log_2 |V_h|$, and $|val_i| = \beta_{\text{float}}$.

The FLYCOO format generation is a preprocessing task. Even though the remapping process is executed in real-time, the FPGA external memory spaces (i.e., memory layout) for each shard can be statically decided during the preprocessing time (see Section 4.7).

While FLYCOO and HiCOO [12] adopt similar tensor ordering strategies, such as Z-Morton ordering during the format generation, they have significant differences, including: (1) Intermediate value communication: According to the FLYCOO tensor format introduced in Section 4.3.1, each super-shard contains all the nonzero

tensor elements of an output factor matrix interval. Since each interval can be fit in the FPGA internal memory, each interval can be computed without communicating intermediate values generated during the computation to the FPGA external memory. However, HiCOO requires intermediate values to be communicated to the FPGA external memory. Since spMTTKRP is a memory-bound operation and FLYCOO avoids communicating intermediate values to the FPGA external memory, FLYCOO significantly reduces the overall execution time. (2) Tensor partitioning scheme and nonzero element distribution: HiCOO partitions contain an equal number of tensor indices for each mode, while the FLYCOO shards have the same number of nonzero tensor elements. The approach used by FLYCOO leads to load-balanced computation in a multi-PE accelerator, as discussed in Section 4.6. HiCOO partitions vary in size because nonzero elements are not distributed evenly across tensor indices. Hence, HiCOO leads to non-uniform partitioning of the tensor. (3) Applying Z-Morton ordering: FLYCOO applies Z-Morton ordering for each super-shard independent of each other along the indices of the input mode. HiCOO applies Z-Morton ordering once considering all the modes. FLYCOO enables better data reuse of input factor matrices while computing spMTTKRP with the use of the accelerator hardware (see Section 4.5).

## 4.4 Parallel Algorithm

We perform mode-by-mode super-shard computation. Since each super-shard can be executed independently by construction (see Section 4.3), the order of super-shard computation does not affect the outputs. Further, multiple super-shards can be executed in parallel.

Algorithm 2 shows the parallel algorithm for a super-shard for a given mode. The functions **Load** and **Store** correspond to loading and storing data from the FPGA external memory. The parallel algorithm consists of (1) spMTTKRP Computation and (2) Data Remapping for the next mode. These 2 stages are executed concurrently.

In Algorithm 2, all the factor matrices are accessed in row-major order. Hence, the factor matrices are stored in the FPGA external memory in row-major order.

At the beginning of Algorithm 2, a super-shard gets assigned to a PE for execution. In the spMTTKRP Computation, the shards that belongs to the same super-shard are loaded into an internal buffer one by one (Algorithm 2: line 7). After a shard is loaded, the tensor elements inside the shard are assigned for execution. First, the coordinates of the modes are extracted from the tensor element in FLYCOO format (Algorithm 2: line 10). Then the corresponding rows (based on input mode indices of the tensor element) of the input factor matrices are loaded into the PE from the FPGA external memory. After, the element-wise operations between the tensor element and the rows of the input factor matrices are performed inside the PEs (Algorithm 2: line 15-20). The PE maintains an internal memory buffer to store the intermediate values of the computation. Keeping this internal buffer size proportional to interval size, $|I|$ (see Section 4.3.1) ensures that all the intermediate values can be updated only using the internal memory of FPGA. After a super-shard is completely processed, the generated output interval is stored in the FPGA external memory (Algorithm 2: line 26).

After computing spMTTKRP for mode $n$, the accelerator should compute spMTTKRP for the subsequent mode (i.e., mode $(n+1) \mod N$). Nonzero tensor elements should be ordered according to the output mode to support our proposed parallel algorithm. Therefore, the tensor should be remapped according to the shard IDs of mode $(n+1) \mod N$ to compute the spMTTKRP of the upcoming mode. Hence, as the current mode runs, the tensor is remapped in parallel according to the shard IDs of the upcoming mode, allowing sequential execution of all the modes.

**Algorithm 2:** Computation of a *super-shard* in mode $n$

1 Input: Input factor matrices-set $\mathbf{Y} = \{Y_0, Y_1, ...Y_{N-1}\} \setminus \{Y_n\}$,
2 super-shard in mode $n$ $SS_{n,j} = \{\text{shard}_{n,j,0}, \ldots, \text{shard}_{n,j,k-1}\}$
**Result:** Store interval $I$ from output factor matrix $Y_n$,
remap and store each tensor element $x_i$
3 **Compute & On-the-fly Remap($SS$, Y):**
4 Initialize $I$ as a zero matrix
5 $y \leftarrow (n+1) \mod N$ // upcoming mode
6 **for** *each* $\text{shard}_{n,j,a}$ *in* $SS$ **do**
7 $\quad$ $\textbf{Buffer}_t \leftarrow \textbf{Load}(\text{shard}_{n,j,a})$
8 $\quad$ **for** *each element* $x_i = \langle s_i, p_i, val_i \rangle$ *in* $\textbf{Buffer}_t$ **do**
9 $\quad\quad$ $value \leftarrow val_i$
10 $\quad\quad$ $p_i = (c_0, \ldots, c_{N-1})$
11 $\quad\quad$ $shard\_ids$, $s_i = (b_0, \ldots, b_{N-1})$
12 $\quad\quad$ $z \leftarrow b_y$
13 $\quad\quad$ // $\ell$ is a vector of size $R$. Each element initialized to 1
14 $\quad\quad$ $\ell \leftarrow 1$
15 $\quad\quad$ **for** *each input mode* $w \in \{0, \ldots, N-1\} \setminus \{n\}$ **do**
16 $\quad\quad\quad$ $vec \leftarrow \textbf{Load}$(row $c_w$ from $w^{\text{th}}$ factor matrix)
17 $\quad\quad\quad$ **for** *each rank* $r$ *in* $R$ ***parallel*** **do**
18 $\quad\quad\quad\quad$ $\ell(r) \leftarrow \ell(r) \times vec(r)$
19 $\quad\quad$ **for** *each rank* $r$ *in* $R$ ***parallel*** **do**
20 $\quad\quad\quad$ $I(c_n, r) \leftarrow I(c_n, r) + value \times \ell(r)$
21 $\quad$ // Inside Remap_Cache_Buffer
22 $\quad$ $\text{shard\_collector}_{(b_y)} \leftarrow \textbf{Update}(\text{shard\_collector}_{(b_y)} \cup x_i)$
23 $\quad$ **if** *Remap_Cache_Buffer full* **then**
24 $\quad\quad$ **for** *each collected* $shard\_collector_{(b_y)}$ *in* *Remap_Buffer* **do**
25 $\quad\quad\quad$ $\textbf{Store}(\text{append shard\_collector}_{(b_y)} \text{ to } shard_{y,b_y})$
26 $\text{Buffer}_{\text{output}} \leftarrow I$
27 $\textbf{Store}(\text{Buffer}_{\text{output}} \text{ to n}^{th} \text{ factor matrix})$
28 **for** *each collected* $shard\_collector_{(b_y)}$ *in* *Remap_Cache_Buffer* **do**
29 $\quad$ $\textbf{Store}(\text{append shard\_collector}_{(b_y)} \text{ to } shard_{y,b_y})$

