SpDISTAL: Compiling Distributed Sparse Tensor Computations

Rohan Yadav
Stanford University
Stanford, CA
rohany@cs.stanford.edu

Alex Aiken
Stanford University
Stanford, CA
aiken@cs.stanford.edu

Fredrik Kjolstad
Stanford University
Stanford, CA
kjolstad@cs.stanford.edu

***Abstract*—We introduce SpDISTAL, a compiler for sparse tensor algebra that targets distributed systems. SpDISTAL combines separate descriptions of tensor algebra expressions, sparse data structures, data distribution, and computation distribution. Thus, it enables distributed execution of sparse tensor algebra expressions with a wide variety of sparse data structures and data distributions. SpDISTAL is implemented as a C++ library that targets a distributed task-based runtime system and can generate code for nodes with both multi-core CPUs and multiple GPUs. SpDISTAL generates distributed code that achieves performance competitive with hand-written distributed functions for specific sparse tensor algebra expressions and that outperforms general interpretation-based systems by one to two orders of magnitude.**

I. Introduction

Sparse tensor algebra is ubiquitous and has applications across many fields, including scientific computing, machine learning, and data analytics [1]–[5]. These application domains operate on ever-increasing amounts of data, and can benefit from the growing compute and memory offered by modern distributed machines. However, efficiently utilizing distributed machines for sparse computations remains difficult.

We present SpDISTAL, a system that compiles sparse tensor algebra to distributed machines. SpDISTAL allows for independent descriptions of how data and computation should be mapped onto the memories and processors of a target machine. Figure 1 shows C++ code implementing a distributed SpMV using SpDISTAL. Lines 12-22 describe the sparse format and data distributions of tensors through a *format language*, and lines 30-39 describe a row-based distribution strategy through a *scheduling language*. These separate languages for data and computation allow for independently changing the data format or distribution and the algorithm to distribute the computation.

Efficient kernel implementations for sparse tensor algebra operations can be complex, even on a single thread. Distributing these computations makes it even harder to ensure correctness and performance. There are two modern approaches to tackling this complexity: a library of kernels and interpretation. Examples of libraries of hand-written kernels include PETSc [6]–[8] and Trilinos [9], [10]. An example of an interpretation-based system is the Cyclops Tensor Framework (CTF) [11], [12].

Library approaches, such as PETSc and Trilinos, implement a predefined set of operations, each with a fixed data format and distribution strategy. These systems provide bare-metal performance but are inflexible in the face of three sources of variability. First, the fixed set of implementations in a library means that the implementation of some the countably infinite set of tensor algebra expressions using these building blocks will be suboptimal, and in practice there are important computations that incur such a performance penalty. Second, when users have a data distribution or data format not directly supported by the system, they must reshape their data to fit the interface, incurring significant cost. Finally, the library approach is difficult to adapt to new hardware, as each kernel must be rewritten and re-tuned for each new platform.

Interpreted approaches, such as CTF, execute a tensor algebra expression using a series of distributed matrix multiplication and transposition operations. This approach can implement all tensor algebra expressions, but cannot achieve optimal performance for all expressions. Our experiments show that the interpreted approach can be one to two orders of magnitude slower than hand-tuned implementations due to unnecessary data reorganization and communication. Through compilation, SpDISTAL achieves the benefits of both approaches, supporting the full generality of sparse tensor algebra but also specializing implementations to the desired computation and data layouts.

The recent work of DISTAL [13] introduced separate scheduling and data distribution languages for distributed dense tensor algebra compilation, making it capable of expressing a large variety of dense tensor algebra algorithms. However, DISTAL has no notion of sparsity in its language or implementation.

We address adding sparsity into DISTAL's programming model in a way that is separated from but also composes well with the scheduling and data distribution languages, allowing SpDISTAL to express a wide range of distributed sparse tensor algebra algorithms. We show how to add sparsity to DISTAL's scheduling and data distribution languages in a way that is expressive and composable. We then show how to leverage *dependent partitioning* [14] to translate tensor algebra expressions and data distribution declarations with sparsity specifications into assignments of specific sub-tensors to a distributed machine.

