Exploring Data Layout for Sparse Tensor Times Dense Matrix on GPUs

KHALID AHMAD, University of Utah, USA
CRIS CECKA and MICHAEL GARLAND, NVIDIA Corporation, USA
MARY HALL, University of Utah, USA

An important sparse tensor computation is sparse-tensor-dense-matrix multiplication (SpTM), which is used in tensor decomposition and applications. SpTM is a multi-dimensional analog to sparse-matrix-dense-matrix multiplication (SpMM). In this article, we employ a hierarchical tensor data layout that can unfold a multi-dimensional tensor to derive a 2D matrix, making it possible to compute SpTM using SpMM kernel implementations for GPUs. We compare two SpMM implementations to the state-of-the-art PASTA sparse tensor contraction implementation using: (1) SpMM with hierarchical tensor data layout; and, (2) unfolding followed by an invocation of cuSPARSE's SpMM. Results show that SpMM can outperform PASTA 70.9% of the time, but none of the three approaches is best overall. Therefore, we use a decision tree classifier to identify the best performing sparse tensor contraction kernel based on precomputed properties of the sparse tensor.

CCS Concepts: • **Mathematics of computing** → **Computations on matrices**; *Mathematical software performance*; • **Computing methodologies** → *Shared memory algorithms;*

Additional Key Words and Phrases: Sparse tensors, SpMM, data layout

ACM Reference Format:
Khalid Ahmad, Cris Cecka, Michael Garland, and Mary Hall. 2024. Exploring Data Layout for Sparse Tensor Times Dense Matrix on GPUs. *ACM Trans. Arch. Code Optim.* 21, 1, Article 20 (February 2024), 20 pages. https://doi.org/10.1145/3633462

1 INTRODUCTION

Sparse tensor computations include important applications in data analysis, data mining, health care, natural language processing, machine learning, and social network analytics [15]. Sparse tensors are represented by multidimensional arrays. A vector is a 1-way tensor, and a matrix is a 2-way tensor (rows and columns), as shown in Figure 1. A tensor is sparse if most of its entries are zero. Similar to sparse matrices, sparse tensors are usually stored in a compressed representation that does not store zero values; instead, we use auxiliary data structures to map nonzero values to their logical locations in the tensor; e.g., **coordinate** (**COO**) and **compressed sparse fiber** (**CSF**) [28].

To achieve high performance on sparse tensor computations requires careful attention to expensive memory latency arising from indirection through the compressed representation. The *layout*

Authors' addresses: K. Ahmad (Corresponding author) and M. Hall, University of Utah, Salt Lake City, UT 84108; e-mails: {Khalid, m.hall}@cs.utah.edu; C. Cecka and M. Garland, NVIDIA Corporation, Santa Clara, California 95051; e-mails: {ccecka, mgarland}@NVIDIA.com.

ACM 1544-3566/2024/02-ART20
https://doi.org/10.1145/3633462

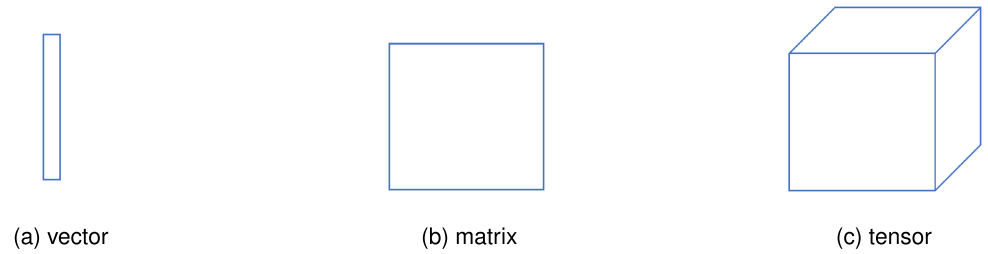


Fig. 1. Multidimensional array (tensors).

of a sparse tensor in memory—what is typically called a format when referring to sparse matrices—significantly impacts this latency. Different sparse layouts tradeoff performance, storage footprint, and nonzero values distribution of a given input tensor [16]. Optimizing data layout and reorganization of nonzero values can lead to reduced latency and significant performance improvements.

An important sparse tensor computation is **sparse-tensor-dense-matrix** (**SpTM**) multiplication, which is used in tensor decomposition and applications [15]. Shi et al. [26] make the connection that most tensor contractions are equivalent to dense matrix multiply from the **basic linear algebra subprogram** (**BLAS**) library which is a highly optimized and widely used routine in scientific computing. Similarly, Li et al. [18] observe that SpTM is a multi-dimensional analog to **sparse-matrix-dense-matrix** (**SpMM**). SpMM is the multiplication of a sparse m-by-n matrix A by a tall and narrow dense n-by-k matrix B (k<<n) [7]. SpMM is used in a variety of sparse matrix computations such as block Krylov subspace methods, as well as obtaining several eigenpairs of eigenvalue problems. Other applications that use SpMM include: (i) machine learning; (ii) image segmentation in videos, where a set of initial guess vectors are used to predict a succeeding frame in the video; (iii) aerodynamic design optimization; (iv) the search engine PageRank algorithm; and (v) atmospheric modeling.

Applying the observation that SpTM is a multi-dimensional analog of SpMM, existing SpTM implementations, such as in Tensor Toolbox [4] and **Cyclops Tensor Framework** (**CTF**) [29], transform an n-dimensional sparse tensor for $n > 2$ into an equivalent sparse matrix using a technique called *unfolding* [6]. They then use an SpMM kernel to perform the computation. These solutions were developed for CPUs and do not currently support GPUs. Sparse computation performance is significantly impacted by architecture, data layout, and nonzero structure of the input data. The massive parallelism, higher memory bandwidth, and availability of sparse tensor cores in GPUs make them well-suited for sparse tensor computations. This also provides us with the opportunity to leverage the hardware and take advantage of the efficient use of resources to significantly speed up the computation. Therefore, a GPU implementation that performs SpTM in terms of SpMM after unfolding the sparse tensor leverages prior work on sparse matrix libraries.

An alternative SpTM implementation strategy for both CPU and GPU is provided in the PASTA benchmark suite [17], which consists of various reference implementations for different tensor applications. SpTM is implemented for GPUs using a coordinate layout (COO) of the multi-way tensor, with lexicographic ordering.

We observe that prior work on CPUs that converts SpTM to SpMM has two main advantages. First, SpMM can be performed using existing highly tuned libraries. Second, the computational complexity of SpTM matches that of SpMM, which implies that SpTM should scale as well as SpMM does. On the other hand, Li et al. [17] state that an approach based on unfolding has two main drawbacks: the conversion time cost, and the irregular shape of the matrix after unfolding—potentially very tall or very wide. Therefore, they dismiss the unfolding data layout transformation approach and instead carry out the computation natively on the input tensor.

None of the approaches mentioned dominates performance across all available sparse tensor cases. Therefore, we use a decision tree classifier to identify the best performing tensor contraction kernel with 94.5% accuracy based on readily available features of the sparse tensor. Since we have a relatively small dataset size of only 55 sparse tensor contraction cases we used a decision tree classifier with cross-validation, which is well suited for smaller dataset sizes [13].

In this article, we explore an SpTM implementation on GPUs using a hierarchical tensor data layout. This layout can be unfolded to derive a matrix and allows us to plug-in different SpMM kernel implementations—for example, the highly-optimized SpMM kernels in NVIDIA's cuSPARSE library [24]—with the caveat that we initially need to convert the data layout to match that of cuSPARSE. We compare two SpMM implementations to PASTA's best performing sparse tensor contraction implementation: (1) manually-written SpMM kernel using the hierarchical tensor layout; and, (2) unfolding followed by an invocation of cuSPARSE's SpMM. In the rest of this article, we will refer to our manually-written kernel as the SpTM kernel.

This article makes the following contributions:

— it describes a hierarchical data layout that allows us to unfold or reshape tensors in multiple ways.
— it describes the two GPU approaches using the hierarchical layout and unfolding the data to a 2D matrix and then using either (1) a manually-written kernel code (SpTM) or (2) invoking cuSPARSE.
— it identifies the sparse tensor features required by the decision tree classifier to select the best performing sparse tensor contraction kernel.
— it provides extensive profile data on two different NVIDIA GPU architectures (V100 and A100 generations) to characterize the performance behavior of these implementations as compared to PASTA.

We ran experiments using a total of 55 sparse tensors on an A100 GPU and using 16 multi-vectors. The average performance speedup obtained by the kernel selected by the decision tree algorithm over only using PASTA was 6.42× and a maximum of 85.40× and a minimum of 1.00×. Even though the decision tree classifier mis-predicted three cases obtaining 94.5% accuracy, it used PASTA's kernel so we did not suffer any performance downgrade compared to the current state-of-the-art. Two out of the three cases were assigned PASTA's kernel when PASTA was not the best-performing kernel and one case was mis-assigned to cuSPARSE when the SpTM implementation was the best-performing kernel.