The on-the-fly remapping for the upcoming mode is performed as follows: First, the shard ID of the subsequent mode to be executed is extracted from each tensor element. Then, the tensor elements with the same shard ID of the subsequent mode are collected together inside the Remap_Cache_Buffer using the update function (Algorithm 2: line 22). Remap_Cache_Buffers are not large enough to store all shards; therefore, as the Remap_Cache_Buffer becomes full, we store partially aggregated shards (i.e., shard collectors) in

the corresponding shard locations in the FPGA external memory (Algorithm 2: lines 23-25 & 28-29).

The total computations per mode in the proposed Algorithm 2 is $N \times |T| \times R$. Here, the factor $N$ comes from $N-1$ multiplications and the addition we perform per nonzero tensor element. The accelerator loads all the nonzero tensor elements and stores the factor matrix of the output mode. Meanwhile, it also stores the remapped tensor optimized for the next mode. Hence it requires a total of $2 \times |T| + I_{out} \times R$ memory transfers. Here, $I_{out}$ represents the length of the output dimension ($I_{out} \in \{I_0, I_1, \ldots, I_{N-1}\}$). For factor matrices of rank $R$ with no data reuse, the total factor matrix elements transferred per mode is $(N-1) \times |T| \times R$. Hence, the total amount of data transferred from the FPGA external memory per mode is $2 \times |T| + (N-1) \times |T| \times R + I_{out} \times R$.

## 4.5 FPGA Design

Algorithm 3 is designed for multi-SLR [25] datacenter FPGAs. These FPGAs contain a large number of DSPs and multiple DRAM memories to provide high compute power and memory bandwidth.

**Algorithm 3:** FLYPAR: FLYCOO-based Parallel algorithm

```
1  Input: Input factor matrices-set Y = {Y_0, Y_1, ...Y_{N-1}},
2  super-shards of mode 0, {H_0} = {SS_{0,j} : ∀ j}
3  Output: Updated factor matrices-set Ŷ = {Ŷ_0, Ŷ_1, ...Ŷ_{N-1}}
4  for each mode n = 0, ..., N − 1 do
5      while H_n ≠ ∅ do
6          for each ACCEL_k parallel do
7              if ACCEL_k is idle then
8                  Fetch super-shard SS ∈ H_n in DRAM_k to
                     ACCEL_k
9                  H_n ← H_n \ {SS}
10                 (H_{(n+1) mod N}, Ŷ) ← Compute &
                     On-the-fly Remap(SS, Y) // Algorithm 2
11     Wait(All ACCELs are idle)
```

The proposed accelerator contains multiple custom hardware units called ACCELs. For a FPGA with $p$ DRAMs, we employ $p$ ACCELs where each ACCEL is directly connected to a DRAM via memory controller as shown in Figure 3 (a). ACCELs communicate with each other using ACCEL routers in a ring interconnection. Inter-SLR switches in the FPGA [24] are used to connect neighboring ACCELs in different Super Logic Regions (SLRs).

Our design has the following features: (1) custom multi-pipeline PEs to support concurrent element-wise tensor multiply and add operations, (2) internal buffers in each pipeline to ensure all the intermediate values remain in the FPGA internal memory while processing a super-shard, (3) use Direct Memory Accesses (DMAs) to load shards and store output factor matrix rows as bulk transfers while optimally using the FPGA external memory bandwidth, (4) multi-cache subsystem to exploit locality while loading input factor matrices, and (5) tensor remapping module to support on-the-fly remapping.

Each ACCEL is assigned a super-shard during the execution time in which the super-shard resides in its directly connected DRAM as described in FLYPAR: Parallel spMTTKRP Accelerator Algorithm based on FLYCOO (Algorithm 3). The $p$ ACCELs process $p$ super-shards in parallel following the parallel algorithm in Section 4.4. All the ACCELs synchronize at the end of the computations in each output mode. Our workload distribution (see Section 4.6) ensures PE idle time is minimized. As shown in Figure 3 (b), each ACCEL has a Processing Engine (PE) and a memory controller. Each PE executes the element-wise computation on the tensor elements. The memory controller manages the data flow between the PEs and their external memory.

*4.5.1 Processing Engine (PE).* PE is the compute unit inside an ACCEL. It can concurrently process $q$ input tensor elements in each clock cycle using $q$ ($q \geq 1$) processing pipelines as shown in Figure 3 (c). A PE processes a shard with the following 3 steps: (1) a pipeline reads an element of the shard from the tensor buffer, (2) pipeline extracts the coordinates of the tensor element and requests corresponding input factor matrices from the caches in the memory controller, (3) When all the requested input factor matrix rows become available inside the pipeline, it performs element-wise compute as indicated in Algorithm 2 and stores the corresponding partial value in its matrix buffer, (4) after computing all the elements in a super-shard, PE computes the final output factor matrix rows by adding the partial sums in the partial matrix buffers in all the pipelines with the same output mode indices using PE's adder tree. For a PE with $q$ pipelines, the adder tree has $\log_2(q)$ stages with $q(q+1)/2$ total adders, (5) outputs the output factor matrix rows to the output matrix buffer inside the memory controller. This process generates a interval of the output factor matrix that corresponds to the input super-shard. At the end of the process, it is transferred to the FPGA external memory as a bulk transfer using DMA inside the memory controller.