Figure 1 demonstrates how sparsity is integrated into DISTAL's programming model in an encapsulated manner. Line 16 declares that the matrix in a matrix-vector multiplication is sparse. Apart from the sparsity annotation, the remainder of Figure 1 is a valid DISTAL program that implements a row-wise distributed matrix-vector multiplication. SpDISTAL allows

```
1  // Declare input parameters for generated code.
2  Param pieces, n, m;
3  // Define the machine M as a 1D grid of processors.
4  Machine M(Grid(pieces));
5
6  // Define the data structure and distribution for
7  // each tensor. We define two dense vector formats,
8  // one blocked onto M, the other replicated onto all
9  // processors in M. Finally, we define a CSR matrix
10 // format, distributed row-wise. The format notation
11 // is discussed in Subsection II-B.
12 DistVar x, y;
13 Format BlockedDense({Dense}, Distribution({x}, M, {x}));
14 Format ReplDense({Dense}, Distribution({x}, M, {y}));
15 Format BlockedCSR({Dense, Compressed},
16                   Distribution({x, y}, M, {x}));
17
18 // Create our tensors, using the defined formats. Our
19 // SpMV algorithm will block a and B, and replicate c.
20 Tensor<double> a({n}, BlockedDense);
21 Tensor<double> B({n, m}, BlockedCSR);
22 Tensor<double> c({m}, ReplDense);
23
24 // Declare the computation, a matrix-vector multiply.
25 IndexVar i, j;
26 a(i) = B(i, j) * c(j);
27
28 // Map the computation onto M via scheduling commands.
29 IndexVar io, ii;
30 a.schedule()
31  // Block i for each node.
32  .divide(i, io, ii, M.x)
33  // Distribute each block of i onto each node.
34  .distribute(io)
35  // Communicate the needed sub-tensor for each chunk of i.
36  .communicate({a, B, c}, io)
37  // Schedule the leaf computation that runs on each node.
38  // Here, we parallelize chunks of i over CPU threads.
39  .parallelize(ii, CPUThread);
```

Figure 1: Distributed CPU SpMV kernel in SpDISTAL.

for independent description of the sparsity structure of tensors to yield distributed sparse tensor computations.

We implement SpDISTAL by extending the DISTAL [13] and TACO [15] compilers and targeting the Legion [16] distributed runtime system. SpDISTAL implements sparsity in the model through a combination of compilation techniques and runtime analysis, avoiding the need to perform complex control and data flow analyses over imperative code.

The specific contributions of this work are:

1) A programming model that separates data distribution and computation distribution for sparse tensors,
2) a data structure specification language separated from data distribution specifications, and
3) compilation techniques to support distribution of sparse tensor algebra computations.

We evaluate SpDISTAL by comparing against the state-of-the-art distributed sparse linear and tensor algebra libraries PETSc, Trilinos and CTF on matrices and tensors from the SuiteSparse [17], FROSTT [18] and Freebase [19] datasets. We find that SpDISTAL is competitive with PETSc and Trilinos, and can outperform CTF by over an order of magnitude.

II. Programming Model

The first contribution of SpDISTAL is a programming model that combines sparse data structure specifications with separate data and computation distribution languages. Figure 1 utilizes the three input sub-languages of SpDISTAL to implement a distributed SpMV: a *computation language* that describes the desired kernel (line 26), a *format language* (lines 12-22) that describes how input tensors store non-zeros and are distributed onto a machine, and a *scheduling language* (lines 30-39) that describes how to optimize and distribute the computation. Components of these input languages have been proposed by prior works, as discussed in the next few sections. However, the first key contribution of SpDISTAL is the novel combination of these languages to support distributed sparse tensor algebra.

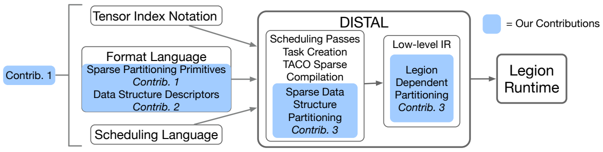


Figure 2: Overview of SpDISTAL's contributions.

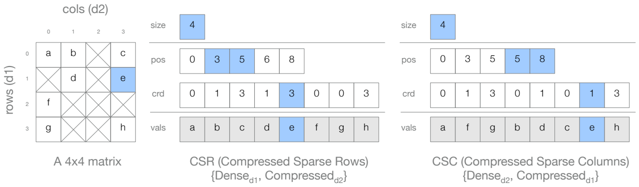


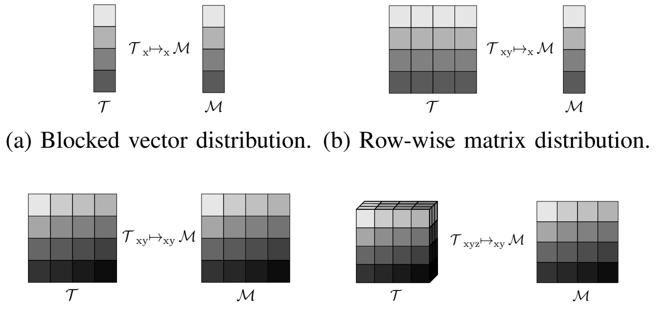
Figure 3: Different encodings of a sparse matrix in TACO's format language. Blue squares indicate how the coordinate (1,3) is represented by each encoding.

