Tucker Decomposition Report

KAI HOGAN and OLIVER BOORSTEIN, University of Oregon, USA

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; *Shared memory algorithms*; • **Theory of computation** → *Design and analysis of algorithms.*

ACM Reference Format:
Kai Hogan and Oliver Boorstein. 2025. Tucker Decomposition Report. *ACM Trans. Graph.* 1, 1, Article 1 (October 2025), 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Abstract

This project implements and evaluates sparse Tucker decomposition on GPUs using both HIP and CUDA backends. We focus on two computationally intensive components of the decomposition pipeline: tensor–matrix multiplication (TMM) and core tensor generation. To support large sparse tensors efficiently, we adopt the BLCO storage format and develop GPU kernels that operate directly on this representation. The HIP implementation uses wavefront-level reductions and optional shared-memory tiles, while the CUDA implementation extends this design with a fully generalized N-dimensional core kernel capable of handling tensors up to sixteen modes without recompilation. Our tests found that CUDA consistently outperformed HIP in both TMM and core generation, often by large margins for high nonzero counts. CUDA's generalized core kernel achieves order-of-magnitude speedups over the CPU for 3D and 4D tensors, though performance declines for tensors with five or more modes due to for loop overhead and atomic contention. Our results highlight the strengths and limitations of sparse GPU-based Tucker decomposition and suggest several directions for improving performance, including alternative TMM output tensor representations, kernel fusion, and more sophisticated reduction strategies.

2 Introduction

2.1 Applications of Tucker decomposition

One of the main advantages of Tucker decomposition is its ability to significantly reduce storage costs by expressing a large tensor as a small core tensor together with factor matrices for each mode. This is especially valuable when working with extremely large datasets under tight memory constraints. For example, consider a 100×400×350×700 containing integer data. Storing it directly as an array would require about 9.8 gigabytes of memory. In contrast, a Tucker decomposition with ranks 8,13,12 and 17 would require only about 48,620 bytes to store the core tensor and the four factor matrices. The decomposed representation preserves the essential structure of the data while using only a fraction of the memory. Beyond memory savings, Tucker decomposition also acts as an effective noise filter. Many real-world datasets, such as MRI scans, contain variability that masks meaningful relationships among variables. Approximating the original tensor with a lower-rank Tucker model suppresses this noise and produces a representation that is easier to analyze. Tucker decomposition also provides insight into variation along each mode through its factor matrices. For instance, imagine a three-mode tensor of user data for a shopping app, where mode 1 represents age, mode 2 the average number of monthly purchases, and mode 3 gender. After decomposition, the factor matrix for mode 1 captures patterns related to age, the mode 2 matrix captures purchasing behavior, and the mode 3 matrix captures patterns associated with gender. Because of these benefits (memory efficiency, noise reduction, and interpretability), Tucker decomposition is widely used in chemical analysis, signal processing, and image processing.

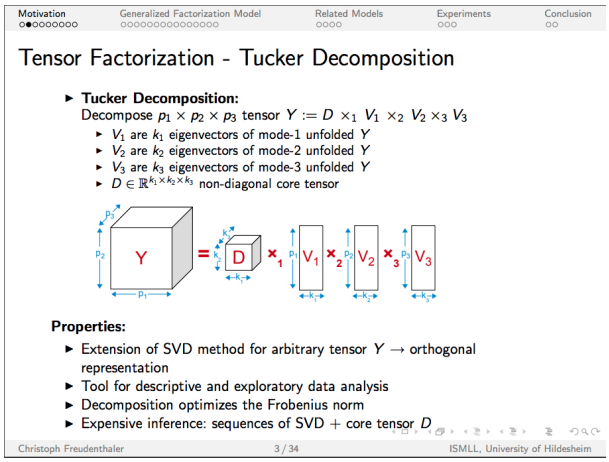

Fig. 1. Tucker Decomposition Depiction

2.2 ALTO and BLCO Tensor Storage

Sparse tensors can be challenging to store because their full set of possible entries often far exceeds the memory available on typical machines. For instance, the Amazon tensor in the FROSTT repository has dimensions 4,821,207 x 1,774,269 x 1,805,187 which means it has 1.544e+19 entries. Since most entries in a sparse tensor are zeros, and zero values do not need to be stored, large sparse tensors typically record only their nonzero elements. However, if a tensor contains a large number of nonzeros, even storing just those entries can become memory-intensive. Consider a four-dimensional tensor with 200 million nonzero entries stored in single-precision floating-point format. Each nonzero must store its value and its indices. A typical representation requires around 20 bytes per entry,

Authors' Contact Information: Kai Hogan, khogan@uoregon.edu; Oliver Boorstein, obo@uoregon.edu, University of Oregon, Eugene, Oregon, USA.

ACM 1557-7368/2025/10-ART1
https://doi.org/XXXXXXX.XXXXXXX

but with alignment padding, this can easily rise to 32 bytes. At that size, storing all 200 million nonzeros would require between 4 and 6.4 gigabytes of memory. This is where the ALTO and BLCO storage formats become useful. Both formats encode the coordinates of each nonzero as bit-interleaved integers, greatly reducing the overhead compared to storing separate integer indices for each mode. ALTO is designed for CPU-based processing, and interleaves coordinate bits so that the upper and lower subsets of each integer correspond to smaller regions within the overall tensor. BLCO, in contrast, is optimized for GPUs. It also encodes coordinates into a single integer, but stores them as contiguous bits rather than an interleaved pattern. BLCO additionally groups nonzeros into fixed-size blocks; within each block, the linearized coordinate gives the local position, and the block index identifies its location in the global tensor. Returning to the example tensor with 200 million nonzero single-precision entries, suppose its coordinates fit within 64 bits. In that case, ALTO and BLCO need only 8 bytes to store each entry's encoded coordinate. The value itself adds 4 bytes, and alignment brings the per-entry cost to about 12–16 bytes. The full tensor would therefore require roughly 2.4 to 3.2 gigabytes of memory substantially less than the simple non zero entry format. Due to these advantages in memory footprint and access efficiency, the BLCO format was selected for representing tensors on the GPU.

l		v
0	$(000000)_2$	1.0
4	$(000100)_2$	2.0
5	$(000101)_2$	4.0
10	$(001010)_2$	8.0
12	$(001100)_2$	6.0
15	$(001111)_2$	9.0
33	$(100001)_2$	5.0
48	$(110000)_2$	3.0
57	$(111001)_2$	10.0
61	$(111101)_2$	11.0
62	$(111110)_2$	7.0
63	$(111111)_2$	12.0

(a) Initial linearization.

b	l		v
0	0	$(00000)_2$	1.0
	16	$(10000)_2$	2.0
	17	$(10001)_2$	4.0
	6	$(00110)_2$	8.0
	18	$(10010)_2$	6.0
	23	$(10111)_2$	9.0
1	1	$(00001)_2$	5.0
	8	$(01000)_2$	3.0
	11	$(01011)_2$	10.0
	27	$(11011)_2$	11.0
	30	$(11110)_2$	7.0
	31	$(11111)_2$	12.0

(b) BLCO tensor.