*4.5.2 Memory Controller.* Figure 3 (d) shows the details of the memory controller (MC). It consists of a cache subsystem, Direct Memory Accesses (DMA) module, a Tensor Remapper Module (TRM), an ACCEL router, and an external memory interface. The cache subsystem exploits the spatial and temporal locality of the input factor matrix accesses enabled by the proposed FLYCOO format. Each cache is a set-associative cache with the Least Recently Used (LRU) cache-line replacement policy. Multiple input factor matrices can share a cache. The number of caches inside a MC, the number of input factor matrices that share a single cache, and cache size are chosen based on available FPGA resources. DMAs can efficiently access sequential data as bulk transfers. DMAs load shards to the tensor buffer and store the output factor matrix from the output matrix buffer as bulk transfers. By keeping the size of a shard as same as the tensor buffer size and the interval size equal to the output matrix buffer size, we avoid data overflows inside the DMAs. The DMA uses double buffers to overlap the communication time of the shards from the FPGA external memory with the compute time of the PEs. On-the-fly remapping is supported by the tensor remapping module (TRM) in the MC. When the PE requests a tensor element from the tensor buffer, a copy of the same tensor element is passed to the TRM using a shared bus, as shown in Figure 3 (d). TRM extracts the shard ID of the upcoming output mode of the spMTTKRP in the CPD decomposition iteration (see Algorithm 1). The TRM collects the tensor elements with the same shard ID until it fills a complete row of the TRM buffer and transfers it to the FPGA external memory.

ACCEL router maintains the super-shard scheduling while controlling the data flow between PE, DRAM, and neighboring ACCELs.

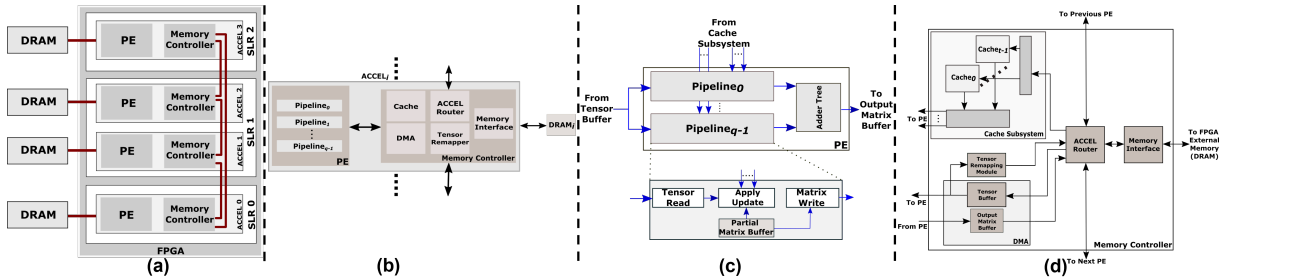



**Figure 3: FPGA design: (a) Overall FPGA design, (b) architecture of an ACCEL, (c) Processing Engine, (d) memory controller**

ACCEL router resides inside the memory controller of each ACCEL. Each ACCEL Router is connected to neighboring ACCEL routers as a ring (see Figure 3 (b)). Each ACCEL router communicates data as data packets between its neighboring ACCEL routers in the ring. It uses large data buses between ACCELs while maximally using inter SLR [24] routing resources to transfer multiple data packets simultaneously. The accelerator maintains a virtual address space to keep track of the memory addresses of each tensor shards of all the modes and the intervals of all the factor matrices. The accelerator also maintains memory pointers to identify the location to be read or stored in a shard to support the proposed tensor remapping. The virtual addresses are generated during the tensor generation process and stored in the FPGA external memory as initial meta-data.

*4.5.3 ACCEL Router.* Each ACCEL Router performs Algorithm 4 that, (1) loads shards from directly connected external DRAM, (2) stores the output factor matrix rows depending on the PE requests, (3) loads the input factor matrix rows depending on the PE requests from different ACCELs, and (4) stores the remapped tensor elements in the corresponding shard location. The destination location of the remapped tensor element can be in any of the external DRAMs connected to the FPGA.

The data router loads each shard from its directly connected DRAM and forwards it to Tensor Buffer inside DMA (Algorithm 4 line 2-8). Additionally, memory access requests are received from Cache Subsystem (Algorithm 4 line 9-10) and Tensor Remapping Module (Algorithm 4 line 11-12) to the ACCEL router. In addition, data packets are forwarded from the previously connected ACCEL router (Algorithm 4 line 13-14). For the input factor matrix row read requests, if the factor matrix is in the directly connected DRAM, it is loaded from the DRAM and forwarded either to the previous ACCEL router in the ring or to the cache subsystem. Otherwise, the request is forwarded to the next ACCEL router in the ring (Algorithm 4 line 15-26). When a tensor remapping request is reached to the ACCEL, if the corresponding shard of the nonzero tensor element is located inside the directly connected DRAM, it is stored in the DRAM. Otherwise, remap request with the tensor element is forwarded to the next ACCEL router in the ring (Algorithm 4 line 27-32).

We use memory interface IPs [7, 26] to maintain the low-level signals (e.g., refresh, and pre-charge) between DRAMs and FPGA.