A. Computation Language

Computation is described in SpDISTAL using *tensor index notation* (TIN). We adopt the concrete syntax of TACO [15] and DISTAL [13] for TIN. TIN consists of *accesses* that index tensor dimensions with lists of *index variables*. TIN statements are assignments into a left-hand side access, while the right hand side is an expression constructed from multiplication and additions between any number of accesses. For example, the tensor-times-vector operation is expressed in TIN as $A(i, j) = B(i, j, k) \cdot c(k)$, declaring that each element $A(i, j)$ is the inner product between the final dimension of B and the vector c. Intuitively, each distinct index variable corresponds to a loop, and variables contained only in the right-hand side of the assignment represent sum-reductions over their domain.

B. Format Language

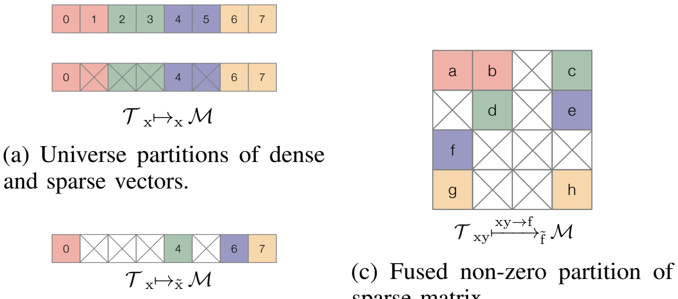
Sparse Data Structures. SpDISTAL adopts the format language of TACO [15], letting users control tensor data structures and thus how zero entries are compressed. TACO's format language allows users to specify the format used to store each dimension of a sparse tensor. Two formats considered in TACO are the `Dense` and `Compressed` formats. A k-dimensional tensor is stored using any combination of k instances of these formats. A `Dense` format represents a standard array containing all coordinates of the dimension. A `Compressed` format encodes only the non-zero coordinates of the dimension using two arrays: a `crd` array that stores the non-zero coordinate values and a `pos` array that stores the range of coordinates associated with each entry in the previous dimension. Figure 3 shows how TACO's per-dimension approach allows for the expression of


$\mathcal{T}_{\text{x}} \mapsto_{\text{x}} \mathcal{M}$
$\mathcal{T}$
$\mathcal{M}$
$\mathcal{T}_{\text{xy}} \mapsto_{\text{x}} \mathcal{M}$
$\mathcal{T}$
$\mathcal{M}$
$\mathcal{T}_{\text{xy}} \mapsto_{\text{xy}} \mathcal{M}$
$\mathcal{T}$
$\mathcal{M}$
$\mathcal{T}_{\text{xyz}} \mapsto_{\text{xy}} \mathcal{M}$
$\mathcal{T}$
$\mathcal{M}$

(a) Blocked vector distribution. (b) Row-wise matrix distribution.

(c) Tiled matrix distribution. (d) Map 3-tensor onto proc. grid.

Figure 4: Examples of tensor distribution notation statements.[1]


0
1
2
3
4
5
6
7
0
4
6
7
$\mathcal{T}_{\text{x}} \mapsto_{\text{x}} \mathcal{M}$
0
4
6
7
$\mathcal{T}_{\text{x}} \mapsto_{\tilde{\text{x}}} \mathcal{M}$
a
b
c
d
e
f
g
h
$\mathcal{T}_{\text{xy}} \overset{\text{xy}\to\text{f}}{\longmapsto}_{\tilde{\text{f}}} \mathcal{M}$

(a) Universe partitions of dense and sparse vectors.

(b) Non-zero partition of a sparse vector.

(c) Fused non-zero partition of sparse matrix.

Figure 5: Sparse tensors distributed with TDN. Each color indicates a group of coordinates mapped to the same processor.

a variety of different common formats. For example, the CSR matrix encoding (shown in the center) is constructed by using a `Dense` encoding of the first dimension and a `Compressed` encoding of the second dimension. The CSC matrix encoding (shown on the right) instead uses a `Dense` encoding of the second dimension and a `Compressed` encoding of the first dimension, and orders the dimensions in reverse.