Fig. 2. ALTO and BLCO formats (5)

3 HIP Implementation

3.1 Tensor Matrix Multiplication Algorithm

Tensor–matrix multiplication applies a matrix to a tensor along one of its modes, replacing that mode's dimension with the number of rows in the matrix. For example, if a tensor has dimensions I x J x K and you multiply it along mode 1 by a matrix of size M x N, the resulting tensor has dimensions M x J x K. In the context of Tucker decomposition, this step involves multiplying the tensor by a transposed factor matrix of size R x M, where R is the mode's decomposition rank, and M is the original length of the mode. Because R is typically much smaller than the original dimension, this operation contributes to compressing the tensor.

In the HIP implementation, the original tensor was stored in BLCO format, the matrix was stored as a flattened one-dimensional array, and the output tensor was also stored in a one-dimensional flattened layout. Each thread processed one nonzero entry, and for a multiplication along a given mode, each thread wrote to R output positions. For instance, if a thread handled an entry at coordinates (100, 300, 800) and the mode-1 decomposition rank was 10, that thread would produce outputs at (0, 300, 800), (1, 300, 800), ... (9, 300, 800).

Non-target indices refer to the coordinates in the modes not involved in the multiplication. For mode-1 multiplication, entries at (4, 7, 2) and (1, 7, 2) share the same non-target indices. Threads holding entries with identical non-target indices all write to the same region in the output tensor, which means atomic operations are needed to avoid race conditions. Atomics ensure correctness but reduce concurrency when many threads try to update the same location.

To mitigate this, shuffle operations were used to reduce the number of atomic updates. In HIP, a wavefront is a group of threads (typically 64) that execute in lockstep and can exchange data using shuffle instructions. These instructions operate on registers and are extremely fast. In the implementation, each thread generated a bitmask identifying other threads in its wavefront that shared the same non-target indices. The first thread in the wavefront matching that pattern became the group leader, and all other matching threads accumulated their contributions into it. This within-wavefront reduction increased overall concurrency by ensuring that fewer threads attempted to update the same output location, thereby reducing contention.

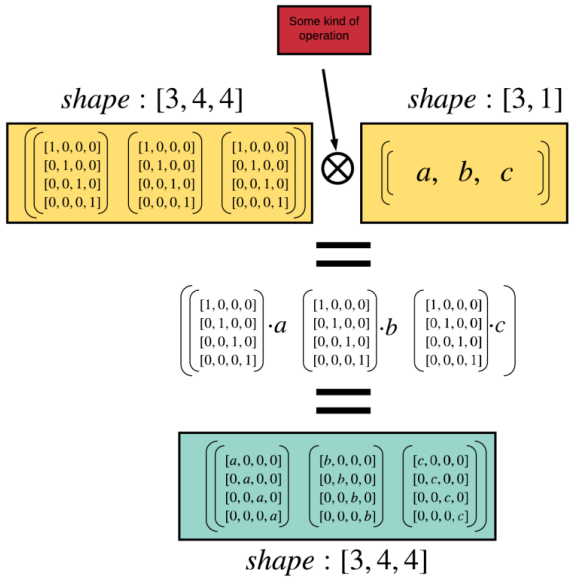

Fig. 3. Demonstration of Tensor Matrix Multiplication

3.2 Core Tensor Computation Algorithm

In Tucker decomposition, the core tensor is obtained by multiplying the original tensor by each of its transposed factor matrices. As a result, the core tensor has dimensions rank1×rank2...×rankn. Because our implementation used a single universal rank R for all modes, the resulting core tensor had dimensions R×R...×R.

In the HIP kernel, each thread began by loading its nonzero value and its coordinates from the BLCO representation. The thread then iterated through a nested loop over all modes, retrieving the appropriate entries from the factor matrix. Inside the innermost loop, the thread multiplied its value by the relevant factor matrix values to compute its contribution to the core tensor.

Since many threads contributed to a relatively small core tensor, atomic updates could create heavy contention. To reduce this, the core tensor was stored in shared memory whenever possible. If the full core tensor could not fit in shared memory, a partial core tensor containing only the region needed for that block was stored instead.

When a thread computed a contribution, it checked whether the corresponding output index fell within the shared-memory region. If it did, the thread performed an atomic update in shared memory; if not, it wrote directly to the global-memory core tensor. After all contributions were computed, the threads in the block cooperatively wrote the shared-memory core tensor back to the core tensor in global memory.

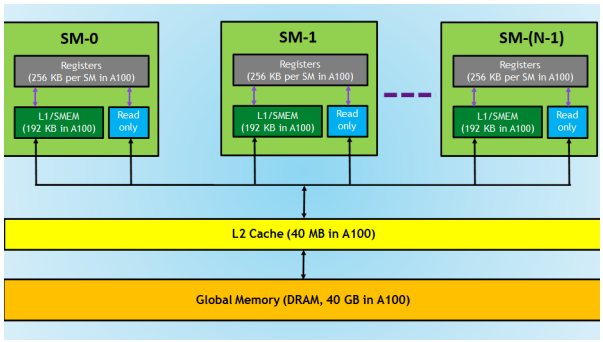

Fig. 4. GPU Memory Hierarchy

4 HIP Results

4.1 Test implementation

I built a testing pipeline that evaluates tensor–matrix multiplication (TMM) and core tensor computation for tensors of three through five dimensions. The process begins by generating a synthetic sparse tensor. I reused a function from an older project that produces a random collection of nonzero entries; given the desired dimensions and a target nonzero count, it returns a list of coordinate–value tuples. These entries are then converted into my BLCO tensor format.

Once the BLCO tensor is constructed, it is passed through the GPU implementations. First, I run TMM along every mode and time, both the GPU kernel itself and the full host-side operation. I then run the core tensor computation using the same timing approach. After collecting all GPU timings, I compute the same operations on the CPU to obtain a reference result.