**Algorithm 4:** Data Routing Algorithm for $ACCEL_i$

```
1  Routing (ACCEL_i, n): // For output mode n
2  if PE is Idle then
3      SS = Get Active super-shard ID
4      shard = Next shard (SS)
5      if all shards of SS processed then
6          interval data = Get output data from PE
7          Store DRAM_i(Get Interval (SS), interval data)
8          SS = Get Next super-shard ID
9  if Factor Matrix Row k is requested by a Cache Miss then
10     {type, src_accel, w, info}← {fm_read, i, n, k}
11 if else remapped tensor y available from Tensor Remapper
       then
12     {type, src_accel, w, info}← {remap_store, i, n, y}
13 if else data_packet from from ACCEL_{i-1} then
14     {type, src_accel, w, info}←data_packet
15 if type = fm_read then
16     I = Extract Interval (info, w)
17     if I is in DRAM_i then
18         value = Read DRAM_i(w, info)
19         if src_ACCEL = ACCEL_i then
20             Forward Cache(value, w, info)
21         else Forward ACCEL_{i-1} ({fm_write, src_accel, n, w,
             info})
22     else Forward ACCEL_{i+1} ({fm_read, src_accel, n, w,
         info})
23 if type = remap_store then
24     S = Get Next Mode shard ID (y)
25     if S is in DRAM_i then
26         Store DRAM(info, S)
27     else Forward ACCEL_{i+1} ({remap, src_accel, n, w, info})
```

## 4.6 Load balancing

Since the accelerator performs spMTTKRP computation for a single output mode at a time, we consider load balancing the total computation mode by mode. In a given mode, the total number of computations corresponding to a super-shard is proportional to the number of nonzero tensor elements in the super-shard. Since each super-shard is further partitioned into shards with same number of tensor elements, the total number of computations also proportional to the number of shards in a super-shard. Each super-shard contains a different number of shards depending on the sparsity of the tensor. We use a greedy approach to distribute the super-shards and perform the spMTTKRP computation. For each mode, the proposed method evenly distributes the total workload among the ACCELs. Suppose a FLYCOO tensor of size $T$ is partitioned into super-shards $\{SS_{n,j} : \forall\, j\}$ for a mode $n$. Let $K_{n,j}$ be the number of

shards in the super-shard $SS_{n,j}$. We reorder the indices of super-shards so that $K_{n,j} \geq K_{n,j'}$ if $j < j'$, so that they are sorted in descending order of number of shards. Each super-shard $SS_{n,j}$ is iteratively assigned to the ACCEL that is currently the least heavily loaded (i.e., with the least number of shards). We perform the above operation for all the modes of the tensor.

For a given output mode, let $K_{\max}$ be the largest number of shards assigned to a single ACCEL and $K^*_{\max}$ be the value of $K_{\max}$ in an optimal shard distribution among the ACCELs. Then our proposed greedy approach above guarantees that $K_{\max} \leq 4/3 \cdot K^*_{\max}$ [4]. The proof of this approximation guarantee is shown by induction in the following.

Theorem 4.1. *For a given mode n, based on our greedy approach, the number of shards assigned to each ACCEL is at most* $4/3 \cdot K^*_{max}$.

Proof. Assume the theorem is true for at most $k' < k$ super-shards. Suppose there are $k$ super-shards, and the $\ell$-th super-shard is the last one that is assigned to the ACCEL $A$ containing the most shards. Consider two cases: (i) If $\ell < k$, then since the $k$-th super-shard is not assigned to $A$, removing it doesn't affect the result. Therefore, the theorem follows immediately by the induction hypothesis on the first $k-1$ super-shards. (ii) If $\ell = k$, then $K_{\max} \leq K^*_{\max} + K_{n,k}$. If $K_{n,k} \leq K^*_{\max}/3$ then the theorem holds immediately. Suppose $K_{n,k} > K^*_{\max}/3$, then $K^*_{\max} < 3K_{n,k}$, so each ACCEL has been assigned either one or two super-shards, which is in fact an optimal assignment: exchanging any two super-shards in different ACCELs will increase the largest number of shards in an ACCEL. So $K_{\max} = K^*_{\max}$ in this case. □

## 4.7 Tensor Remapping

As described in Section 4.4, tensor elements are remapped according to the shard IDs of the upcoming mode. Theorem 4.2 shows that it only requires $2 \times |T|$ FPGA external memory to perform on-the-fly tensor remapping. Note that the mode-specific formats require the number of tensor copies proportional to the number of modes of an input tensor. Theorem 4.3 shows the tensor layout generated through the proposed tensor remapping technique is load balanced across all ACCELs, and the tensor data is locally available to each ACCEL in its directly connected external DRAM memory during each output mode computing time.

The accelerator requires the external memory address pointers for each shard to identify the destination memory address of nonzero elements while tensor remapping. The memory address pointers for the initial position of each shard are computed during the tensor format generation time. These memory address pointers are updated as the shards are getting filled during the runtime. During the remapping, the accelerator only requires to keep track of the address pointers corresponding to a single mode. Also, following the routing algorithm discussed in Section 4.5.2, each ACCEL only requires to keep track of the shards mapped into its directly connected DRAM.

Theorem 4.2. *The total FPGA external memory required to store an input tensor T in the FLYCOO format for our proposed algorithm FLYPAR is* $2 \times |T|$, *where* $|T|$ *is the size of the tensor, independent of the number of modes.*

Proof. Consider a FPGA external memory space of size $B = 2 \times |T|$, divided to $B[0]$ and $B[1]$. If the current mode ($n$) being executed is even (i.e., $n = 2, 4, 6 \ldots$), then it uses $B[0]$ to keep the tensor copy ordered according to mode $n$ shard IDs. $B[1]$ is used to keep the tensor copy ordered according to shard IDs of mode $(n+1) \mod N$ during the on-the-fly remapping. Similarly, if the current output mode being executed is odd (i.e., $n = 1, 3, 5 \ldots$), it uses space $B[1]$ to keep the tensor copy ordered according to mode $n$. $B[0]$ is used to keep the tensor copy ordered according to mode $(n+1) \mod N$. As a result, we only need $|B| = 2 \times |T|$ space to store the tensor to support the algorithm FLYPAR. □

Theorem 4.3. *Algorithm FLYPAR performs spMTTKRP for all the modes of an input tensor such that the computation is load balanced across all ACCELs and the tensor data is available for each ACCEL in its directly connected external DRAM memory for all the output mode computations.*

Proof. In the FLYCOO format, each tensor element contains its shard IDs for all the modes (see Section 4.3). For an input tensor with $N$ modes, during the computation of mode $n-1$, the on-the-fly-remapping remaps each nonzero tensor element according to the shard IDs of mode $n$ ($0 \leq n < N$). The data routing algorithm (Algorithm 4.5.3) allows this by remapping each tensor element to the DRAM directly connected to the PE. Hence, at the beginning of the spMTTKRP computation for an output mode $n$ (Algorithm 2), all the nonzero elements in the tensor have been assigned to the shards of mode $n$ $\{shard_{n,j,z} : \forall\, j, z\}$. Therefore, the tensor data is available for each ACCEL in its directly connected external DRAM memory for all the output mode computations.