Data Distribution. SpDISTAL extends the *tensor distribution notation* (TDN) language developed in DISTAL [13] with new constructs for describing distributions of sparse tensors. TDN lets users specify how each dimension of a tensor is partitioned onto different dimensions of an abstract machine represented by an n-dimensional grid. A TDN statement assigns names to each dimension of a tensor and a machine, and tensor dimensions that share a name with a machine dimension are partitioned by the corresponding machine dimension. For example, the TDN statement $\mathcal{T}_{\text{xy}} \mapsto_{\text{x}} \mathcal{M}$ maps a matrix $\mathcal{T}$ row-wise onto a one dimensional machine $\mathcal{M}$, declaring that the first dimension of $\mathcal{T}$ is partitioned by the first dimension of $\mathcal{M}$. We include four examples of TDN statements for dense tensors in Figure 4.

In SpDISTAL, we extend TDN with *universe* and *non-zero* partitions, and *coordinate fusion*. TDN's default partition is a *universe partition*: when a tensor dimension d is partitioned by a machine dimension, the range of coordinates of d (i.e., the universe[2] $\mathbb{U}$) is partitioned equally among processors in that machine dimension. As seen in Figure 5a, universe partitions can be applied to sparse tensor dimensions, partitioning the non-zero coordinates according to the equal partition of $\mathbb{U}$.

Universe partitions of sparse tensor dimensions may result in imbalance, as the non-zero coordinates may not align with the universe partitions. We therefore introduce *non-zero partitions*, which declare that the non-zero coordinates of d should be partitioned evenly using the tilde operator $\tilde{d}$ on a machine dimension. For example, the TDN statement $\mathcal{T}_{\text{x}} \mapsto_{\tilde{\text{x}}} \mathcal{M}$ declares that the non-zeros of a sparse vector should be distributed equally onto $\mathcal{M}$ and is visualized in Figure 5b.

In isolation, non-zero partitions are not sufficient to evenly distribute higher order tensors. For example, a matrix with no empty rows but a varying number of non-zeros per row has the same distribution under a universe and non-zero partition of the first dimension. We therefore introduce *coordinate fusion*, allowing for operations such as evenly distributing the non-zero coordinates of a sparse matrix. Coordinate fusion collapses multiple dimensions into a single logical dimension, which can be the target of a non-zero partition. The syntax $\mathcal{T}_{\text{xy}} \overset{\text{xy}\to\text{f}}{\longmapsto}_{\tilde{\text{f}}} \mathcal{M}$ utilizes coordinate fusion to equally distribute the non-zeros of a matrix by flattening x and y into a new coordinate f and then performing a non-zero partition of f, visualized in Figure 5c. The combination of fusion and non-zero partitioning allows for expression of a variety of non-trivial data partitioning strategies. For example, the three distributions $\mathcal{T}_{\text{xyz}} \mapsto_{\tilde{\text{x}}} \mathcal{M}$, $\mathcal{T}_{\text{xyz}} \overset{\text{xy}\to\text{f}}{\longmapsto}_{\tilde{\text{f}}} \mathcal{M}$, and $\mathcal{T}_{\text{xyz}} \overset{\text{xyz}\to\text{f}}{\longmapsto}_{\tilde{\text{f}}} \mathcal{M}$ map a 3-tensor onto a machine by equally distributing the non-zero slices, the non-zero tubes and non-zero values respectively. Fusion and non-zero partitions do not subsume universe partitions—Subsection II-D describes tradeoffs between the gained load balance and additional communication.

C. Scheduling Language

Like many domain specific languages [20]–[26], SpDISTAL separates the computation description from how the computation should be executed through a *scheduling language* that describes optimizing transformations. Common transformations introduced by prior work that SpDISTAL uses include `parallelize` (parallelize loop iterations), `precompute` (hoist computation out of a loop), `split/divide` (break a loop into two nested loops), `fuse` (collapse two loops into a single loop), and `reorder` (change the order of loops).

Though these common loop transformations have generally applied to dense loop nests, Senanayake et al. [20] show how to extend TACO to support these transformations on sparse iteration spaces on a single node. To perform additional optimizations on sparse iteration spaces, Senanayake et al. also introduce variants of the `split` and `divide` transformations that enable the iterations over only non-zero values to be strip-mined into equal pieces. These variants compose with the `fuse` and `parallelize` commands to enable statically load balanced iteration over certain sparse loops. For example, Senanayake et al. show how to implement an SpMV that is load-balanced over CPU threads by `fuse`-ing the i and j dimensions of the computation, applying the non-zero based variant of `split`,

[1] Figures used with author permission from Yadav et al. [13]

[2] We refer to the set of all coordinates of a tensor dimension as the universe, from which a sparse tensor may store only a subset.

and then `parallelize`-ing the resulting outer loop. We refer to Sections 2 and 3.3 of Senanayake et al. [20] for more details.