In the remainder of the article, we initially provide related definitions in the next section. Section 3 describes sparse tensor preprocessing phases and contraction. Afterward in Section 4, we provide details of the implementation. Section 5 compares performance measurements of the implementation. Section 6 describes related work. Section 7 describes work in progress, and the final section summarizes the motivation behind this work.

2 BACKGROUND

In this section, we provide definitions for terms that are used in the article; some of the definitions and examples are adapted from the Kolda et al. survey [15]. Most significantly, we describe the transformation of the tensor data layout using *unfolding*, also referred to as *tensor matricization* (Section 2.2).

2.1 Tensor Definitions

We focus on definitions that clarify the goals of the approach related to tensor computations we use. A tensor could be viewed as a group of fibers where a fiber is defined by fixing every index in

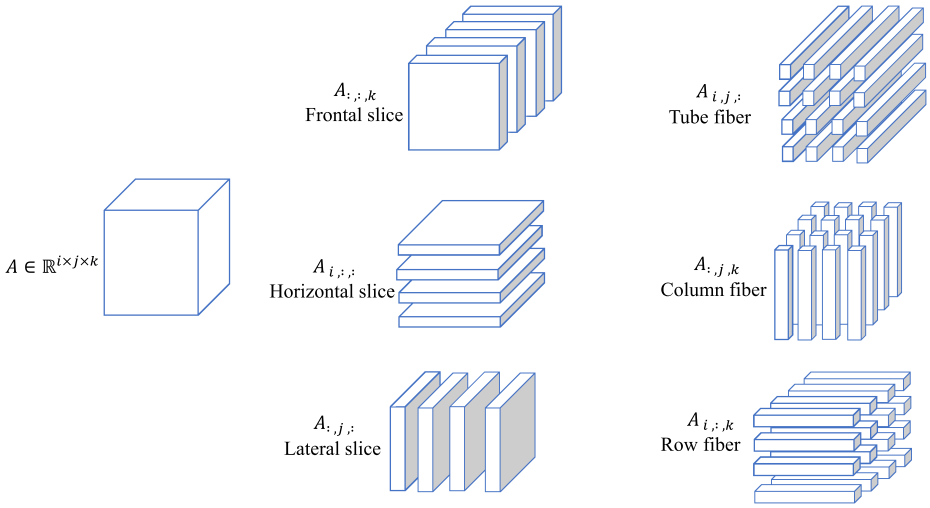


Fig. 2. Viewing tensors using slices and fibers.

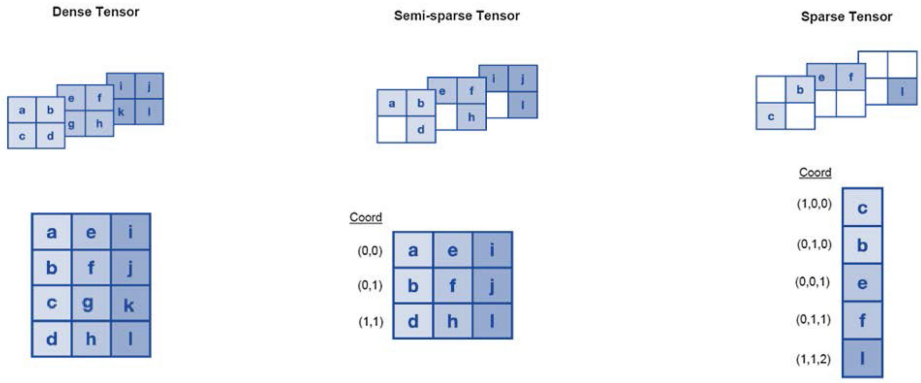


Fig. 3. Dense, semi-sparse, and sparse tensor examples.

a tensor but one. For example, Figure 2 shows a three-dimensional tensor A, which can be viewed as a group of fibers. When index k is the only unfixed index we get tube fibers, while fixing all other indices but i generates column fibers, and fixing all indices but j gives us row fibers. Another way of viewing a tensor is using the concept of slices, where slices are defined by fixing only one index variable. Figure 2 shows that tensor A can also be viewed as layers of slices. When we fix index k we get frontal slices, fixing index i generates horizontal slices, while fixing index j makes lateral slices. Slices and fibers help us explain semi-sparse tensors [18] which are tensors where each fiber at index (i, j) is dense. Figure 3 shows the difference between dense, semi-sparse, and sparse tensors.

Tensor matrix contraction on mode n, is called n-mode product, which is a product of multiplying tensor $A \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ by dense matrix $B \in \mathbb{R}^{I_n \times R}$ along the nth dimension producing semi-sparse tensor $C = A \times_n B$. For example, we can represent tensor contraction on the 3$^{\text{rd}}$ mode by

$$C(i, j, :) = \sum_{k=1}^{K} A(i, j, k)B(k, :).$$

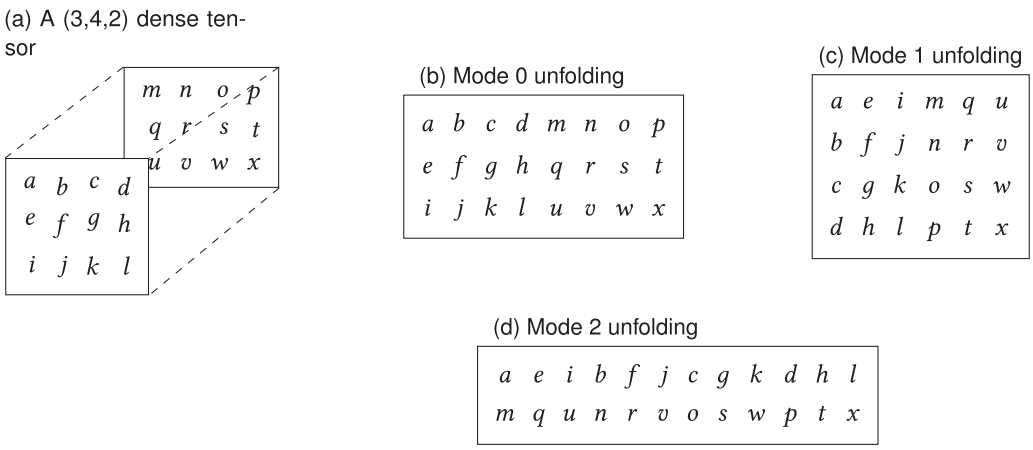


Fig. 4. (a) A (3,4,2) sample dense tensor unfolded into (b) mode 0 unfolding generates a (3,8) matrix (c) mode 1 unfolding generates a (4,6) matrix (d) mode 2 unfolding generates a (2,12) matrix.

Here we fix indices i and j in tensor A, which allows us to iterate over the tube fibers of A shown in the top right of Figure 2 and multiply them with their corresponding elements in dense matrix B.

Both fibers and slices are oriented based on the contraction mode. In this article, there are two main categories of modes we discuss. First, product mode is defined as the mode by which a tensor multiplies a matrix. Second, index mode is defined as all other modes of a tensor excluding the product mode. We need to take into consideration the tensor contraction mode since sparse tensor computations such as multiplication, transpose, or various point-wise operations require iterating over different modes in several orientations during the same computation. The simplest mode generic storage format for sparse tensors is COO format. However, the tradeoff is that COO is less compact than formats like CSF, which takes advantage of mode specificity to reduce indexing overhead cost per nonzero value.

2.2 Tensor Matricization or Unfolding

Tensors can be unfolded in any mode. This is done by selecting a subset of the modes of a given tensor A as the rows and the other modes of A as the columns and then mapping the elements of A to corresponding entries of the resulting matrix. For example, in Figure 4, we unfold the sample dense tensor along a product mode, which creates a dense matrix whose rows correspond to the product mode of the original tensor and the columns correspond to all the remaining modes, i.e., index modes. Each column of the matrix becomes a mode-1 fiber of the original tensor.

We aim to simplify an N-dimensional sparse tensor representation for $N > 2$, reducing its dimensionality to 2 so that high-performance sparse linear algebra libraries such as cuSPARSE [24] or manually-written kernels can be invoked to solve sparse tensor contraction. In this way, we convert sparse tensor contraction into a more common and easier SpMM-like kernel to perform the computation. Matricization reshapes a tensor into an equivalent matrix by arranging all mode-n fibers to be the columns of a matrix. Mathematically, unfolding a tensor is a logical restructuring of the tensor.