The load balancing approach in Section 4.6 proves that the computation is load balanced if each tensor element is in its assigned shard for an output mode $n$. According to the on-the-fly remapping, at the beginning of the computation, all nonzero elements in the tensor are in the shards of mode $n$. Since we perform the overall computation mode-by-mode, the overall computation is load balanced for all the modes. □

Theorem 4.4 proves that the remapping cost introduced in FLYPAR is insignificant compared to total FPGA external memory accesses while accessing input factor matrices in each mode.

Theorem 4.4. *In any mode, without data reuse, the ratio of the total amount of data transferred in remapping to that for accessing the input factor matrices is* $O\left(\frac{1}{(N-1)R}\right)$, *where N is the number of modes and R is the rank of the factor matrices.*

Proof. For a given mode, since every nonzero element is remapped for spMTTKRP in the upcoming mode, the total amount of data transferred per mode for remapping is equal to the size of the total number of nonzero elements ($|T|$). With no data reuse, the total amount of data transferred per mode for the input factor matrices is $(N-1) \times |T| \times R$ (see Section 4.4). Therefore, the ratio of the amount of data transferred for the remapping to that for accessing input factor matrices is $O\left(\frac{1}{(N-1)R}\right)$. □

## 4.8 Super-shard Scheduling

After distributing super-shards among ACCELs in Section 4.6, we further optimize the memory access cost between distinct super-shards assigned to each ACCEL in every mode by statically scheduling these super-shards. The objective of the super-shard scheduling is to reuse cached rows of input factor matrices across multiple super-shards as much as possible. The first step to achieving this is to create a weighted directed complete graph $\Gamma = (\Gamma_V, \Gamma_E, \Gamma_W)$ whose vertex set $\Gamma_V$ consists of the super-shards, which are assigned to the target ACCEL. Here, the weight $w_e \in \Gamma_W$ of each edge

$e \in \Gamma_E$ represents the number of rows of factor matrices that any two super-shards have in common. After constructing the complete graph $\Gamma$, our goal is to find a maximum weighted Hamiltonian path $P$ in $\Gamma$. The order of vertices in the Hamiltonian path $P$ represents the scheduling of the super-shards assigned to each ACCEL. It leads to efficient cache utilization by maximizing the reusability of the common input factor matrices among super-shards after loading them into the caches. Since the maximum weighted Hamiltonian path problem is NP-complete [1], we use a greedy heuristic by adding one edge at a time, as summarized in Algorithm 5. The schedule computed as a pre-processing step for each input tensor.

**Algorithm 5:** super-shard Scheduling

```
Input: A weighted directed complete graph Γ = (Γ_V, Γ_E, Γ_W)
       consisting of the super-shards
Output: A Hamiltonian path P representing the scheduling
        of the super-shards assigned to each ACCEL
P ← ∅
Sort Γ_E in descending order of Γ_W
for each edge e = (u, v) ∈ Γ_E do
    if deg_+(u) = 0 & deg_-(v) = 0 & no cycle in P ∪ {e} then
        P ← P ∪ {e}
return P
```

# 5 EVALUATION

## 5.1 Experimental Setup

*5.1.1 FPGA Platform.* We implement our hardware design on Xilinx Alveo U250 Data Center Accelerator Card [24] using Verilog HDL. This Alveo Card consists of 4 super logic regions (SLRs) [25]. The SLRs connected to DRAM modules through memory interface IPs [26]. Simulation, synthesis, and place and route are performed using Xilinx Vivado Design Suite 2020.2 [27].

*5.1.2 Datasets.* We use the sparse tensors from real-world applications shown in Table 2. All the tensors are from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) dataset[19]. The selected datasets have tensors with different shapes, sizes, and sparsities.

*5.1.3 FLYCOO Format Generation Time.* Since we implement the preprocessing step using Python libraries, our preprocessing algorithm is substantially slower than C/C++ implementations. There are no meaningful comparisons of the execution time between the pre-processing and processing steps with C/C++-based implementations in related works. Hence in this work, we do not show the implications of preprocessing costs which are executed offline.

*5.1.4 Performance Model Simulator.* Performance Model Simulator (PMS) is used for optimizing the accelerator configuration. PMS can estimate the number of clock cycles spent on computing spMTTKRP for a given input tensor. PMS models each hardware module in the accelerator design at the cycle level. The PMS can further estimate the total FPGA internal memory requirement and the DSP usage for a given accelerator configuration.

Our objective is to use PMS to (1) identify the best set of hardware parameters of the accelerator to obtain the least average execution time for a given collection of datasets (see Section 5.1.5), and (2) evaluate the impact of various hardware modules in the proposed design and algorithmic optimizations due to the versatility of PMS. Obtaining module-wise performance results is effortless with PMS compared with the actual hardware implementation.

The system designer has to provide the following inputs to the PMS: (1) resources of target FPGA (i.e., total DSPs, BRAMs, and URAMs and data width of memory interface), (2) design parameters (i.e., number of pipelines per ACCEL, DMA buffer sizes, number of caches, number of cache lines, associativity of a cache, and number of factor matrices shared by a cache), (3) algorithm parameters (i.e., rank of the factor matrices, size of a shard, and size of an interval), and (4) input tensor parameters (i.e., number of modes, and length (dimension size) of each mode).