Finally, DISTAL [13] introduced new scheduling commands to target distributed machines, namely `distribute` and `communicate`. `distribute` directs that iterations of the target loop should execute on different processors. The `communicate` command directs that the necessary subsets of each target tensor should be fetched to a local memory at the beginning of each iteration of the target loop. The `communicate` command is only used for optimization—users control the granularity of communication for performance and DISTAL infers what data to communicate and the source and destination of transfers.

SpDISTAL's scheduling language combines the single-node sparse iteration space transformations in TACO [20] with DISTAL's distributed scheduling transformations [13], enabling the distribution of sparse tensor programs. This combination of scheduling transformations is novel and unique to SpDISTAL.

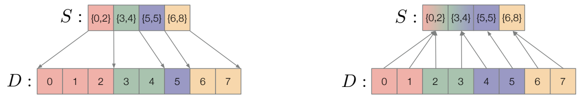
D. Putting It Together: Scheduling SpMV

To showcase the SpDISTAL programming model, we discuss two algorithms for the SpMV kernel, $a(i) = B(i,j) \cdot c(j)$ that target a machine $\mathcal{M}$ organized as one-dimensional grid of processors. The first uses a row-based distribution, implemented using SpDISTAL's C++ API in Figure 1, and the second uses a non-zero-based distribution of the computation. We use the same sparse formats for each tensor in both distributed algorithms: `a, c = {Dense}` and `B = {Dense, Compressed}`.

In the row-based algorithm, each processor is assigned rows of B, the corresponding elements of a, and all of c. To avoid data movement at compute time, we declare the data distributions of each tensor as follows: $a_{\,\mathrm{x}}{\mapsto}_{\mathrm{x}}\,\mathcal{M}$, $B_{\,\mathrm{xy}}{\mapsto}_{\mathrm{x}}\,\mathcal{M}$ and $c_{\,\mathrm{x}}{\mapsto}_{\mathrm{y}}\,\mathcal{M}$. To schedule this algorithm, we `divide` to create a group of rows for each processor, and `distribute` the groups over each processor. We then use `communicate` on `a`, `B` and `c` at the distributed loop, pre-fetching each sub-tensor at the start of processing each group of rows.

When the rows of B have different numbers of non-zeros, a row-based algorithm can suffer from load imbalance. An algorithm that evenly splits the non-zeros, on the other hand, enables perfect load balance at the cost of communication to reduce into the output. This algorithm partitions the elements of a (not necessarily evenly) and replicates c. We choose a non-zero based distribution of B using coordinate fusion and a non-zero partition with $B_{\,\mathrm{xy}}\xmapsto{xy \to f}_{\tilde{\mathrm{f}}}\,\mathcal{M}$. To schedule the computation we `fuse` the `i` and `j` loops, then `divide` the space of non-zero coordinates of B. Finally, we `distribute` and `communicate` as before to complete the schedule.

In both algorithms, we matched the data and computation distributions to avoid unnecessary communication. However, this is not required—a SpDISTAL program that utilized the row-based schedule but the non-zero based data distribution is valid but comes at a performance cost, as extra communication operations would be needed to reshape the non-zero based data distribution into a row-based one.



(a) Image

(b) Preimage

Figure 6: Visualization of image and preimage operations.

III. Distributed Sparse Tensor Data Structures

The first step to realizing the programming model discussed in Section II is to define the data structures used to represent distributed sparse tensors. In this section, we describe abstract distributed data types on which we lay out distributed sparse tensors. Then, in Section IV, we describe how to generate code that distributes and computes on these sparse tensors. These abstract data structures are then implemented by a runtime system (Legion [16]) through dynamic analyses.

A. Abstract Distributed Data Structures

To manage the complexity of distributing sparse tensors, we first reduce them to abstract data types that more directly lend themselves to distribution. These abstract data types come from the Legion ecosystem [14], [16].

An *index space* is an abstract object representing a set of indices or coordinates, where the indices can have any number of dimensions. A *region* is a multi-dimensional array of values, where the values can be primitive types such as integers or floats, or structured data such as index spaces that name sets of indices in other regions. As a multi-dimensional array, a region can be viewed as a function from indices in a multi-dimensional index space to a set of values. Therefore, each region is associated with an index space that describes the valid set of indices that can be used to access the region.

A *partition* is an abstract object representing a mapping from a set of *colors* to (potentially overlapping) subsets of an index space. We depict partitions by shading subsets of regions different colors. Regions can be *partitioned* into *sub-regions* by constructing a partition of the region's associated index space. Each sub-region is associated with the corresponding subset of the original partitioned index space. Regions are distributed by partitioning them into sub-regions that are placed onto different memories across a distributed machine.