3 OVERVIEW

In this section, we describe the layout for an unfolded tensor, which uses a hierarchical representation to store the nonzero values (Section 3.1). To improve data reuse, deriving the layout is followed

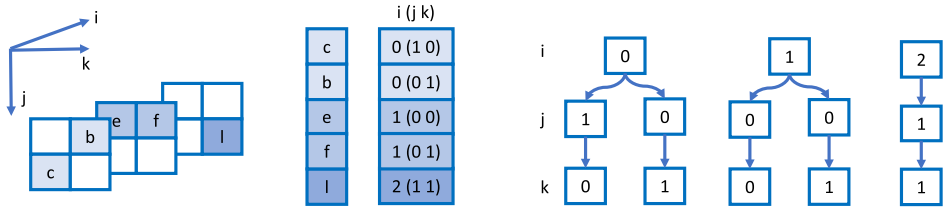


Fig. 5. Sorting of both row (mulit-index) and column (multi-index) of tensor A lexicographically.

by a lexicographic ordering of the nonzero values (Section 3.2). Finally, we give an overview of the tensor contraction operation (Section 3.3).

3.1 Hierarchical Data Layout

Our approach to unfolding a tensor uses a hierarchical tree-like structure to store nonzero values and their corresponding coordinates. The first level corresponds to the first mode of the tensor; for example, the x coordinate of the first nonzero value. The second level corresponds to the second mode of the tensor; for example, the y coordinates of the nonzero values. Lastly, the leaf nodes contain the values.

```
1  // create a tuple containing an int, a int, and a float
2  thrust::tuple<T, int, float> t(thrust::get<0>(t0), 23, 0.1f);
3  // individual members are accessed with the free funtion get
4  std::cout << "The first element's value is" <<thrust::get<2>(t) << std::endl
     ;
```

Listing 1. Thrust library tuple code.

To represent this hierarchical layout we used the tuple class template from the Thrust library [5] shown in Listing 1 to unfold tensors. Tuples can be instantiated with each argument specifying the type of element in the tuple. Consequently, tuples consist of a mix of tuple, integer, and float. Individual elements of a tuple may be accessed with the `get` function. For example, a (3,4,2) tensor can be unfolded into ((3,8),1), ((4,6),1), or ((2,12),1) matrices as shown in Figure 4 and using Equation (1).

$$A_{i_1 \ldots i_n i_{n+1} \ldots i_N} \rightarrow A_{(i_1 \ldots i_n)(i_{n+1} \ldots i_N)} = A_{mn}. \tag{1}$$

3.2 Tensor Reordering

We use lexicographic ordering [31] in tensor unfolding. Lexicographic ordering is a mapping from an n-dimensional space onto a linear space. One of the benefits of lexicographic ordering is that data close to each other in the folded N-dimensional tensor stays close to each other in the 2-D matrix. This helps with data reuse in the memory hierarchy and reduces total number of data movements required. Similarly, the row-major layout commonly used for CSR matrix formats can be done very efficiently by inducing the natural lexicographic ordering of the dimensions and iterating over the tensor in the natural order [23].

Lexicographical ordering sorts the nonzero values in a linearized manner. Use of tuples allows us to index nonzero values hierarchically. We sort the sparse tensor in order of modes depending on the product mode, and then we compare the indices in lexicographical order for each nonzero value. For example, if we look at Figure 4, it shows the dimension of each slice used to index into the coordinate values for each nonzero value; i.e., for nonzero value e located in the 1^{st} dimension, the coordinate values are $(0, 0)$.

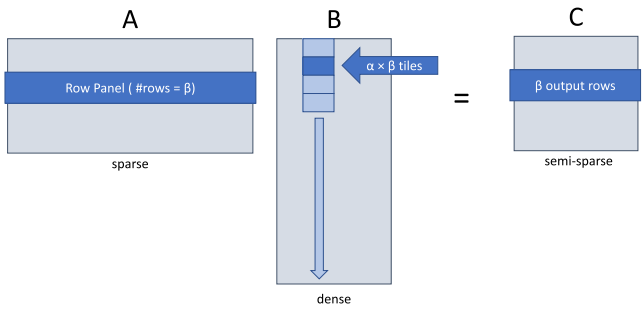


Fig. 6. Tensor contraction shaped into an SpMM computation.

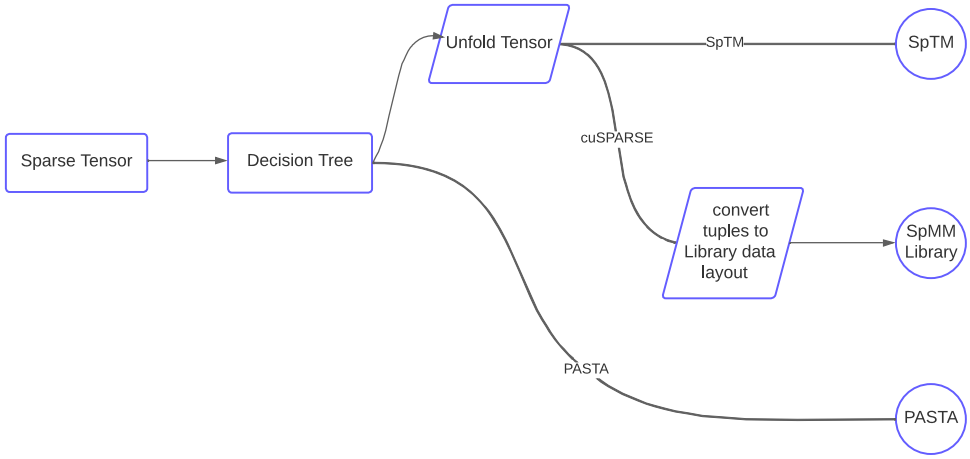


Fig. 7. Algorithmic approach for sparse tensor contraction.

3.3 Tensor Contraction

After unfolding and reorganizing the tensor, we perform the contraction operation, as will be discussed in Section 4. Here we show in Figure 6 that for a single-mode contraction after unfolding, we decompose the product into a series of 1-mode rows in A which, when multiplied by B, produces dense 1-mode rows of C. For matrix multiplication, the index space of rows is 1-dimensional while for tensor contraction the index space of rows in A is n-dimensional. This allows us to treat n-dimensional contraction as a matrix multiplication operation. The result of an SpTM operation is a sparse tensor with one or more dense modes, which is a semi-sparse tensor.

A difference between our approach and PASTA's mode-generic approach is that mode-generic approaches such as COO require inline search such as a binary search or a tree traversal to lookup the range of indices for nonzero values per row. On the other hand, mode-specific storage formats such as F-COO, CSR, and CSF require the reorganization of tensor A for every different mode of contraction. This makes lookup of the range of indices for nonzero values per row faster compared to mode-generic counterparts as it is pre-computed before the contraction kernel is called.

4 SPTT IMPLEMENTATION

In an SpTM tensor contraction, the inputs are a sparse tensor A and dense matrix B; the product is a semi-sparse tensor C. Figure 7 illustrates the main phases of our approach to solving SpTM.

Our implementation first reads the tensor data as coordinates and values and stores them in a tuple structure using thrust library tuples [5]. We also identify which mode of the tensor is the desired contraction mode. Afterward, we convert the COO data into CSR format and sort the values in lexicographical order as shown in the example of Figure 5. In this step, we simultaneously unfold the tensor into a matrix; this allows us to treat the SpTM computation as an SpMM computation. Following unfolding, we either continue the tensor contraction using the tuple data layout or opt for using a library's SpMM kernel implementation to perform the SpMM computation, in which case we will have to transform the data into the library's specific data structure first. Before we perform the computation on the GPU, we have to allocate memory for both dense matrix B and semi-sparse tensor C on the host and initialize the dense matrix B. Then we transfer the tensor data to the GPU with dense matrix B and allocate memory for semi-sparse tensor C. After that we can call the desired SpMM kernel to perform the tensor contraction operation and write back the results to the host CPU.

```
1  template <class ATensor, class BTensor, class CTensor>
2  __global__ void csr_sptm_kernel(ATensor const A,   // Sparse   (M,K)
3                                  BTensor const B,   // Dense    (N,K)
4                                  CTensor       C)   // Dense    (M,N)
5  {
6    // For each nonzero row in A
7    for (int m_idx = threadIdx.x + blockIdx.x*blockDim.x;
8         m_idx < nzr(A);
9         m_idx += blockDim.x * gridDim.x)
10   {
11     int idx_start = A.begin_nzr(m_idx);
12     int idx_end   = A.begin_nzr(m_idx+1);
13
14         float temp = 0;
15     for (int idx = idx_start; idx < idx_end; ++idx)
16     {
17         auto k = A.get_col_coord(idx);
18         temp += A[idx] * B(n, k);
19     }
20      C(m_idx, n) = temp;
21 }
```

Listing 2. Compressed Sparse Tensor Dense Tensor Multiplication CUDA kernel.