For correctness checking, integer tensors are compared by verifying that every output entry matches exactly. For floating-point tensors, I compute the average difference between the CPU and GPU results to estimate numerical error.

The entire procedure is driven by a test harness. After compiling test-suite with the provided Makefile, tests can be run with: ./test-suite <NNZ> <Decomposition rank> <Block Size> <Type> (three to five different dimensions).

*Note: the shared-memory reduction used in core tensor computation was only completed near the end of development. As a result, only the final two five-dimensional tests use the full shared-memory version of the core algorithm; all earlier tests rely on the earlier, slower reduction path.

4.2 3D kernels

4.2.1 100 x 100 x 100 tensor with 5000 non zeros and decomposition rank 15: For tensor–matrix multiplication, all kernel configurations showed similar performance across different block sizes. Execution times ranged from 0.07 to 0.1 milliseconds, with the 320-thread block size performing slightly better than the others across all modes. However, the kernels themselves accounted for only about 20 percent of the total time spent on tensor matrix multiplication; most of the time was consumed by GPU memory allocation.

Core generation required significantly more time than tensor–matrix multiplication, taking between 8 and 11 milliseconds. This is expected, since the core tensor is relatively small (3,375 entries) and every thread contributes to it using atomic operations, which introduces substantial contention and reduces concurrency.

Floating-point error remained small for tensor matrix multiplication (2e-07), but was noticeably larger for the core computation (approximately 0.025).

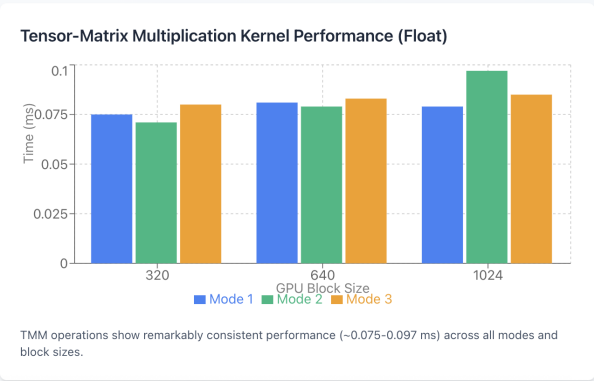

Fig. 5. Tensor Matrix Multiplication Performance for Floating Point Operations

4.2.2 2500 x 2500 x 2500 tensor with 50000 non zeros and decomposition rank 15: Both the tensor–matrix multiplication kernels and the core-generation kernels scaled reasonably well when the number of nonzeros increased to 50,000. The tensor–matrix multiplication kernels ran in approximately 0.11 to 0.18 milliseconds, which is only about an 80 percent slowdown compared to the case with 5,000

nonzeros, which is a tenfold increase in input size. Core-generation time improved only slightly, with roughly an eight percent speedup.

However, the memory-allocation time for tensor–matrix multiplication increased dramatically, reaching around 110 milliseconds. This is expected: the intermediate tensor produced by the multiplication grew from 100 x 100 x 15 to 2500 x 2500 x 15, which required much larger GPU allocations. In contrast, the total time for core generation changed very little because the core tensor remained the same size (15 x 15 x 15) regardless of the number of nonzeros.

Floating-point error for tensor–matrix multiplication decreased slightly, remaining very small (between 2e-08 - 2e10). The floating-point error for core computation, however, increased significantly, rising to the range of 1.0 to 1.5.

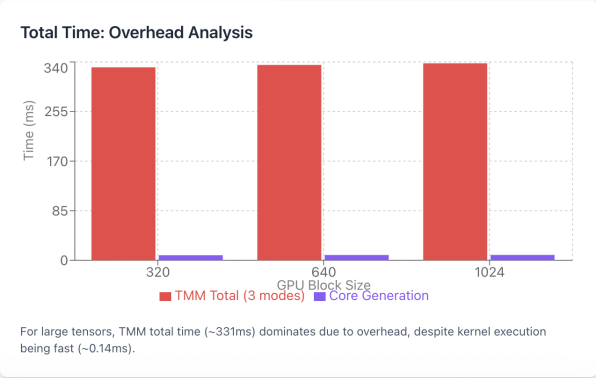

Fig. 6. Total Overhead for TMM and Core Generation

4.2.3 10000 x 10000 x 10000 tensor with 500000 non zeros and decomposition rank 15: The tensor matrix multiplication kernel scaled well in terms of times taking in between .21 and .42 milliseconds, however the total execution time skyrocketed to between 1.7 seconds and 3.1 seconds. This is likely because the product of tensor matrix multiplication was an array with 1.5 billion entries and likely hit memory limits. The core generation time increased to around 35 seconds for all iterations however the floating point accuracy for core generation completely broke resulting in an average discrepancy of up to 452.

4.3 4D kernels

4.3.1 50 x 50 x 50 x 50 tensor with 5000 non zeros and decomposition rank 15: In terms of TMM kernel speed the kernels still performed relatively well taking around .08 milliseconds to 0.1 milliseconds for each mode. The amount of time it took to generate the core tensor was much longer than the 3D version with the same number of non zeros taking in between 120 and 150 milliseconds to execute the operation. Because of the extra mode the 4D core tensor is 15 times as large as the 3D tensor thus increasing computation time significantly. Surprisingly the core generation error was less than it was for the 3D version with the same number of non zeros generating around half as much of an error (0.0127).

4.3.2 Testing operations on 250 x 250 x 250 x 250 tensor with 50000 non zeros with decomposition rank 15: The TMM kernel slowed down