**Table 2: Characteristics of the sparse tensors**

| Tensor | Shape | #NNZs | Density |
|---|---|---|---|
| NELL-1 | $2.9M \times 2.1M \times 25.5M$ | $143.6M$ | $9.1 \times 10^{-13}$ |
| NELL-2 | $12.1K \times 9.2K \times 28.8K$ | $76.9M$ | $2.4 \times 10^{-05}$ |
| PATENTS | $46 \times 239.2K \times 239.2K$ | $3.6B$ | $1.4 \times 10^{-03}$ |
| LBNL | $1.6K \times 4.2K \times 1.6K \times 4.2K \times 868.1K$ | $1.7M$ | $4.2 \times 10^{-14}$ |
| DELICIOUS | $532.9K \times 17.3M \times 2.5M \times 1.4K$ | $140.1M$ | $4.3 \times 10^{-15}$ |

The overall execution time (in cycles) reported by PMS is validated using FPGA run-time results. Figure 5 compares the total execution time of each tensor on FPGA and the PMS. The behavior of DRAM technology and packet routing on ACCEL routers can not be accurately simulated using a software model. Hence PMS shows below 10% error compared with the actual hardware.

We performed an extensive parameter search to select the configuration of the accelerator. The goal is to identify the hardware parameters with the least average execution time for all the datasets in Table 2. spMTTKRP is executed per single iteration for each dataset.

*5.1.5 Optimizing Accelerator Configuration.* The execution time depends on the total number of clock cycles and FPGA operating frequency ($f_{FPGA}$). We use PMS to estimate the number of clock cycles for computing spMTTKRP. We determine the average number of FPGA cycles ($C_{avg}$) by taking the PMS results for all the targeted tensors. PMS can also estimate the total FPGA internal memory requirement and the DSP usage for a given accelerator configuration. We can choose an accelerator configuration that provides the least $C_{avg}$ and is subject to the resource constraints of the FPGA using PMS. First, we focus on identifying the shard and interval sizes that are suitable for the target FPGA. These sizes directly correlate with the accelerator buffer sizes. We use PMS to identify the shard and interval sizes that fit the target FPGA and take the minimum number of FPGA clock cycles. Using PMS, we select 10 suitable configurations with a minimum of $C_{avg}$ and fit the target FPGA resources. Then we use Place and Route (P&R) using Xilinx Vivado to obtain the $f_{FPGA}$ of each selected accelerator configuration. Finally, we pick the configuration with the least $C_{avg}/f_{FPGA}$ configuration. Table 3 shows the selected configuration for our targeted dataset and FPGA. We observe that using more FPGA resources leads to lower FPGA operating frequency which reduces the overall performance of the design.

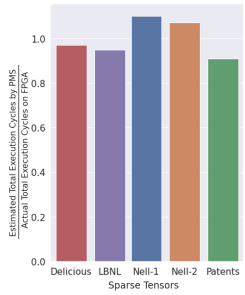



**Figure 5: PMS accuracy**

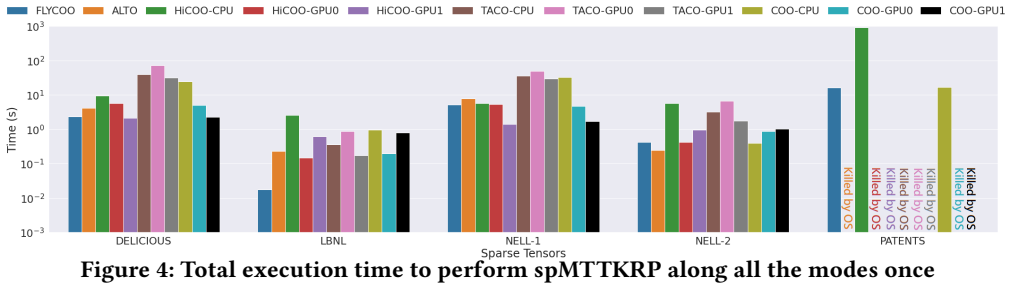



**Figure 4: Total execution time to perform spMTTKRP along all the modes once**

**Table 3: Module Parameter Configuration**

| Module | Parameter | Configuration |
|---|---|---|
| PE | No. of pipelines | 16 |
| | Partial Matrix Buffer size | 16 KB |
| Cache subsystem | No. of caches | 2 |
| | Set-associativity | 4 |
| | No. of cachelines | 4096 |
| | cache line width | 64 B |
| DMA | Tensor Buffer | 64 KB |
| | Output Matrix Buffer | 16 KB |
| | Tensor Remapping Buffer | 64 KB |

**Table 4: Resource utilization of the selected design**

| LUT | FF | BRAM | URAM | DSP |
|---|---|---|---|---|
| 37.23% | 24.52% | 20.84% | 20.52% | 25% |

*5.1.6 Baselines.* The baseline experiments are conducted on an Intel Xeon Gold 5120 CPU, an NVIDIA RTX 3090 GPU, and an NVIDIA RTX A6000 GPU. Platform specifications are summarized in Table 5.

**Table 5: Specifications of the platforms**

| Platform | CPU Intel Xeon Gold 5120 | GPU0 NVIDIA RTX 3090 | GPU1 NVIDIA RTX A6000 | FPGA Xilinx Alveo U250 |
|---|---|---|---|---|
| Technology | Intel 14 nm | Samsung 8 nm | Samsung 8 nm | TSMC 16 nm |
| Frequency | 2.20 GHz | 1695 MHz | 1410 MHz | 230 MHz |
| Peak Device Performance | 14.9 GFLOPS | 35.6 TFLOPS | 38.71 TFLOPS | 0.6 TFLOPS |
| Device Internal Memory | 19.25 MB L3 Cache | 6 MB L2 Cache | 6 MB L2 Cache | 54 MB |
| External Memory Bandwidth | 107.3 GB/s | 936.2 GB/s | 768 GB/s | 77 GB/s |

We evaluate our work against mode-specific CSF format[9], mode-specific COO format[11], mode-agnostic HiCOO format[12] on both CPU and GPU platforms. We also evaluate the ALTO format [5] on the CPU. For the COO format, we use the library ParTI[11]. HiCOO is the required input parameter *block size* for its partitioning scheme, similar to the shard size in our proposed work. Similar to literature [5, 12], we use a block size of 128 to evaluate the performance of HiCOO. We ran OpenMP-enabled HiCOO CPU implementation using all the CPU threads. We evaluate the performance of mode-specific CSF formats using TACO[9]. We optimize the TACO code to our target CPU and GPU platforms using their command-line tool[9].