Partitions are created either through direct coloring of subsets of indices, or from existing partitions through a class of operations known as *dependent partitioning* [14] operations. We utilize two dependent partitions operations *image* and *preimage* that are applicable to regions containing index spaces as values. Regions with index spaces as values encode pointers to indices of other regions. For example, in Figure 6a, the first element of the top region is an index space corresponding to the set $\{0, 1, 2\}$, pointing to the first three elements of the bottom region. Intuitively, image colors all destinations of pointers with the same color as their source, while preimage colors all sources of pointers with the same color as their destination.

We now give precise definitions of image and preimage. Consider a source region S containing index spaces that name indices in a destination region D. Given a partition

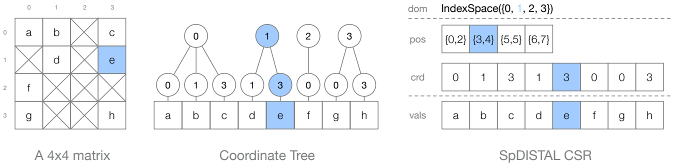


Figure 7: Coordinate tree and SpDISTAL CSR encodings.

P_S of S, `image`(S, P_S, D) is a partition P' of D such that $\forall c \in P_S, \forall i \in P_S[c], S[i] \subseteq P'[c]$, depicted in Figure 6a. Preimage performs the inverse operation: given a partition P_d of D, `preimage`(S, P_D, D) is a partition P' of S such that $\forall c \in P_D, \forall i \in P_D[c], \forall i'$ s.t. $i \in S[i'], i' \in P'[c]$, depicted in Figure 6b. We reiterate that partitions can contain overlapping subsets of index spaces. As seen in Figure 6b, the resulting partition of S colors some indices with multiple colors. Data referenced by multiple sub-regions in a partition is shared across the different memories that sub-regions are placed in, and the runtime system manages coherence between the different copies.

B. Distributed Sparse Tensors

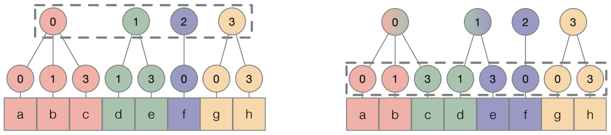
Having described abstract data structures that represent distributed arrays and operations to create expressive partitions of these arrays, we show how to use them to represent sparse tensors. Our distributed sparse tensor encoding is a distributed extension of the encoding proposed by TACO [15] and discussed in Subsection II-B. To give intuition about our (and TACO's) sparse tensor encoding, we describe the *coordinate tree* representation of a tensor.

A tensor $\mathcal{T}$'s coordinate tree is a tree with one level per dimension of $\mathcal{T}$ (in addition to a root), as shown in the center of Figure 7. Each path from the root to a leaf node represents a non-zero tensor coordinate. The levels of the coordinate tree are ordered in the way the dimensions of $\mathcal{T}$ are stored. For example, a row-major layout of a matrix would have rows followed by columns, while a column-major layout would have columns followed by rows. Each level of the coordinate tree is represented with a *level format*, which encodes the non-zero coordinates of the level with different data structures.

Sparse tensors are encoded by specifying how each level of the coordinate tree is stored independently of the other levels. Each coordinate tree level is represented by a *level format* that defines how the non-zero coordinates of the coordinate tree level are represented with different physical data structures. In this paper, we consider two level formats: the `Dense` level format that stores all coordinates of a level in a dense array, and the `Compressed` level format that encodes the non-zero coordinates of a level by compressing away the zero coordinates.

The `Dense` level format is used to encode coordinate tree levels that contain few zero values. We encode `Dense` levels with an index space over the range from zero to the size of the dense level, named `dom`. Multiple contiguous `Dense` levels are collapsed into a single logical multi-dimensional `Dense` level that is represented by a multi-dimensional index space.

`Compressed` levels are encoded in TACO with an array of coordinates (`crd`) and an array that stores bounds on the range of coordinates associated with each entry in the parent level (`pos`). The coordinates for an entry i in the parent level are stored in the positions of `crd` within the range `[pos[i], pos[i+1])`. SpDISTAL stores the `pos` and `crd` arrays as regions to enable partitioning and distribution. As the `pos` region contains ranges of the `crd` region, our goal is to utilize image and preimage to relate partitions of `pos` and `crd`. Therefore, we store tuples in the `pos` region that contain the lower and upper bounds of coordinate positions, representing a set of indices in the `crd` region. The coordinates for an entry i in the parent level are stored in positions `[pos[i].lo, pos[i].hi]` of `crd`. Figure 7 visualizes an SpDISTAL CSR matrix.