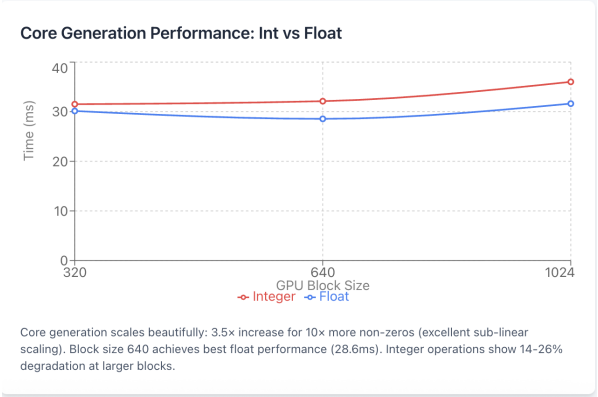

Fig. 7. Core Generation Performance Ints vs Floats

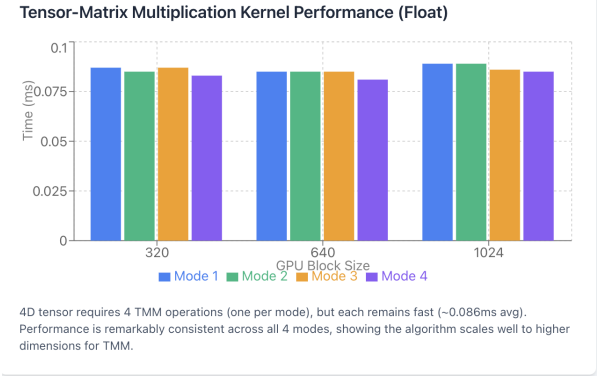

Fig. 8. Core Generation Time

even further as the overhead of output-tensor storage and allocation grew. The kernel itself still ran quickly, around 0.15 milliseconds, but the total execution time ballooned to roughly 300 milliseconds. In contrast, core generation scaled almost perfectly: the 4D core took only about 3 milliseconds longer than the core generated from 5000 nonzeros. The block size of 640 also emerged as the strongest overall performer, producing the lowest floating-point error, the fastest integer core-generation time, and the most efficient TMM overhead.

4.3.3 500 x 500 x 500 x 500 tensor with 500000 non zeros and decomposition rank 15: Although TMM kernel time increased only slightly, rising to between 0.17 and 0.25 milliseconds, the overall runtime jumped dramatically to roughly 1,800–3,000 milliseconds. This sharp increase suggests thrashing or some other memory-allocation-related failure. Core-generation time also spiked, taking between 500 and 1,200 milliseconds, and the resulting numerical error exceeded 350, making the output effectively unusable. As for block sizes, 1024 delivered the best performance for core generation, while 640 remained the most effective for TMM. Block size 320 performed the worst by a wide margin across both operations.

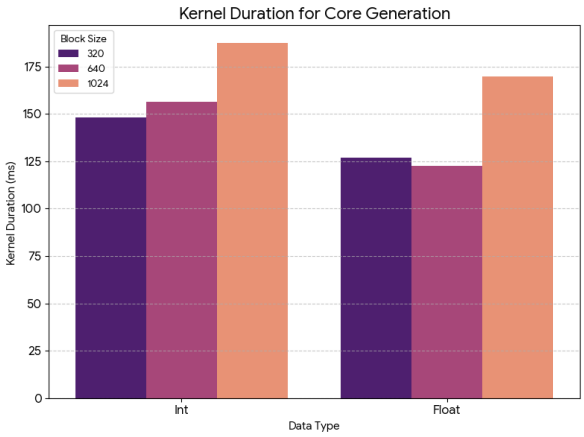

Fig. 9. Core Generation Time

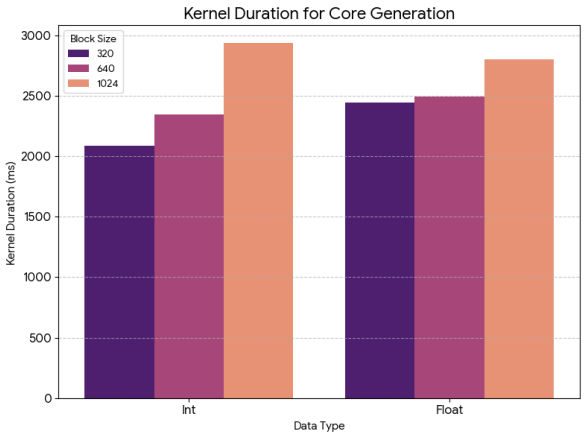

Fig. 11. Core Generation Kernel Times

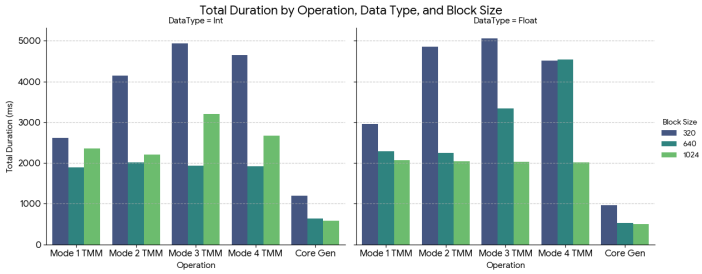

Fig. 10. Operation Times

for smaller block sizes and integer types, but loses efficiency for other data types.

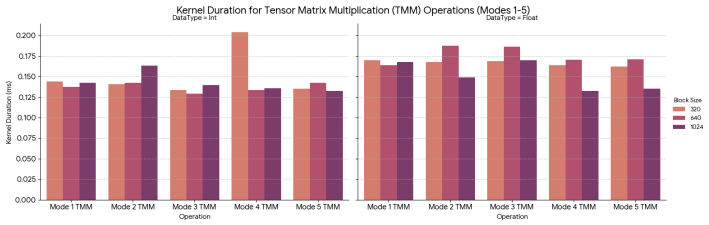

Fig. 12. Overall Times

4.4 5D kernels

4.4.1 Testing operations on 50 x 50 x 50 x 50 x 50 tensor with 5000 non zeros with decomposition rank 15: For tensor matrix multiplication, kernel time remained steady at roughly 0.10–0.15 milliseconds, but the total duration lagged at 100–130 milliseconds. This gap makes sense given that the TMM output contained around 93 million entries, making allocation and data movement the dominant cost. Core generation also performed reasonably, finishing in about 2500 milliseconds. Floating-point error for TMM was negligible, falling in the range of 2e-12 to 2e-13, while core-generation error was more noticeable at around 0.006.

*4.4.2 *Note:* For the next two tests the CPU version was not able to perform the operations, therefore the next two summaries will not include floating point errors.

4.4.3 Testing operations on 100 x 100 x 100 x 100 x 100 tensor with 50000 non zeros with decomposition rank 15: Kernel execution time stayed steady at 50,000 nonzeros, but the total runtime shot up to about 3300 milliseconds. Core generation showed much more variability, ranging from 2000 to 6500 milliseconds. With block sizes of 320 and 640 on integer data, core generation stayed near the low end, around 2000 and 2400 milliseconds. Every other configuration performed worse, drifting toward the 6500-millisecond range. This pattern suggests that the shared-memory reduction performs well

4.4.4 Testing operations on 100 x 100 x 100 x 100 x 100 tensor with 500000 non zeros with decomposition rank 15: For tensor–matrix multiplication, the total runtime unexpectedly dropped to around 1500–1800 milliseconds. This is an anomalous result that hints at something odd in memory allocation or a possible mistake in how timings were recorded. In contrast, core generation time ballooned to roughly 30,000 milliseconds. That spike suggests the shared-memory reduction becomes far less effective as the tensor size increases.