Using the kernel code in Listing 2 on the sample sparse tensor in Figure 5 we can see that line 7 assigns a thread to each row in the tensor, and then lines 11 and 12 identify the range of nonzero values in a row. In the example tensor, the first dimension corresponds to two nonzero values e and f. Line number 17 finds the column coordinates for those two values. In line 18, we identify the corresponding values in dense matrix B that we need to multiply with nonzero values from tensor A and accumulate the results in a temporary variable *temp*. Finally, we write back the results to semi-sparse tensor C in global memory at line 20.

Our approach is product mode-specific and index mode-generic, so if the computation does not change the product mode but swaps any of the index modes then this approach can be used as a mode generic approach. Table 1 compares the SpTM implementation and PASTA's approach in solving sparse tensor contraction.

Table 1. Differences Between Our Implementation of Tensor Contraction and PASTA's Implementation

Phase/Implementation	PASTA	SpTM	SpMM library
Storage format	COO	Tuples (Hierarchical)	library dependent (COO/CSR/BCSR)
Tensor unfolding	No	Yes	Yes
Ordering	Lexicographical	Lexicographical	Lexicographical
Mode specificity	Mode generic	Index modes generic	Mode specific
Contraction dimensions	Original number of dimensions	Column dimension contains all index dimensions	Two dimensions
Kernel	Manually-written	Manually-written	Library call

5 EXPERIMENTAL RESULTS AND EVALUATION

This section presents performance results and analysis. We begin by describing the experimental methodology.

5.1 Experimental Methodology

We describe the target architectures, input datasets, and the approach to data collection.

5.1.1 Target Architecture. A well-suited architecture for memory-intensive kernels like sparse computations is **graphics processing units** (**GPUs**). NVIDIA A100 GPUs exhibit peak performance of more than 15.7 Teraflops in single precision and peak memory bandwidth in excess of 1.9 TB/s. To evaluate the performance impact of our approach, we used two machines that are part of the University of Utah **Center for High Performance Computing** (**CHPC**) Notchpeak cluster. The first is a dual AMD 7502 CPU at 2.5 GHz, with 64 physical cores and 512 GB memory; it is equipped with an NVIDIA Tesla A100 (Ampere) GPU with 40 GB of global memory. The second machine is a dual Intel Xeon Gold 6130 CPU at 2.10 GHz, with 32 physical cores and 192 GB memory; it is equipped with an NVIDIA Tesla V100 (Volta) GPU with 16 GB of global memory, peak single-precision and double-precision performance of 14 and 7 TFlops, respectively, and total memory bandwidth of 900 GB/s. We used CUDA version 11.4 and GCC compiler version 8.1 with -O3 compiler options. As of this writing NVIDIA's cuSPARSE [24] and cuTENSOR [25] libraries do not support tensor computations, and it is an active area of research.

5.1.2 Input Datasets. For benchmarking, we used 13 of the 14 tensor groups from the **Formidable Repository of Open Sparse Tensors and Tools** (**FROSTT**) [27] collection, as listed in Table 2. We excluded one tensor group *Matrix Multiplication* as all the tensors in it are dense and too small to provide us with any meaningful insights. Using the remaining tensors, the total number of possible variations we have from using different modes of contraction is 58; every tensor can be unfolded into a different matrix shape or structure depending on the dimension we use to perform the unfolding. On the A100 GPU, we were able to collect performance data for a total of 55 cases. All techniques and implementations presented in this article are constrained by global memory capacity of the GPU. For example, tensors like *Patents* are larger than the GPU's 40 GB memory. This constraint is not unique to our implementation, as PASTA suffers from the global memory capacity issue too. The density of the matrices generated after unfolding the tensors had a maximum value of 5.00e−01 and a minimum of 8.63e−08 with an average of 2.98e−02. On the V100 GPU we only were able to run 36 variations as the V100 has less than half the global memory of the A100 GPU; therefore, several tensors such as *Amazon Reviews* and *Reddit-2015* failed to execute using both PASTA and our implementation due to global memory limitations. To maximize

Table 2. Sparse Tensors from FROSTT Collection

#	Name	Order	NNZ	Dimensions	Density
1	Amazon Reviews	3	1,741,809,018	4,821,207 x 1,774,269 x 1,805,187	1.13E-10
2	Chicago Crime	4	5,330,673	6,186 x 24 x 77 x 32	1.46E-02
3	Delicious	4	140,126,181	532,924 x 17,262,471 x 2,480,308 x 1,443	4.26E-15
4	Enron E-mails	4	54,202,099	6,066 x 5,699 x 244,268 x 1,176	5.46E-09
5	Flickr	4	112,890,310	319,686 x 28,153,045 x 1,607,191 x 731	1.07E-14
6	LBNL-Network	5	1,698,825	1,605 x 4,198 x 1,631 x 4,209 x 868,131	4.23E-14
7	NELL-1	3	143,599,552	2,902,330 x 2,143,368 x 25,495,389	9.05E-13
8	NELL-2	3	76,879,419	12,092 x 9,184 x 28,818	2.40E-05
9	NIPS Publications	4	3,101,609	2,482 x 2,862 x 14,036 x 17	1.83E-06
10	Patents	3	3,596,640,708	46 x 239,172 x 239,172	1.37E-03
11	Reddit-2015	3	4,687,474,081	8,211,298 x 176,962 x 8,116,559	3.97E-10
12	Uber Pickups	4	3,309,490	183 x 24 x 1,140 x 1,717	3.85E-04
13	VAST 2015 Mini-Challenge 1	5	26,021,945	165,427 x 11,374 x 2 x 100 x 89	7.77E-07

Table 3. Shape Variations Due to Different Modes of Contraction on Nell-1 Tensor

#	Contraction Mode	Number of Rows	Number of Columns	Density
1	0	2,902,330	113,299,246	4.37e−07
2	1	2,143,368	119,126,880	5.62e−07
3	2	25,495,389	17,372,417	3.24e−07

the number of tensors that we can benchmark, we also opted to use single precision floating point storage and operations across all implementations.

In some cases like *nell-1*, the tensor when unfolded into a matrix—as Table 3 details—the number of columns varies considerably which in turn affects the number of rows in the dense matrix B leading to problem sizes large enough to be terminated. So even though we are operating on the same sparse tensor and the same number of nonzero values in sparse tensor A, the shape of dense matrix B is dictated by product mode which varies the memory footprint of the computation, sometimes leading to an oversubscription of the GPU's memory capacity.

Sparse storage formats are usually used as they require less memory footprint to store nonzero values. A back-of-the-envelope calculation to compare the memory footprint required to handle tensor A using PASTA's COO-based approach and our CSR-based approach shows that we save around 50% storage footprint on average compared to PASTA.

5.1.3 Data Collection. For all of the experiments conducted, we used the same random seed to populate the multi-vector matrix B. We tested several random seeds. We also studied the impact of increasing the magnitude of random values in the multi-vector matrix B using the following ranges [1, 10, 100, 1000] on both execution time and accuracy. We observed no significant change in either accuracy of the final result or the execution time.

We validate the correctness of the output as compared to a CPU version of sparse tensor contraction by examining each element of the output semi-sparse tensor. There are insignificant differences in the range of 10e−06 to 10e−07, which is consistent with the accuracy single precision floating point provides.

5.2 Performance Evaluation

Figure 8 shows the SpTM kernel and cuSPARSE SpMM kernel performance compared to the current state-of-the-art implementation for sparse tensor contraction available from the PASTA suite [17].

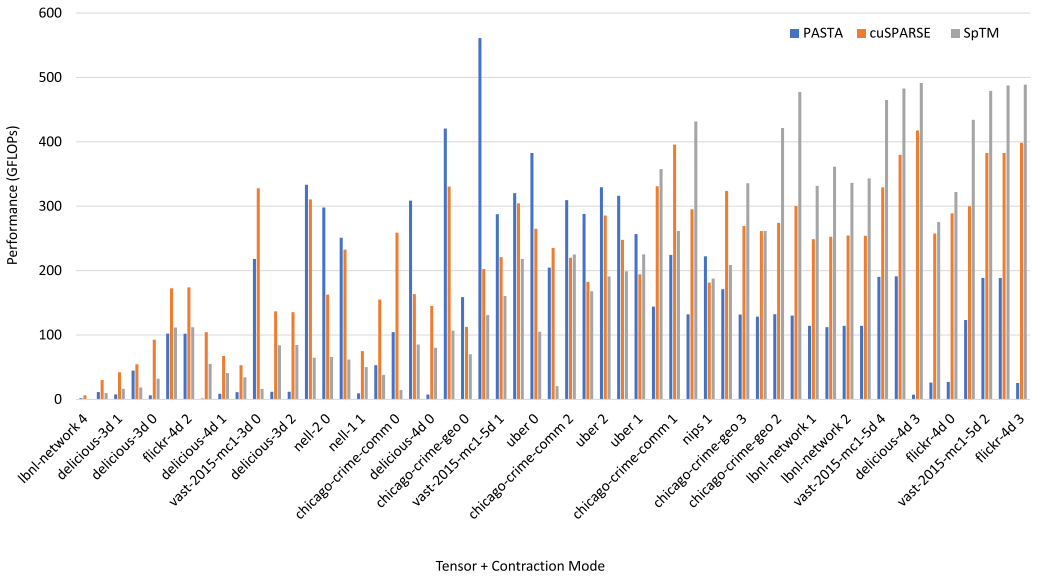


Fig. 8. Performance of tensor contraction with N = 16 on Ampere 100 GPU using PASTA, cuSPARSE, and SpTM kernels.