## 5.2 Overall Performance

Following the baselines[5, 12], we set the tensor rank ($R$) as 16. The interval size ($I$) is set to the same size as the output matrix buffer size, which is determined during optimizing the accelerator. We also keep the shard size equal to the tensor buffer size. Our experiments are conducted on the actual FPGA hardware. The tensor format generation is a one-time preprocessing step. As in the baselines[9, 12], we do not include the tensor generation time in the overall execution time.

Figure 4 displays the total execution time while computing spMTTKRP along all the modes. Our work is the only implementation that delivers consistent performance across all the modes of the datasets. Our work achieves an average of 8.8× and 3.8× speedup compared with the baseline CPU and GPU implementations. Table 6 summarizes the average speedup of each baseline and our work compared to COO-CPU[11].

We observe that LBNL generates a large number of intermediate values. Therefore, baselines that hold intermediate data in the external memory have to load and store intermediate data several times while computing a single row of the output factor matrix. Since our approach avoids partial outputs being stored in the external memory, it reduces the communication time between the FPGA and its external memory. LBNL also has limited data reuse while accessing the rows of input factor matrices. Our cache system exploits the data reuse of the input factor matrices of LBNL better than the other datasets. This leads to a significant execution time reduction for the LBNL dataset compared with the rest of the benchmarks. When executing PATENT, the OS kills ALTO, TACO CPU, and all GPU implementations because the data generated during compute time does not fit in the device external memory. The selected devices did not have sufficient external memory to hold the intermediate values generated during the execution time of PATENT for our baselines.

## 5.3 Impact of Tensor Remapping

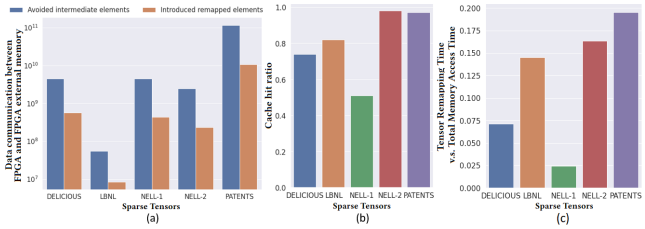



**Figure 6: (a) Number of intermediate elements avoided being transferred to FPGA external memory by tensor remapping vs. total tensor elements remapped, (b) Cache-hit ratio while accessing rows of input factor matrices, (c) Tensor remapping time over total memory access time**

The proposed parallel algorithm, FLYPAR (Algorithm 3), avoids intermediate values being transferred to the FPGA external memory at the additional cost of tensor remapping. Note that it introduces additional data transfers during on-the-fly tensor remapping. Figure 6 (a) shows a comparison between the amount of remapped tensor elements transferred to the FPGA external memory and the amount of data transfers (to the FPGA external memory) avoided by combining the intermediate values inside the FPGA internal memory for every super-shard. Our results indicate that FLYCOO

**Table 6: Comparison of baselines on their targeted platforms and this work**

| Tensor Formats | FLYCOO FPGA | ALTO CPU | HiCOO CPU | HiCOO GPU0 | HiCOO GPU1 | TACO CPU | TACO GPU0 | TACO GPU1 | COO CPU | COO GPU0 | COO GPU1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Speedup (over COO-CPU) | 18.0 | 3.9 | 2.2 | 4.4 | 9.2 | 1.0 | 0.5 | 1.90 | 1.0 | 4.2 | 7.8 |
| Device Peak Bandwidth (GB/s) | 77 | 107.3 | 107.3 | 936.2 | 768 | 107.3 | 936.2 | 768 | 107.3 | 936.2 | 768 |
| Device Peak Performance (TFLOPS) | 0.60 | 0.0149 | 0.0149 | 35.6 | 38.71 | 0.0149 | 35.6 | 38.71 | 0.0149 | 35.6 | 38.71 |

with FLYPAR reduces the FPGA external memory traffic by 9.2× on average. Figure 6 (b) shows the cache hit rate while accessing input factor matrices. It confirms that FLYCOO significantly increases data reuse. Figure 6 (c) compares the total tensor remapping time with the time spent on the rest of the memory accesses. The results show that even though the input factor matrices are cached in the FPGA, the tensor remapping cost is still insignificant compared with the rest of the memory access cost.

## 5.4 Impact of Memory Controller

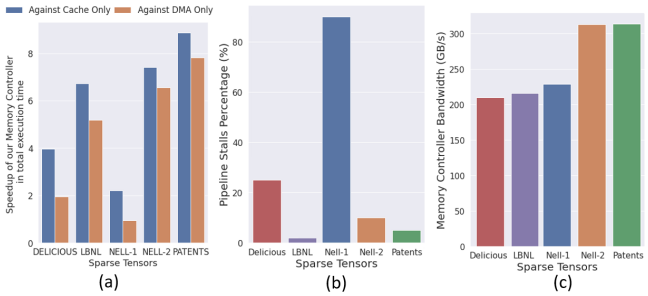



**Figure 7: (a) Improvement of proposed MC in execution time, (b) Number of pipeline stalls wrt. total number of pipeline requests, (c) Sustained MC bandwidth**

We compare our proposed memory controller (MC[cache + DMA]) with 2 other alternative memory controller designs, namely cache-only (MC[cache only]) and DMA-only (MC[DMA only]) memory controllers. As the name implies, MC[cache only] uses only caches. The DMA in our proposed MC[cache + DMA] is replaced by a cache of the same size. Similarly, MC[DMA only] replaces the caches in the original design with a DMA of the same size. A comparison of the total execution time for each dataset is shown in Figure 7 (a). The datasets with higher data locality while loading input factor matrices show significant execution time reduction with MC[cache + DMA] compared with MC[DMA only]. Replacing DMAs with caches hinders bulk data transfers when loading, storing, and remapping shards. As a result, MC[cache only] takes longer to perform the above memory operations than MC[cache + DMA]. With MC[cache + DMA], total execution time is improved by 4.5x and 5.8x compared with MC[cache only] and MC[DMA only].