5 CUDA Implementation

The CUDA implementation mirrors the structure of the HIP code but removes all hard-coded assumptions about tensor order. Instead, it uses a single, fully dynamic kernel that can compute the core tensor for any tensor dimensionality up to a configurable maximum. The goal was not to redesign the decomposition algorithm, but to factor out 3D-specific logic so that the same code path cleanly supports 3D through 16D tensors.

5.1 Tensor–Matrix Multiplication and Core Generation

The CUDA versions of tensor–matrix multiplication (TMM) and the original 3D core kernel are direct ports of the HIP implementations described earlier. The overall data flow is unchanged:

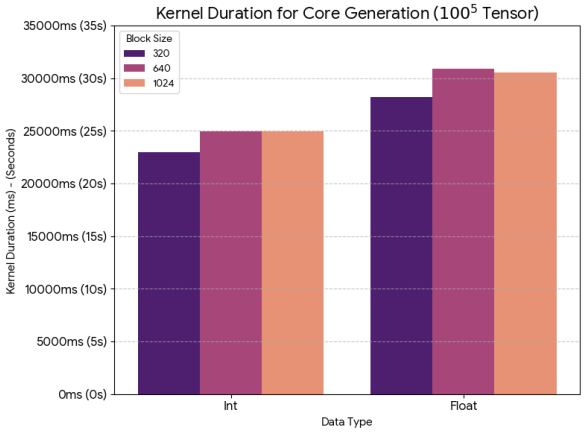

Fig. 13. Core Generation Times

- The input tensor is stored in BLCO format and copied to the GPU.
- Each TMM kernel launches one thread per nonzero entry; that thread reads the BLCO coordinate and value, gathers the appropriate factor entries, and writes its contributions into a dense output tensor.
- The core kernel takes the dense TMM outputs and factor matrices and accumulates contributions into a dense core tensor using atomic updates.

The main differences are purely backend–specific:

- HIP wavefront intrinsics are replaced with CUDA warp intrinsics. Where the HIP implementation used 64-thread wavefronts, the CUDA version uses 32-thread warps, but the reduction pattern (mask construction, leader election, and intra-warp accumulation) is identical.
- HIP runtime calls (for streams, events, and memory management) are replaced with their CUDA counterparts. The host timing infrastructure is the same: each call can optionally record upload, kernel, and download durations using CUDA events.

Because the CUDA TMM and fixed-dimension core kernels behave almost identically to their HIP counterparts, most of the interesting design work centers on the generalized N-dimensional core path described below.

5.2 Generalized N-Dimensional Core Generation

To support core computation for tensors with arbitrary order, the CUDA implementation introduces a new path built around two components:

(1) a fully dynamic device kernel, tucker_core_kernel_nd_sparse, and
(2) a host launcher, tucker_compute_core_nd_cuda, that prepares all metadata at runtime.

Together, these components eliminate the need for separate 3D, 4D, or 5D kernels. Instead, the tensor order appears only as a loop bound inside the kernel and in small fixed-size thread-local arrays, so the same binary handles every dimensionality up to the configured limit (Tested up to 16D).

5.2.1 Device Kernel. The tucker_core_kernel_nd_sparse kernel assigns one BLCO nonzero (NNZ) to each thread. Its responsibilities are:

Coordinate extraction. All per-mode coordinates are recovered from the BLCO encoding on the device. Each thread repeatedly calls extract_mode_nd inside a loop over the number of modes, unpacking the bit-interleaved (or bit-packed) coordinate into a small fixed-size array:

- coords[kMaxTensorModes] holds the tensor index along each mode for the current NNZ.
- The NNZ value itself is cached in a register (thread_val) to avoid rereads from global memory.

Because the number of modes never exceeds kMaxTensorModes, this array remains stack-allocated and does not depend on template parameters.

Iterating over rank combinations. To compute the contribution of one NNZ to the core tensor, the kernel has to consider every combination of factor indices across all modes. Conceptually, this is a nested loop over R^N combinations, where R is the (uniform) rank and N is the tensor order. Instead of hard-coding nested loops, each thread maintains a small integer counter:

- rank_coords[kMaxTensorModes] represents the current combination of rank indices.
- A manual "increment with carry" step advances this counter in prefix-product order, exactly like incrementing a multi-digit number in base R.
- This increment logic is implemented as a simple loop over the number of modes, so it naturally scales from 3D to 16D without templates or code generation.

For each setting of rank_coords, the thread:

(1) Looks up one factor entry per mode from an array of device pointers:
$f_m = factors[m][rank\_coords[m] * dim\_m + coords[m]]$,
where dim_m is the length of mode m.
(2) Multiplies all factor entries together with thread_val to form the contribution for that core index.
(3) Linearizes the rank coordinates into a single core index using prefix-product strides (row-major order over the rank modes).
(4) Atomically accumulates this contribution into the dense core buffer in global memory.

All accumulation occurs in global memory in the ND kernel. Unlike the HIP shared-memory core path, no per-block shared core tile is allocated, so the required shared memory does not grow with rank or tensor order. Only the small thread-local arrays (coords and rank_coords) scale with the maximum number of modes, which is bounded by kMaxTensorModes.

5.2.2 Host Launcher. The tucker_compute_core_nd_cuda function wraps the dynamic kernel in a reusable interface that matches the rest of the project's tensor abstractions. Its responsibilities are:

Metadata preparation. On the host, the launcher:

- Validates that the tensor order does not exceed kMaxTensorModes and that the rank is within a safe range.
- Retrieves BLCO blocks, per-mode dimensions, bit widths, and bitmasks from the existing tensor container (get_blco, get_dims, get_bitmasks).
- Copies this metadata to device memory so the kernel can decode coordinates for each NNZ without any host assistance.

Factor matrix upload. Each factor matrix is flattened on the host and copied to the GPU. Rather than passing a separate pointer parameter for each mode, the launcher:

- allocates device memory for each factor matrix,
- stores the resulting device pointers in a host array, and
- copies that pointer array into device memory so the kernel can index factors[mode_idx] dynamically.