N is the number of multi-vectors or columns in the dense matrix part of the contraction. The number of columns of the dense matrix is set to 16 which is the usual size in tensor decompositionâĂŹs lowrank property [18]. On the x-axis, each test case is designated by the tensor name followed by the contraction mode of the tensor.

5.2.1 Load Balancing. From our experiments, we noticed a correlation between performance and the distribution of work per row of the unfolded tensor. Therefore, we used the load balancing ratio shown in Equation (2) to sort the performance of different tensor contraction cases in descending order. Load balancing ratio measures the difference between the maximum and minimum number of nonzero values of different rows, which is a proxy for the irregularity of a matrix. The lower the ratio, the less difference there is between the minimum and maximum number of nonzeros per row. Low ratios lead to a more balanced computation, as each row of the unfolded tensor has approximately equal amount of work that needs to computed.

$$LB = \frac{max(nnz/row)}{min(nnz/row)}. \tag{2}$$

As we can see from Figure 8, the more unbalanced the tensor is (left side of graphs), PASTA's implementation usually outperforms our SpTM implementation. We also note that load balancing is generally a good indicator of performance, but that is not always the case. For example, *vast-2015-mc1-3d* mode 0, *enron* mode 2, *nell-2* modes 0 and 2 all show lower performance with PASTA's implementation even though these cases suffer from high load imbalance. To understand the reasons behind the performance gains in these results, we examined the output of `Nsight`, discussed next.

5.2.2 Memory Behavior and Reorganization Cost. We record the performance of 100 iterations of each sparse tensor contraction kernel on GPU computations, and we report the average performance in GFLOPs. The standard deviation for the measurements obtained from all three kernels is less than 1 GFLOPs, which indicates that the measurements are close to the average value of

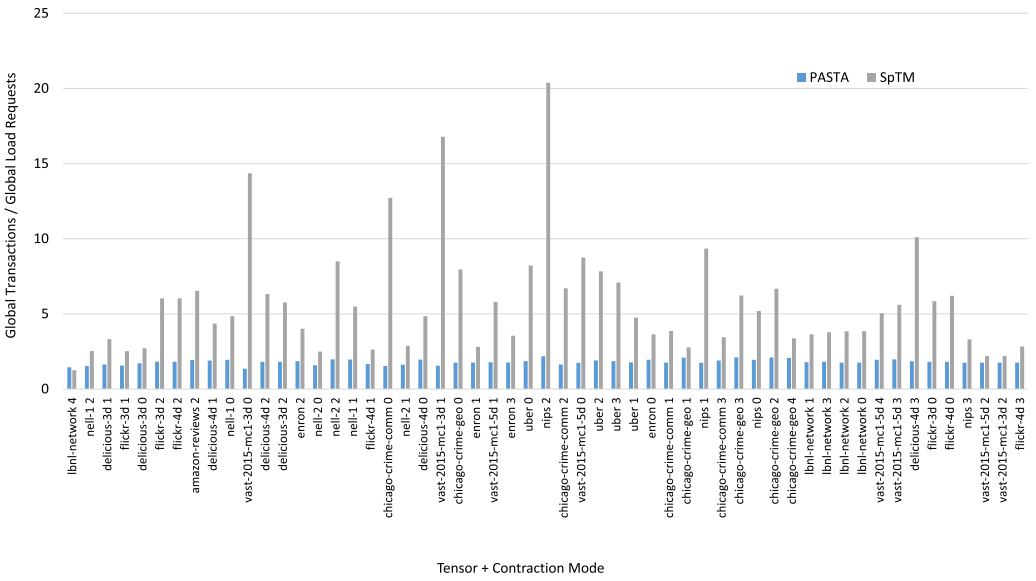


Fig. 9. Coalescing proxy measurements on A100, lower means more coalesced with a maximum of 32.

the measurements. This approach is consistent with how PASTA's sparse tensor contraction implementation is tested, where the relevant data is also transferred from the host to the device memory for the computation to take place.

PASTAâĂŹs implementation focuses on coalescing both dense matrix B and product tensor C. The dense matrix can be reused by threads with the same thread id. Large sparse tensors often have low reuse degree because of the very sparse nonzero values distribution. For example, the density of *nell2* tensor is 2.40e-05. Thus, the reuse of the dense matrix is negligible.

We collected global load transactions and global load requests using NVIDIA `Nsight` CUDA profiler to measure and determine how well SpTM coalesces loads and stores, based on Equation (3) below:

$$Coalescing = \frac{gld\_transactions}{global\_load\_requests}. \tag{3}$$

From the current `Nsight` manual, the required flags needed to collect the denominator global load requests and the numerator global transactions count in Equation (3) are

$$l1tex\_\_t\_requests\_pipe\_lsu\_mem\_global\_op\_ld.sum \text{ and}$$
$$l1tex\_\_t\_sectors\_pipe\_lsu\_mem\_global\_op\_ld.sum$$

respectively. If we calculate this ratio, we get 32 transactions per request. Therefore, each thread in the warp is generating a separate transaction. Higher values indicate that memory access patterns are not coalesced. Figure 9 shows that PASTA's kernel implementation produces a consistently coalesced access regardless of the tensor or the mode of contraction while our SpTM implementation is affected by the distribution of the data being processed. This observation is similar on both the V100 and A100 GPUs.

Furthermore, we collected global store transactions measurements using Equation (4). Figure 10 shows the transactions issued from the warp targeting the global address space. These global transactions could hit in a cache and not need to access lower levels of the memory hierarchy. We see that PASTA's tensor contraction algorithm performance suffers when the global store transactions

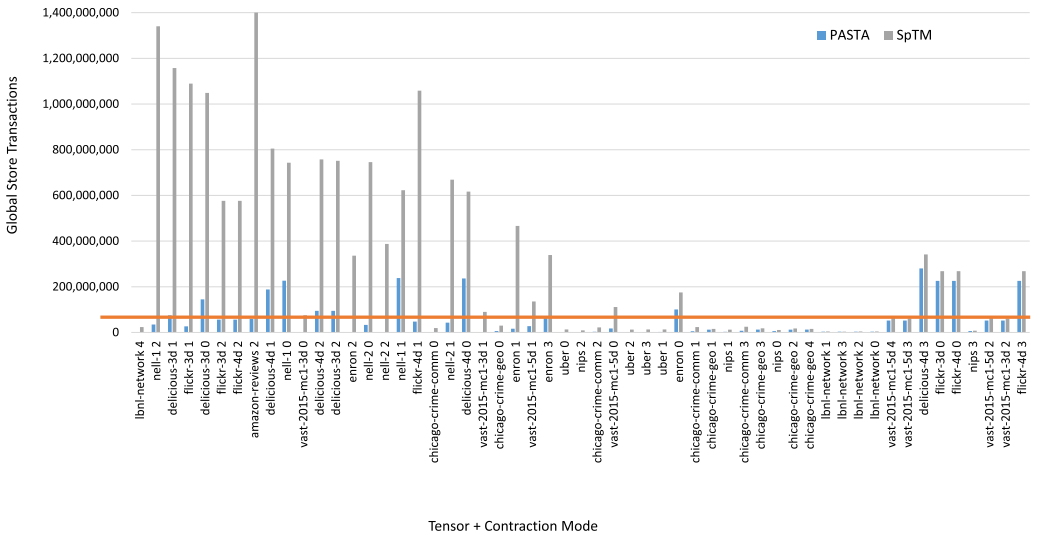


Fig. 10. Global store transactions on A100. Orange line indicates 59 million global store transactions. PASTA's tensor contraction algorithm performance suffers when the global store transactions are above the orange line.

are high, greater than 59 million transactions depicted by the orange line in Figure 10; on the other hand, the SpTM implementation does not seem to be as sensitive to global store transactions.

$$GlobalStoreTransactions = l1tex\_\_t\_sectors\_pipe\_lsu\_mem\_global\_op\_st.sum. \quad (4)$$

For completeness we measured the time required to to convert the sparse tensors from COO format to CSR, unfolding and sorting as depicted in Figure 7. To put it into context, the conversion takes place once at runtime, prior to running the iterative SpTM phase in Tucker decomposition. For an n-mode Tucker decomposition [21], there are n different sorting orders throughout the entire algorithm, which can be reused among iterations. An index mode sequence of tensor times matrix multiplications can be ordered such that the sorting order of the input sparse tensor A keeps the same order for their sparse modes. Hence, only one sorting per tensor contraction sequence for a sparse tensor A is needed. Therefore, the one-time cost of this conversion is amortized over numerous iterations, and the performance gain of the unfolding and sorting in our implementation reduces the execution time of each iteration. Furthermore, we did not spend any time to optimize or parallelize our conversion process and unfolding approach, which is currently performed on the CPU side of the computation.