Figure 7 (b) shows the total number of pipeline stalls in the system compared with the total number of operations executed by all the pipelines. Due to irregular external memory access to the input factor matrices, NELL-1 has a significant percentage of pipeline stalls. The rest of the datasets has less than 25% pipeline stalls during computing spMTTKRP on all modes.

Figure 7 (c) shows the sustained memory controller bandwidth while executing spMTTKRP. The memory controller bandwidth is defined as the total amount of data communicated between PEs and memory controllers over the total spMTTKRP execution time for each input tensor. Our memory controller achieves sustained memory controller bandwidth that is more than twice the peak DRAM bandwidth.

## 5.5 Impact of Rank of the Factor Matrices

In this section, we evaluate the impact of the rank ($R$) of the factor matrices on total execution time. We alter $R$ to 8, 16, and 32 as they are the most common sizes used in tensor decomposition applications [2, 15, 18, 22]. Results from PMS (see Section 5.1.5) suggest that we can use the same accelerator parameters listed in Table 3 by adjusting the interval size of the tensor format.

Figure 8 shows the total execution time for different datasets over $R$ = 8, 16 and 32. With increasing R, the number of elements in a row in output interval and input factor matrices increases. It leads to the following 2 important observations: (1) as $R$ increases, the caches need multiple cache-line requests to load a single row of factor matrices, and (2) as $R$ decreases, multiple neighboring rows of factor matrices can fit in a single cache line. In the case of $R$ = 8, the contribution of each subsequent factor matrix row to the computation is very low even if multiple neighboring factor matrices are loaded in one cache line load. Hence we can not observe a significant reduction in total execution time. When $R$ = 16, a single cache line request loads a single row of the input factor matrix. On the other hand, when $R$ = 32, a single input factor matrix row occupies multiple cache lines. Hence a cache miss on a single input factor matrix row results in multiple cache line misses.


**Figure 8: Impact of rank ($R$)**

## 6 CONCLUSION AND FUTURE WORK

We proposed the FLYCOO format to reduce the total memory access time on FPGA. The on-the-fly tensor remapping supported by FLYCOO avoids storing intermediate values in the external memory. Using FLYCOO and the proposed FPGA accelerator, we outperform existing benchmarks on a variety of real-world sparse tensors.

In the future, we plan to parallelize the preprocessing algorithm for the tensor format generation. The algorithmic optimizations discussed in Section 4 can be adapted to general purpose hardware such as CPU and GPU. Using our proposed algorithmic optimizations, CPU and GPU can accelerate spMTTKRP by efficiently using their limited internal cache and external memory bandwidth.

## ACKNOWLEDGMENTS

This work is supported by the National Science Foundation (NSF) under grants OAC-2209563, CNS-2009057 and in part by DEVCOM Army Research Lab (ARL) under ARL-USC collaborative grant DIRA-ECI:DEC21-CI-037.

# REFERENCES


[1] Alfred V Aho and John E Hopcroft. 1974. The design and analysis of computer algorithms. (1974).

[2] Zhiyu Cheng, Baopu Li, Yanwen Fan, and Yingze Bao. 2020. 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, 3292–3296.

[3] Sofia Fernandes, Hadi Fanaee-T, and João Gama. 2020. Tensor decomposition for analysing time-evolving social networks: An overview. *Artificial Intelligence Review* (2020), 1–26.

[4] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[5] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: Adaptive Linearized Storage of Sparse Tensors. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 404–416. https://doi.org/10.1145/3447818.3461703

[6] David Hong, Tamara G. Kolda, and Jed A. Duersch. 2020. Generalized Canonical Polyadic Tensor Decomposition. *SIAM Rev.* 62, 1 (2020), 133–163. https://doi.org/10.1137/18M1203626 arXiv:https://doi.org/10.1137/18M1203626

[7] Intel. 2022. External Memory Interfaces Intel Arria 10 FPGA IP User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-20115.pdf. Online; accessed 29 January 2022.

[8] Oguz Kaya and Bora Uçar. 2015. Scalable sparse tensor decompositions in distributed memory systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[9] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 943–948.

[10] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.

[11] Jiajia Li, Yuchen Ma, and Richard Vuduc. 2018. ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs. http://parti-project.org Last updated: Jan 2020.

[12] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 238–252. https://doi.org/10.1109/SC.2018.00022

[13] Jiajia Li, Bora Uçar, Ümit V Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*. 227–237.

[14] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and Effective Sparse Tensor Reordering. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 227–237. https://doi.org/10.1145/3330345.3330366

[15] Marco Mondelli and Andrea Montanari. 2019. On the connection between learning two-layer neural networks and tensor decomposition. In *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 1051–1060.

[16] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An Efficient Mixed-Mode Representation of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 49, 25 pages. https://doi.org/10.1145/3295500.3356216

[17] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P. Sadayappan. 2019. Load-Balanced Sparse MTTKRP on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 123–133. https://doi.org/10.1109/IPDPS.2019.00023

[18] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos. 2017. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Transactions on Signal Processing* 65, 13 (2017), 3551–3582. https://doi.org/10.1109/TSP.2017.2690524

[19] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. http://frostt.io/

[20] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 61–70. https://doi.org/10.1109/IPDPS.2015.27

[21] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702. https://doi.org/10.1109/HPCA47549.2020.00062

[22] Fuxi Wen, Hing Cheung So, and Henk Wymeersch. 2020. 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, 4572–4576.

[23] Sasindu Wijeratne, Rajgopal Kannan, and Viktor Prasanna. 2021. Reconfigurable Low-latency Memory System for Sparse Matricized Tensor Times Khatri-Rao Product on FPGA. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC49654.2021.9622851

[24] Xilinx. 2022. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html. Online; accessed 29 January 2022.

[25] Xilinx. 2022. UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs. https://docs.xilinx.com/r/2021.2-English/ug949-vivado-design-methodology/. Online; accessed 1 August 2022.

[26] Xilinx. 2022. UltraScale Architecture-Based FPGAs Memory IP v1.4. https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf. Online; accessed 29 January 2022.

[27] Xilinx. 2022. Vivado Design Suite User Guide. https://docs.xilinx.com/v/u/2018.2-English/ug910-vivado-getting-started. Online; accessed 29 January 2022.