This pointer-array indirection is what allows a single kernel to handle any number of modes without recompilation.

Core allocation and timing. The launcher computes the core size as R^N using a safe integer power helper and allocates a dense device buffer of that length. It zero-fills the buffer, optionally records an "upload" timestamp, and then launches tucker_core_kernel_nd_sparse with a simple one-dimensional grid:

- The grid is sized by NNZ count; each thread processes exactly one nonzero.
- No specialization of block size or grid shape is required for different N, since the kernel's inner loops handle the dimensionality.

After the kernel completes, the launcher:

- records "kernel" and "download" times using CUDA events,
- copies the dense core tensor back to host memory, and
- frees all temporary device allocations (BLCO buffers, metadata arrays, factor matrices, and the device pointer array).

The caller can either ignore timings or capture them directly from the launcher's output fields without parsing standard output.

5.2.3 Supporting Infrastructure and Design Trade-offs. Several helper routines support the ND core path:

- Utilities to compute per-mode bit widths and masks for BLCO coordinates and to build row-major strides for the dense core.
- Functions that copy factor matrices and their pointer array to the device, encapsulating all CUDA allocation and error checking.
- A configurable maximum tensor order (kMaxTensorModes) that allows the kernel to rely on fixed-size thread-local arrays while still supporting orders well beyond the 3D case.
- Explicit template instantiations at the end of the CUDA translation unit so that all required (value type, index type, mask width) combinations are compiled once and reused throughout the project.

This design deliberately favors generality and simplicity over aggressive low-level optimization: by moving all dimensionality dependence into loops and small fixed arrays, the same kernel can be reused for every experiment in our CUDA results section. At the same time, the interface remains compatible with the existing BLCO tensor abstraction, so HIP and CUDA backends can share the same storage layer and differ only in how they traverse nonzeros and accumulate into the core.

6 CUDA Results

6.1 Experimental Setup

To evaluate the generalized CUDA core kernel, we generated synthetic tensors whose total logical size remained constant while the number of modes varied from three to seven. For a fixed logical size of approximately one million entries, we used the following dimension patterns:

- 3D: 100000×10×10
- 4D: 10000×10×10×10
- 5D: 1000×10×10×10×10
- 6D: 100×10×10×10×10×10
- 7D: 10×10×10×10×10×10×10

For each dimensionality we sampled three sparsity regimes with 5,000, 500,000, and 5,000,000 nonzeros, and used a uniform Tucker rank R=10 R=10 in every mode. All experiments used the BLCO representation on the GPU and the same factor matrices on both CPU and GPU. For each configuration we measured:

- average tensor–matrix multiplication (TMM) time per mode on the CPU and GPU;
- core tensor computation time on the CPU;
- GPU upload, kernel, and download times for the core computation.

Unless otherwise noted, speedup is reported as the ratio

$$speedup = \frac{CPU\ core\ time}{GPU\ time},$$

so values greater than one indicate a faster GPU implementation.

6.2 Core Kernel Speedup Across Dimensions

Figure 14 shows the speedup of the CUDA core kernel relative to the CPU implementation, using CPU core time divided by GPU kernel time. Each group of three bars corresponds to one tensor order, and the three bars within a group represent the different nonzero counts.

Across all sparsity levels, dimensionality is the dominant factor:

- **3D tensors.** For 3D tensors the GPU core kernel is between 10× and 18× faster than the CPU. With 5,000 nonzeros the speedup is roughly 10.2×; at 500,000 and 5,000,000 nonzeros the speedups increase to about 17.6× and 17.9×, respectively.
- **4D tensors.** The trend continues for 4D tensors, where the GPU core kernel achieves 14.7× to 15.3× speedup over the CPU, largely independent of nonzero count. This indicates that once the core tensor remains reasonably small (here 104 entries), the GPU is able to amortize atomic contention and exploit parallelism effectively.
- **5D tensors.** At five modes the situation reverses. Speedup drops below one, ranging from approximately 0.61× to 0.75× depending on sparsity, meaning the CPU is actually faster than the GPU kernel. The core now has 105 entries, and the cost of looping over all rank combinations and performing atomics in global memory begins to dominate.

- **6D tensors.** For 6D tensors, speedup remains below one for all sparsity levels, between 0.47× and 0.66×. The GPU still exposes massive parallelism over nonzeros, but the per-thread work over 106 core entries and the increased atomic contention make the kernel more expensive than the simpler CPU loop nest.
- **7D tensors.** At seven modes the GPU and CPU are effectively at parity. The speedup values are near one for 5,000 nonzeros and slightly below one (about 0.86×) for the two larger nonzero counts. At this point the core tensor contains 107 entries, and the dynamic ND kernel is dominated by iterating over all rank combinations rather than by raw memory bandwidth.

Overall, the results highlight a clear regime split: for low-order tensors (three and four modes), the generalized CUDA kernel substantially outperforms the CPU, but for higher-order tensors (five or more modes) the cost of enumerating all rank combinations and performing global-memory atomics outweighs the benefits of GPU parallelism.

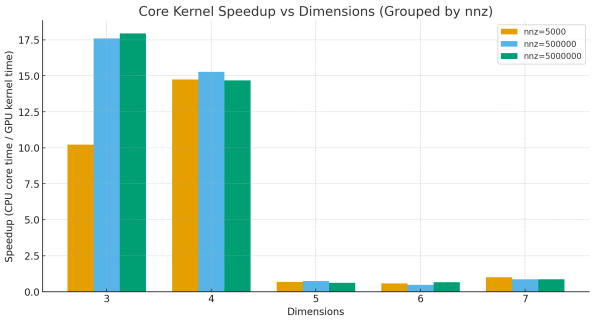

Fig. 14. CUDA core generation speedup (CPU core time divided by GPU kernel time) as a function of tensor order and nonzero count.

6.3 End-to-End Core Time Versus Kernel Time

The previous subsection focuses on kernel-only speedup, which isolates the compute portion of the algorithm. For practical workloads, however, data movement cannot be ignored. When we incorporate upload and download time and compare CPU core time to full GPU wall time, the qualitative picture remains similar but the speedups shrink slightly:

- For 3D and 4D tensors, the GPU still provides substantial acceleration, with end-to-end speedups in the range of roughly 7× to 16×, depending on sparsity.
- For 5D tensors, the GPU is consistently slower than the CPU once transfer overhead is included, with speedups between about 0.60× and 0.74×.
- For 6D and 7D tensors, GPU and CPU times are comparable; in some cases the GPU is slightly slower, reflecting the fact that the kernel is dominated by global-memory atomics and rank-loop overhead.