Figure 11 shows the slowdown and speedup of PASTA's approach in sorting the sparse tensor compared to our implementation, which converts sparse tensor from COO format to CSR, unfolds and sorts it. In Figure 11, anything above the black line indicates that PASTA is faster at sorting the sparse tensor. On average, we see that PASTA's approach is 1.33× faster.

5.2.3 Multi-Vector Size Variation Behavior. Figure 12 shows the performance of SpTM on tensors with increasing number of multi-vectors. We only test on small sizes because as we mentioned previously that tensor decomposition uses lower number of multi-vectors. Furthermore, when we use 32 multi-vectors PASTA fails to run a single test case out of 55 and at 64 multi-vectors PASTA's failed cases jumps to 8. This makes it harder to compare our implementation's performance to PASTA using higher numbers of multi-vectors. We note that the increase in number of

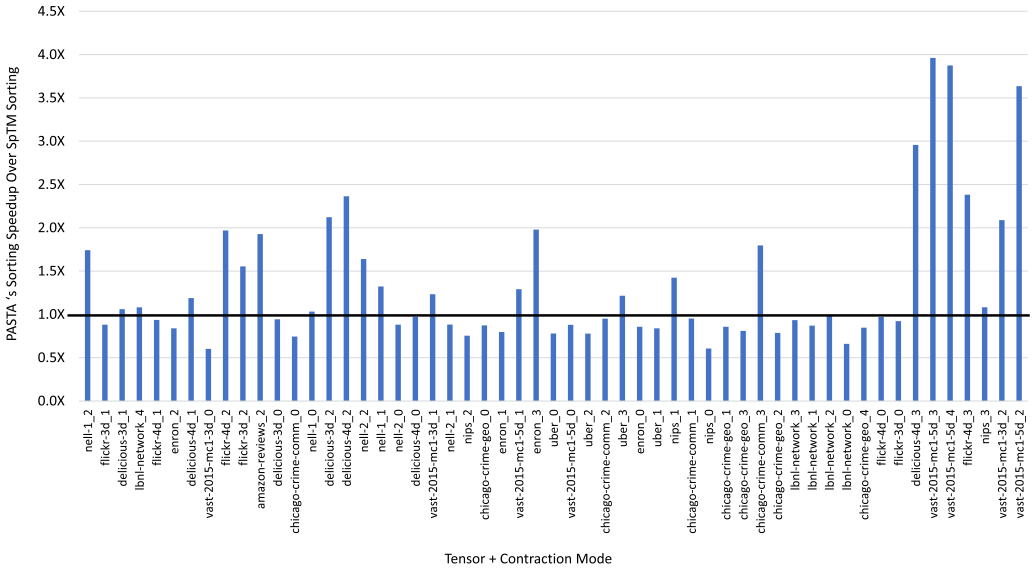


Fig. 11. PASTA's speed at sorting a tensor compared to SpTM. The black line indicates 1X. Any bar above the line means PASTA is faster at sorting the tensor while any bar below the black line indicates that SpTM is faster than PASTA.

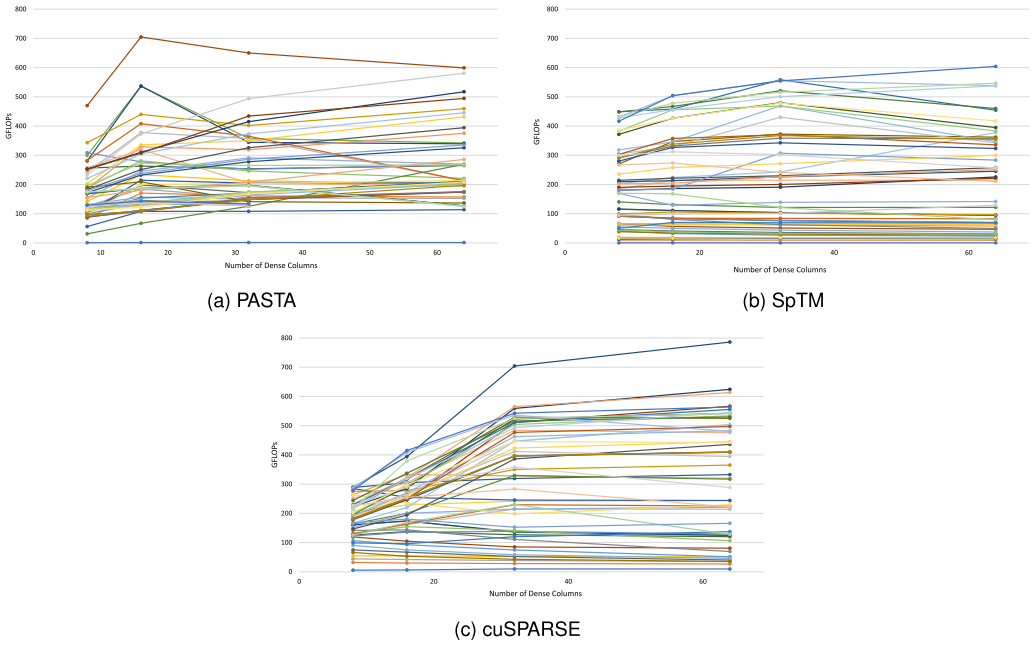


Fig. 12. Execution time of tensor contraction with increasing number of dense columns for (a) PASTA, (b) SpTM, and (c) cuSPARSE.

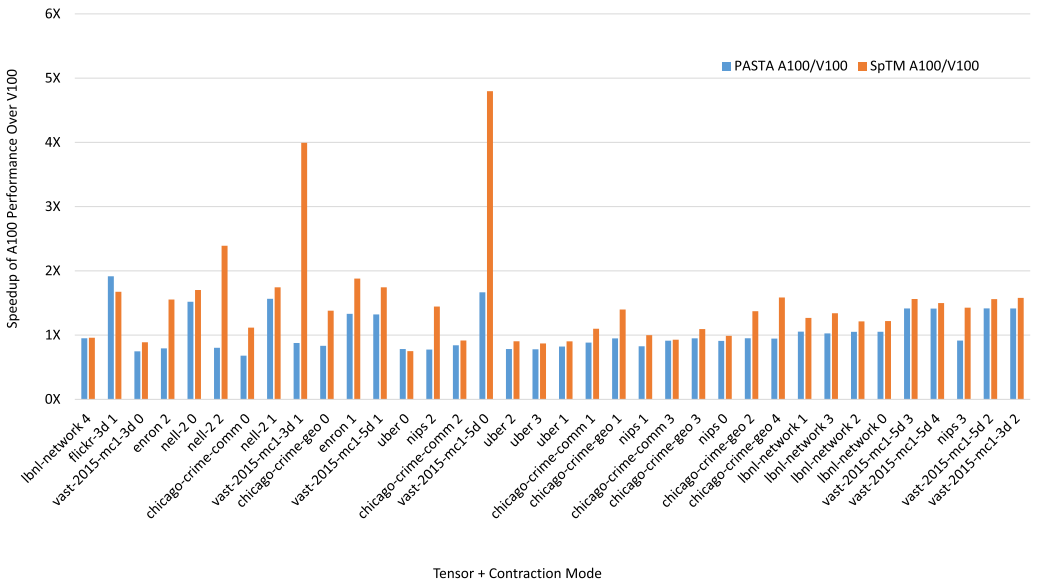
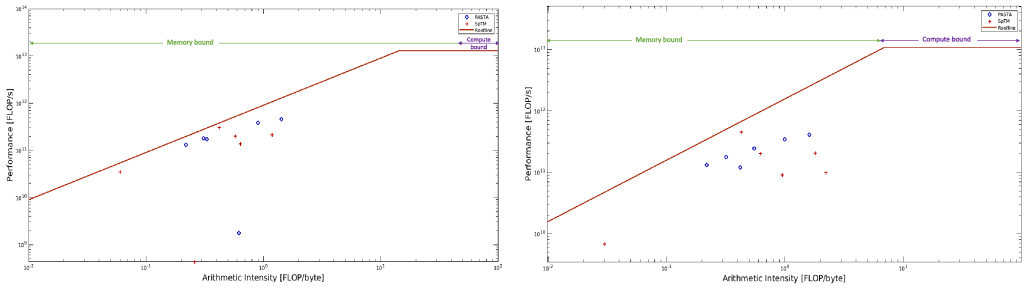


Fig. 13. Our SpTM approach takes advantage of newer hardware accelerators compared to PASTA.

multi-vectors generally leads to an increase in the performance for all three implementations. However, we note that PASTA shows a jump in performance when the number of multi-vectors increases from 8 to 16 compared to other implementations. By comparison, cuSPARSE keeps on achieving higher performance with increasing number of multi-vectors; this is in contrast to PASTA and SpTM performance where both reach a plateau with a higher number of multi-vectors. Even though PASTA performs very well at multi-vector size 16 compared to other implementations, our implementation still achieves higher arithmetic intensity on 36 out of the 55 test cases, or 65.5%, with a maximum of 3.62 achieved *arithmetic intensity*, the ratio of computation (measured in FLOPs) to memory traffic (measured in bytes).