(a) Universe partition of level one. (b) Non-zero partition of level two.

Figure 8: Coordinate tree partitions derived from a level partition. Dashed grey lines indicate the initial level partition.

IV. Compiling Distributed Sparse Tensor Programs

Having described the data structures that represent sparse tensors, we now show how to generate distributed sparse tensor algebra code. Since we can utilize single node code generation for sparse tensor algebra ([15], [20]), the key challenge is partitioning sparse tensors into sub-tensors that can be processed on a single node. Describing partitions of sparse tensors enables moving sparse tensors into a specified distribution or to where they are needed by a computation. Our approach has two components: 1) a set of compiler abstractions to reason about data partitioning, and 2) a code generation algorithm that uses these abstractions to generate partitioning code specialized to a computation or data distribution specification. We first provide the intuition behind our approach, and then describe the novel compiler abstractions and code generation process.

A. Intuition

To build intuition for our approach, we appeal to the coordinate tree representation of a sparse tensor. Consider the row-based and non-zero-based SpMV algorithms discussed in Subsection II-D. The distributed loops correspond to partitions of the matrix's coordinate tree. The first strategy distributes the rows, corresponding to a partition of the first coordinate tree level (Figure 8a). The second strategy distributes the non-zero coordinates, corresponding to a partition of the second coordinate tree level (Figure 8b). In the row-based strategy, each coordinate in the first level will access all child coordinates in the second level of the tree. In the non-zero based strategy, each coordinate in the second level needs to access its parent coordinate in the first level of the tree.

Figure 8a and Figure 8b depict the full coordinate tree partitions for each strategy. Given a partition of a coordinate tree level, we obtain a partition of the child level by applying the partition to the children of each node in the level. Conversely, we obtain a partition of the parent level of a partitioned level

Level Type	Level Function Definitions	

Dense	initUniversePartition(): return ‘C = {}’
	createUniversePartitionEntry(color, bounds):
	  return ‘C[color] = bounds’
	finalizeUniversePartition():
	  return ‘P = partitionByBounds(C, dom)’, ‘P’, ‘P’
		initNonZeroPartition(): return ‘C = {}’
		createNonZeroPartitionEntry(color, bounds):
		  return ‘C[color] = bounds’
		finalizeNonZeroPartition():
		  return ‘P = partitionByBounds(C, dom)’, ‘P’, ‘P’

	partitionFromParent(parentPart):
	  return ‘part = copy(parentPart)’, ‘part’
		partitionFromChild(childPart):
		  return ‘part = copy(childPart)’, ‘part’

Compressed	initUniversePartition(): return ‘C_crd = {}’
	createUniversePartitionEntry(color, bounds):
	  return ‘C_crd[color] = bounds’
	finalizeUniversePartition():
	  return ‘P_crd = partitionByValueRanges(C_crd, crd)
	          P_pos = preimage(pos, P_crd, crd)’,
	          ‘P_pos’, ‘P_crd’
		initNonZeroPartition(): return ‘C_crd = {}’
		createNonZeroPartitionEntry(color, coordBounds):
		  return ‘C_crd[color] = coordBounds’
		finalizeNonZeroPartition():
		  return ‘P_crd = partitionByBounds(C_crd, crd)
		          P_pos = preimage(pos, P_crd, crd)’,
		          ‘P_pos’, ‘P_crd’

	partitionFromParent(parentPart):
	  return ‘P_pos = copy(parentPart)
	          P_crd = image(pos, P_pos, crd)’, ‘P_crd’
		partitionFromChild(childPart):
		  return ‘P_crd = copy(childPart)
		          P_pos = preimage(pos, P_crd, crd)’, ‘P_pos’

Table I: Partitioning level function definitions for `Dense` and `Compressed` levels. Ticks indicate IR fragments.

by partitioning the parent level such that each parent node is colored with all of its children's colors. The resulting partition of the coordinate tree may assign a node in a multiple colors. For example, in Figure 8b, the coordinate 0 in the first level is needed by nodes colored red and green in the second level.

This intuition yields a high level code generation strategy: First, we create an initial partition of a level of each tensors' coordinate tree based on the data or computation distribution directives. Then, we use the initial level partition to create partitions of all levels above and below the initial level.