These trends suggest that the generalized ND kernel is most effective when the core is relatively small and fits naturally into the GPU cache hierarchy. As the tensor order grows, the rank loops become the main bottleneck, and a more sophisticated blocking strategy (for example, tiling the core into shared memory or splitting the rank loops across threads) would be necessary to regain GPU advantage.

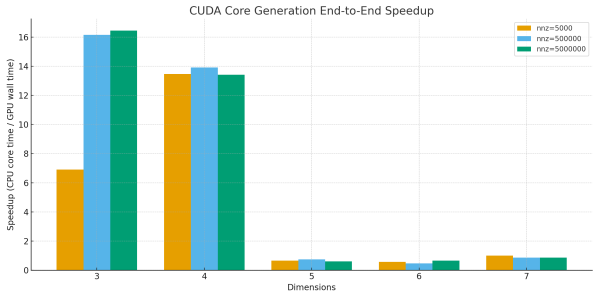

Fig. 15. End-to-end CUDA core generation speedup (CPU core time divided by GPU wall time), separated by dimensionality and nonzero count.

6.4 Tensor–Matrix Multiplication Performance

In contrast to the core computation, tensor–matrix multiplication showed no regime in which the GPU outperformed the CPU. Across all tested dimensionalities and nonzero counts, the average GPU TMM time per mode was between 1.4 and 2.1 times slower than the CPU implementation. This behavior is consistent with the design of the experiment:

- Each TMM kernel performs relatively little arithmetic per nonzero and writes into a large dense output tensor, making the operation strongly memory bound.
- The BLCO representation does not provide any reuse of factor entries or output locations within a single kernel launch, so there is limited opportunity to amortize memory latency.
- On the host side, TMM incurs significant overhead from allocating and zeroing large output arrays; this cost does not disappear on the GPU and often dominates the overall runtime.

These results suggest that the main opportunity for GPU acceleration in our current pipeline lies in the core computation rather than in TMM itself. Improving TMM performance would likely require a different output representation (for example, writing directly into a BLCO tensor or a semi-sparse structure) and more aggressive fusion of multiple TMM stages to reduce allocation overhead.

6.5 Effect of Thread Block Size on Core Generation Performance

To better understand how kernel configuration influences performance, we conducted an additional experiment sweeping over six thread block sizes: 16, 32, 64, 256, 512, and 1024. For each configuration, we evaluated tensor–matrix multiplication and core tensor generation on the same 3D tensor used in previous tests, running the experiment across three nonzero counts (5,000, 500,000, 5000,000).

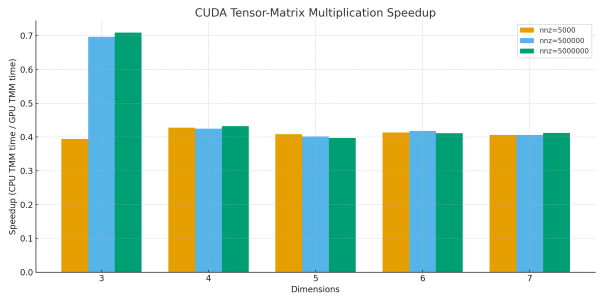

Fig. 16. Tensor–matrix multiplication speedup (CPU time divided by GPU time) across dimensionalities and nonzero counts.

The goal of this sweep was to characterize how thread grouping affects both kernel execution time and overall wall-clock performance for the most compute-intensive phase of Tucker decomposition.

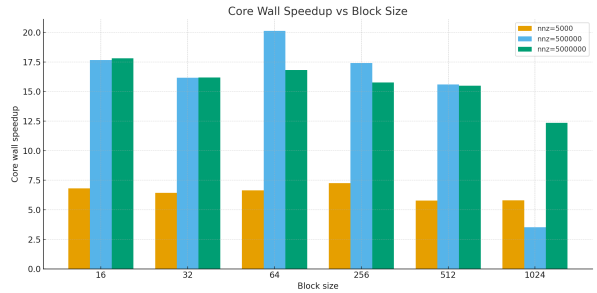

Fig. 17. Core generation wall-clock speedup (CPU vs GPU) across different thread block sizes and nonzero counts.

The results reveal several consistent trends. First, core tensor generation shows substantial GPU speedup over the CPU across all configurations, but the magnitude of this speedup depends strongly on block size. Figures 17 and 18 summarize the wall-time and raw kernel-time speedups, respectively. Each figure displays three bar groups per block size corresponding to the three tested nonzero counts.

For core wall-time speedup, block sizes of 64 and 256 provide the highest acceleration for medium and large tensors, achieving speedups of approximately 20x for 500,000 nonzeros and around 17x for 5,000,000 nonzeros. In contrast, very large blocks (1024 threads) consistently underperform, with the wall-time speedup dropping sharply—most notably for the 500,000-NNZ case, where performance collapses due to increased contention and diminished scheduling flexibility. These observations suggest that overly large thread blocks limit effective parallelism for the BLCO-based computation pattern, likely by reducing active warps per SM and increasing atomic-update interference.

Kernel-only speedups exhibit similar behavior. Blocks of size 64 achieve the strongest performance, reaching kernel-level speedups above 22x for the 500,000-NNZ case, while sizes 16, 32, and 256 follow closely behind. Again, block size 1024 performs the worst across all NNZ levels, reducing speedups to the 13–14x range. The close correspondence between kernel and wall-time trends indicates that most configuration-dependent behavior originates inside the GPU kernel rather than in upload or download overhead.

Overall, these experiments show that moderate block sizes (64–256 threads) deliver the best performance for core tensor generation, balancing warp occupancy, atomic-write contention, and scheduling granularity. Both very small and very large blocks lead to reduced acceleration, though for different reasons: smaller blocks underutilize available compute resources, while larger blocks introduce warp serialization and higher atomic pressure.

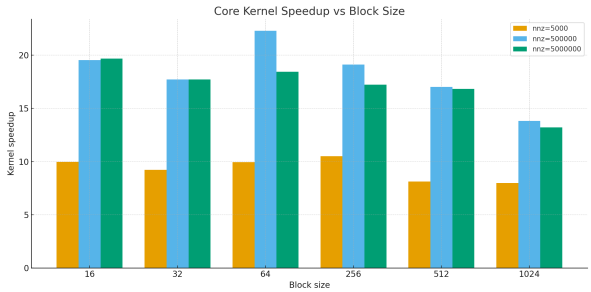

Fig. 18. Kernel-only speedup for core generation across thread block sizes. Moderate block sizes (64–256) consistently yield the highest acceleration.