5.2.4 Roofline Model. We observed that our implementation improves its relative performance compared to PASTA's implementation as we move the computation from the V100 to the A100 GPUs. On 36 tensor cases that run without any errors on both V100 and A100 GPUs, the SpTM implementation achieves an average of 1.49× speedup, a 0.75× minimum, and a maximum of 4.80× using A100 GPU compared to a V100 GPU; PASTA's implementation only achieves an average of 1.05×, 0.68× minimum and maximum of 1.92×. Speedup of A100 over V100 performance for both PASTA and our SpTM implementation are shown in Figure 13.

We also performed roofline model analysis using the **Empirical Roofline Tool** (**ERT**) [20] included in NVIDIA's Nsight which measures the GPU's characteristics and generates Roofline data. Roofline models allow us to analyze the limitations of an implementation, mainly determining if it is memory bandwidth limited or compute limited. We profiled both our SpTM implementation and PASTA's implementation and compared the results on both V100 and A100 GPUs as shown in Figure 14. We selected the sparse tensor cases that correspond to the minimum, median, and maximum performance for both PASTA and SpTM implementations, totalling 6 sparse tensor cases. However, for cuSPARSE Nsight did not generate achieved arithmetic intensity values in the Roofline model; therefore, we could not show roofline graphs for cuSPARSE. Both PASTA and SpTM implementations fall in the memory bandwidth-bound part of the roofline chart (under the



(a) V100 (b) A100

Fig. 14. Roofline model of sparse tensor contraction with 16 multi-vectors for PASTA (blue circles) and SpTM (red plus sign) on (a) V100 GPU and (b) A100 GPU. Note that both x-axis and y-axis are on logarithmic scale.

left side and slanted part of the chart), which is expected in the case of a sparse computation as the memory access patterns are the limiting factor.

We note that SpTM's arithmetic intensity increases when running on an A100 GPU compared to V100 GPU as the red colored plus sign data points in Figure 14 show a shift to the right side of the plot closer to the compute region of the roofline model. However, PASTA's implementation did not show a similar increase in arithmetic intensity when running on A100 GPU compared to a V100 GPU.

5.3 Classifying Best-Performing Implementation

The performance results in the previous subsections show that none of the implementations perform optimally across all sparse tensor cases. Moreover, the load balancing ratio is not a sufficient predictor of performance of the three kernels. To provide an automated scheme for selecting the best-performing implementation, we used MATLAB's decision tree classifier to create a decision tree using available measurements. The classifier function call shown in Listing 3 identifies the required features that select the best-performing implementation. To train the decision tree, we used seven features of the sparse tensor, available at run time: number of nonzeros, number of rows multiplied by number of columns of the unfolded tensor, ratio of rows to columns of the unfolded tensor, log of the rows to column ratio, the load imbalance ratio of the unfolded tensor, load imbalance ratio divided by the log of rows to columns ratio, and unfolded tensor density information. We used the best-performing single precision kernel on the A100 GPU as the target for the classifier. We trained the classification decision tree using default values for tree depth control and cross-validated the model using 10-fold cross-validation. The decision tree algorithm selected only three out of the seven features shown in Figure 15 as the necessary features needed to identify the best-performing sparse tensor contraction kernel. The three features are load imbalance ratio of the unfolded tensor, number of nonzeros, and rows to column ratio of the unfolded tensor. Performance of kernels selected by the decision tree verses PASTA are shown in Figure 16.

```
1    fitctree(X,Y,'CrossVal','on');
```

Listing 3. Matlab decision tree classifier.

The decision tree starts by using the load imbalance ratio to identify all the tensor contraction cases where the manually-written SpTM kernel outperforms both PASTA and cuSPARSE. Afterward, the decision tree uses the number of nonzero values of the tensor as an arbiter to identify the sparse tensor cases where cuSPARSE outperforms PASTA, and finally the shape of the unfolded

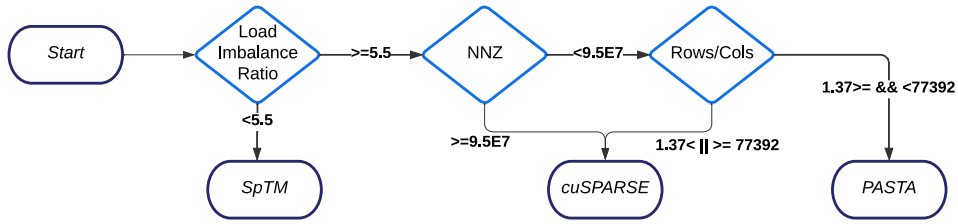


Fig. 15. Decision tree classifier identifies the best performing tensor contraction kernel.

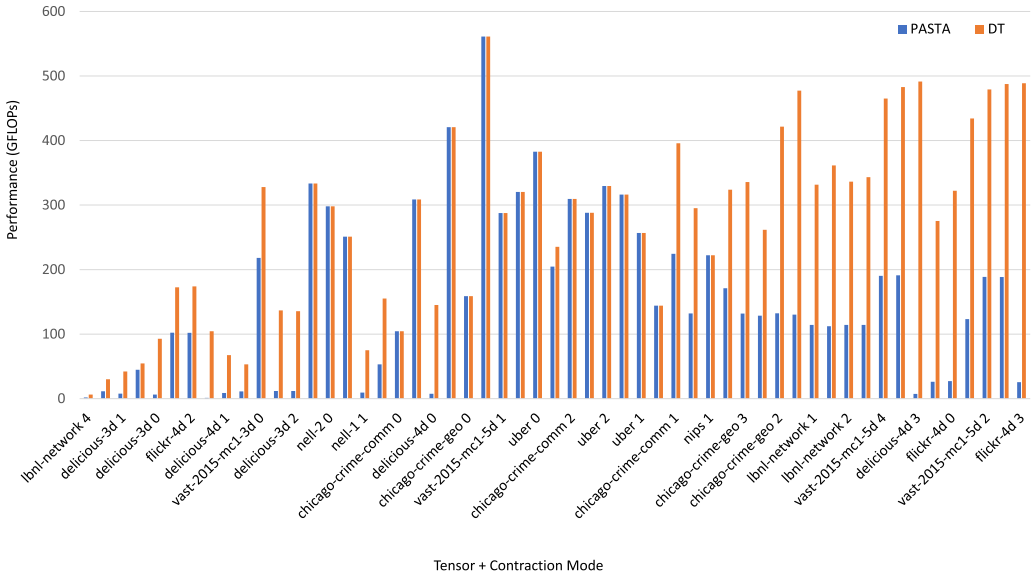


Fig. 16. Decision tree selected kernel performance verses PASTA's tensor contraction performance using N = 16 on Ampere 100 GPU.

sparse tensor is used to identify the sparse tensor cases where PASTA delivers the best performance. The decision tree mis-classified only three cases out of 55, which gives us an error rate of 5.45%.

6 RELATED WORK

6.1 Tensor Computations

Kolda et al. [15] wrote a comprehensive overview of tensor decomposition literature and tensor decomposition methods such as CP or Tucker. They also mathematically define unfolding of tensors. A tensor can be decomposed into the multiplication of matrices or vectors based on the nonzero entries. The function guiding the decomposition minimizes the error between the multiplication of the decomposed matrices or vectors and the nonzero values of the tensor. Tensor decomposition method finds acceptable values for the unknown values in a tensor through multiplying the tensor's decomposed matrices or vectors.

Ma et al. [21] developed parallel algorithms for **sparse tensor times dense matrix multiply** (**SpTTM**) for both CPU and GPU. Their parallel algorithm shows 4.1× speedup on multicore Intel Core i7 and 18.8× speedup on NVIDIA K40c GPU over their sequential SpTTM implementation, respectively. They conclude that different input sparse tensors, multi-vectors, and operating on different modes all influence SpTTM's performance.