6.6 Summary

The CUDA experiments demonstrate that the generalized ND core kernel can deliver significant performance gains for low-order tensors while remaining correct and stable up to seven modes. For 3D and 4D tensors, the GPU provides more than an order of magnitude speedup in core computation even when data-transfer overhead is included. For tensors with five or more modes, however, the current ND design becomes dominated by rank-loop work and atomic contention, and the CPU implementation becomes competitive or superior. These observations point directly to future work: introducing shared-memory tiling over rank combinations, exploring cooperative-group reductions for the core tensor, and redesigning TMM output storage so that the GPU spends less time moving dense intermediate tensors and more time performing useful computation.

7 HIP vs CUDA

7.1 Tensor Matrix Multiplication Times

CUDA generally delivered much faster TMM times than HIP. For 3D TMM with 5,000 nonzeros, CUDA took 2.56 milliseconds, while HIP completed the operation in 0.449 milliseconds. But at 500,000 nonzeros, the trend reversed sharply: CUDA finished in 6.42 milliseconds, whereas HIP required 1800 milliseconds. This difference is most likely due to the fact that the for 500,000 non zero entries HIP kernel took in a tensor with 1 billion non-zero entries, whereas the CUDA kernel took in a tensor with 10 million entries; thus, the output tensor for the CUDA kernel was much smaller than the output tensor for the HIP kernel in most cases. This large gap was likely due to inefficient memory allocation in the HIP implementation. The same pattern appeared in the 4D and 5D cases—HIP was slightly faster

at 5,000 nonzeros, but CUDA was more than 100× faster at 500,000 nonzeros. However, these discrepancies are also probably due to the fact that the HIP output tensor was much larger than the CUDA output tensor for both of the 4D and 5D 500,000 non-zero test cases.

7.2 Core Tensor Generation Times

Core tensor generation showed a similar disparity. For 3D tensors with 5,000 nonzeros, CUDA produced the core in about 4 milliseconds, while HIP took around 33 milliseconds. At 500,000 nonzeros, the difference grew even larger: HIP's peak time was about 480 milliseconds, compared to CUDA's roughly 18 milliseconds. This trend held for 4D and 5D tensors as well, with CUDA consistently outperforming HIP at both the small and large nonzero counts.

8 Future Directions

8.1 Project Limitations:

Because the computations we implemented were both complex and time-intensive, most of our effort went into coding rather than tuning and testing. As a result, we were only able to run a small number of tests, and those tests varied only a limited set of parameters. There are still many unexplored directions and parameters that could be investigated in future work.

8.2 Different Tensor–Matrix Multiplication Output Storage:

We stored the output of tensor–matrix multiplication as a dense array. Since the output tensor can become extremely large, allocating memory for this array on both the GPU and CPU and transferring data between them quickly became a major bottleneck. This was reflected in the fact that the raw HIP kernels were sometimes over 1000× faster than the full tensor–matrix multiplication function. A more efficient output representation is therefore essential. One option is to store the result as a BLCO tensor, though this would require determining how to compute BLCO coordinates on the GPU. Another promising direction is the development of an intermediate representation for semi-sparse tensors, since tensors often transition from sparse to dense during Tucker decomposition.

8.3 Multiple Rounds of Tensor Matrix Multiplication in a Kernel:

Because allocating output tensors on the GPU is so costly, reducing the number of allocations would likely improve performance significantly. Possible approaches include fusing multiple stages of tensor–matrix multiplication into a single kernel, preallocating all necessary output buffers, or streaming portions of the output tensor onto the GPU in stages. Each method aims to reduce the severe bottleneck created by repeated memory allocation.

8.4 Cooperative Groups:

We have already implemented wavefront-level reductions, but cooperative groups would allow reductions across larger groups of threads. This could further decrease the number of atomic operations. For instance, all threads in a block that contribute to the same output index could first reduce their values to a single thread, which would then perform the atomic write to global memory. This strategy could potentially reduce the number of atomic operations by an order of magnitude. Comparing cooperative-group reductions with our existing wavefront reductions would be a natural next step.

8.5 Test Against cuTENSOR

We would like to have tested against existing GPU implementation libraries such as cuTENSOR, which would allow us to compose decompositions using extremely optimized tensor operations. This would tell us how far off the mark for optimization we were for designing this from scratch.

9 Conclusion

This work demonstrates both the promise and the difficulty of performing sparse Tucker decomposition on GPUs. By combining the BLCO storage format with customized kernels in HIP and CUDA, we were able to evaluate performance across a wide range of tensor orders and sparsity levels. The results show a clear advantage for CUDA which achieves substantially faster tensor–matrix multiplication and dramatically lower core-generation times, especially at large nonzero counts where HIP's output times skyrocket due to memory allocation overhead. The generalized CUDA core kernel also proved effective for low-order tensors, providing strong speedups relative to the CPU while supporting a large range of dimensions through a single dynamic code implementation.

At the same time, the experiments reveal several constraints for the current design. For higher-order tensors, the cost of iterating over all rank combinations and the heavy use of global-memory atomics reduce concurrency to the point where the CPU outperforms the GPU. Tensor–matrix multiplication also remains bottlenecked by dense output allocation which was at times more than a gigabyte, making it a poor candidate for GPU acceleration in its current form. These findings point toward clear paths for improvement: storing TMM outputs in a more compact sparse or semi-sparse format which takes up less memory, reducing repeated allocations, using cooperative-group reductions, and evaluating performance relative to highly optimized libraries such as cuTENSOR. In the future these optimizations may make sparse tucker decomposition more suitable for the GPU.

References

(1) Nguyen, Andy, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi. "Efficient, Out-of-Memory Sparse MTTKRP on Massively Parallel Architectures." arXiv, 27 Jun. 2022, arXiv:2201.12523.

(2) Helal, Ahmed E., Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. "ALTO: Adaptive Linearized Storage of Sparse Tensors." arXiv, 27 Apr. 2021, arXiv:2102.10245.

(3) Kolda, T. G. & Bader, B. W., "Tensor Decompositions and Applications," SIAM Review, vol. 51, no. 3, pp. 455-500, 2009. doi:10.1137/07070111X.