Li et al. [19] also proposed hierarchical variation of the coordinate format called HiCOO that breaks down a sparse tensor into small sparse blocks. This approach reduces the memory requirements to store nonzero values from the tensor. They showcase their approach on the **matricized tensor-times-Khatri-Rao product** (**MTTKRP**) algorithm with small thread-count architectures such as CPUs. More recently Li et al. gathered essential sparse tensor operations from various real-world applications for multicore CPU and GPU architectures in a single parallel sparse tensor algorithm Benchmark suite [17]. These tensor operations are critical to the overall performance of tensor analysis algorithms such as tensor decomposition. Recent related work encodes multi-dimensional sparse tensors in a mode-agnostic way called linear format (ALTO) [9]. They developed their approach for MKTTRP and tested it on a CPU architecture.

The **Surprisingly ParalleL spArse Tensor Toolkit** (**SPLATT**) [28] is a C library for sparse tensor factorization on CPUs. It uses CSF format to perform SpMTTKRP and SPTTMc with options of either shared memory parallelization using OpenMP or distributed-memory parallelism using MPI.

TACO compiler [14] is a sparse tensor algebra compiler that can compile a basic tensor algebra expression with any number of additions and multiplications and operands can be stored in many different data structures. To date, high-performance GPU implementations have not been a focus of work on TACO.

6.2 SPMM

There are several existing high-performance GPU implementations of SpMM. NVIDIA's cuSPARSE library [24] is highly optimized for performance on NVIDIA GPUs, with SpMM performance 30–150× faster than CPU-only alternatives. It supports dense, COO, CSR, CSC, and Blocked CSR sparse matrix formats. Yang et al. [30] applied row-splitting and merge-based [22] algorithms to SpMM to efficiently hide global memory latency. Based on the matrix sparsity pattern, one of the algorithms is applied. Huang et al. [12] developed an acceleration method for **general-purpose SpMM** (**GE-SpMM**) algorithm for GPUs. Their work is tailored for **Graph Neural Networks** (**GNNs**) as it can expedite computations such as **graph convolutional networks** (**GCN**) and pooling-based GNNs. Hong et al. [10] solved SpMM using a dynamic load-balancing approach that takes advantage of tiling. They split the sparse matrix into heavy workload and light workload rows. The heavy work load rows are processed by CSR and the light work load rows are processed by **doubly compressed sparse row** (**DCSR**). Abu-Sufah et al. [1] designed and implemented two CUDA SpMM kernels targeted at structured matrices (DIA-SpMM) and matrices with uniform row lengths (ELL-SpMM). They evaluated the performance of their kernels with SpMV kernels in NVIDIA CUSP and cuSPARSE libraries, SpMM kernel available in cuSPARSE and SpMV kernels available in Intel MKL library executing on multi core processors. The author's kernels showed superior performance because they; (i) used the most efficient storage schemes for the test matrices, (ii) multiplied each matrix by a block of vectors instead of one, and (iii) made the most of GPU registers to exploit data reuse in SpMM.

More broadly, Gale et al. [8] developed a library sparse linear algebra library targeted for deep neural networks on GPUs called Sputnik. It uses optimization techniques such as row reordering, memory alignment, and tiling. These optimizations allow it to outperform state-of-the-art baselines such as Adaptive sparse tiling for sparse matrix multiplication (ASpT) [11] and cuSPARSE, on deep learning inference models. Ahmad et al. [2] optimized an important sparse matrix multivector multiplication kernel within the **Locally Optimal Block Preconditioned Conjugate Gradient** (**LOBPCG**) solver and showed a 3% performance enhancement over the manually-tuned code for the same algorithm [3]. Using an inspector-executor approach, a data transformation converted the large, symmetric sparse matrix from a compressed sparse row format to a **compressed sparse**

block (**CSB**) format. This representation was well suited for parallelization of the matrix and its transpose since it permitted storing only the upper triangular portion. In order to reduce the data movement associated with indices of the matrices, a short integer was used as the type for the matrices that pointed to the beginning of each CSB block.

7 CONCLUSION

This research uses a hierarchical tensor data layout that can unfold a multi-dimensional tensor to derive a two-dimensional matrix, making it possible to compute SpTM using existing SpMM kernel implementations for GPUs. Furthermore, we utilize a decision tree classifier with cross-validation to identify the best performing tensor contraction kernel with 94.5% accuracy based on features extracted from the sparse tensor without any preprocessing overhead. In sparse tensor contraction operations, the first step is to matricize the tensor in the desired mode and then perform the multiplication. This work uses multiple indices and mapping functions to perform tensor contraction without the explicit need to matricize the original tensor. Preserving the indices allows us to circumvent the main disadvantage of tensor unfolding stated in prior state-of-the-art [21], namely, the potential of generating a matrix that can be very large and exceeds the range of a 64-bit unsigned integer. This hierarchical data layout optimization also allows us to treat any order tensor contraction as a more common SpMM operation, which has been extensively researched [24] [30] [12] [8] [11] [2] [3] [10] [1].

REFERENCES

[1] Walid Abu-Sufah and Khalid Ahmad. 2014. On implementing sparse matrix multi-vector multiplication on GPUs. In *Proceedings of the 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and System (HPCC, CSS, ICESS)*. IEEE, 1117–1124.

[2] Khalid Ahmad, Anand Venkat, and Mary Hall. 2016. Optimizing LOBPCG: Sparse matrix loop and data transformations in action. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. Springer, 218–232.

[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1213–1222.

[4] Brett W. Bader, Tamara G. Kolda, et al. 2015. Matlab tensor toolbox version 2.6. Available online. Retrieved from https://www.tensortoolbox.org/

[5] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *Proceedings of the GPU Computing Gems Jade Edition*. Elsevier, 359–371.

[6] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications* 21, 4 (2000), 1253–1278.

[7] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software* 28, 2 (2002), 239–267.

[8] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[9] 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*. 404–416.

[10] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient sparse-matrix multi-vector product on GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 66–79.

[11] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 300–314.

[12] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2017. *An Introduction to Statistical Learning*. Springer.
[14] Fredrik Berg Kjølstad. 2020. *Sparse Tensor Algebra Compilation*. Ph.D. Dissertation. Massachusetts Institute of Technology.
[15] Tamara G. Kolda and Brett W. Bader. 2009. Tensor decompositions and applications. *SIAM Review* 51, 3 (2009), 455–500.
[16] Daniel Langr and Pavel Tvrdik. 2016. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2016), 428–440.
[17] Jiajia Li, Mahesh Lakshminarasimhan, Xiaolong Wu, Ang Li, Catherine Olschanowsky, and Kevin Barker. 2020. A sparse tensor benchmark suite for CPUs and GPUs. In *Proceedings of the 2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 193–204.
[18] Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. 2016. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *Proceedings of the 2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. IEEE, 26–33.
[19] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 238–252.
[20] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. 2015. Roofline model toolkit: A practical tool for architectural and program analysis. In *Proceedings of the High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014*. Springer, 129–148.
[21] Yuchen Ma, Jiajia Li, Xiaolong Wu, Chenggang Yan, Jimeng Sun, and Richard Vuduc. 2019. Optimizing sparse tensor times matrix on GPUs. *Journal of Parallel and Distributed Computing* 129 (2019), 99–109. https://www.sciencedirect.com/science/article/abs/pii/S0743731518305161
[22] Duane Merrill and Michael Garland. 2016. Merge-based sparse matrix-vector multiplication (spmv) using the CSR storage format. *ACM SIGPLAN Notices* 51, 8 (2016), 1–2.
[23] Suzanne Mueller, Peter Ahrens, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Sparse tensor transpositions. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 559–561.
[24] NVIDIA. 2021. CUSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html
[25] NVIDIA. 2021. CUTENSOR. (2021). https://docs.nvidia.com/cuda/cusparse/index.html
[26] Yang Shi, Uma Naresh Niranjan, Animashree Anandkumar, and Cris Cecka. 2016. Tensor contractions with extended BLAS kernels on CPU and GPU. In *Proceedings of the 2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 193–202.
[27] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. 2017. The Formidable Repository of Open Sparse Tensors and Tools. http://frostt.io/
[28] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 61–70.
[29] Edgar Solomonik and Torsten Hoefler. 2015. Sparse tensor algebra as a parallel programming model. arXiv:1512.00066. Retrieved from https://arxiv.org/abs/1512.00066
[30] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design principles for sparse matrix multiplication on the GPU. In *Proceedings of the European Conference on Parallel Processing*. Springer, 672–687.
[31] Shmuel Zaks. 1980. Lexicographic generation of ordered trees. *Theoretical Computer Science* 10, 1 (1980), 63–82.

Received 20 May 2023; revised 19 September 2023; accepted 2 November 2023