B. Format Abstractions for Sparse Tensor Partitioning

To realize the intuitive algorithm on coordinate trees, we must translate the partitioning operations on coordinate tree levels to partitioning operations on the abstract data structures that encode distributed sparse tensors. In SpDISTAL, each tensor dimension is encoded by a level format that encodes how a coordinate tree level is stored in memory. Chou et al. [27] proposed a compile-time interface for level formats that provides an abstraction for a code generation algorithm to target. The abstraction allows for per-dimension reasoning about sparse tensors and for new formats implementing the interface to be added without changing the code generation algorithm. The format abstraction contains *level functions* that return intermediate representation (IR) fragments for the code generator to manipulate. In this section, we introduce new level functions that correspond to the two intuitive phases of coordinate tree partitioning: 1) initially partitioning a coordinate tree level and 2) deriving a partition of the full coordinate tree. Then, in Subsection IV-C we show how a code generation algorithm can utilize these abstractions to generate specialized partitioning code.

There are two groups of functions for creating initial level partitions, corresponding to universe and non-zero partitions. The universe partition group consists of three functions:

```
initUniversePartition() -> IRStmt
createUniversePartitionEntry(color,bounds) -> IRStmt
finalizeUniversePartition() -> (IRStmt,partition,partition)
```

`initUniversePartition` initializes any necessary data structures for partitioning. `createUniversePartitionEntry` takes symbolic values of a color and a coordinate range that should be assigned to the color and maps the range of coordinates to the color. `finalizeUniversePartition` finalizes any data structures and returns a partition to use for partitioning parent levels and a partition to use for partitioning child levels.

The non-zero partition group also consists of three functions:

```
initNonZeroPartition() -> IRStmt
createNonZeroPartitionEntry(color,bounds) -> IRStmt
finalizeNonZeroPartition() -> (IRStmt,partition,partition)
```

`initNonZeroPartition` and `finalizeNonZeroPartition` are the same as their universe partition counterparts. `createNonZeroPartitionEntry` is similar to `createUniversePartitionEntry`, but takes bounds on the positions within the level that encode non-zero coordinates.

There are two functions for constructing derived partitions:

```
partitionFromParent(partition) -> (IRStmt,partition)
partitionFromChild(partition) -> (IRStmt,partition)
```

Each function partitions a level using an existing partition, and returns a partition to use to partitioning child or parent levels.

We show implementations of the partition level functions for `Dense` and `Compressed` levels in Table I. The initial partitioning functions for `Dense` levels color the coordinate space that the `Dense` level represents, and the derived partitioning functions apply input partitions to the coordinate space. For `Compressed` levels, the universe partitioning functions partition the `crd` region by bucketing the coordinates into the ranges demarcated by `createUniversePartitionEntry`, while the non-zero partitioning functions partition the `crd` region according to the position bounds demarcated by `createNonZeroPartitionEntry`. Both use a preimage to recover a partition of the `pos` region. The derived partitioning functions partition the `pos` and `crd` regions from parent and child partitions, and use image and preimage to create partitions of the other region in the level.

C. Code Generation Algorithm

We describe how the format abstraction functions for partitioning are used in an algorithm to generate code that partitions sparse tensors. This algorithm implements the intuitive strategy described in Subsection IV-A by making calls to the format abstraction functions to generate code that partitions each level of a tensor. Our algorithm encodes information about the relationships between tensor levels through creating partitions, and discharges the data movement operations required to materialize these partitions to a runtime system.

The algorithm is a recursive function called on a scheduled TIN statement and recurses on index variables in the scheduled order. For each distributed index variable, the algorithm generates code to create initial level partitions of tensors, and then code to derive partitions of full coordinate trees. Pseudocode for the algorithm is in Figure 9a. Figure 9b contains generated code for a row-based SpMV schedule. Figure 9c and Figure 9d depict partitions created by our algorithm for the row-based and non-zero-based SpMV schedules.

During code generation, TACO breaks iteration over sparse data structures into two kinds: *coordinate value* iteration and *coordinate position* iteration. Coordinate value loops iterate over all possible coordinate values and compute or retrieve present coordinates from tensor levels. Coordinate position loops instead iterate over the non-zero coordinates in a level by directly iterating over the positions that hold non-zero coordinates in a level. Coordinate position loops arise when applied scheduling transformations strip-mine iterations over non-zero coordinates only, as discussed in Subsection II-C. Our code generation algorithm follows these two cases: distributed coordinate value loops distribute the space of coordinates, corresponding to universe partitions, while distributed coordinate position loops distribute the space of non-zero coordinates, corresponding to non-zero partitions.

To generate partitioning code for coordinate value loops, the algorithm creates initial universe partitions of accessed tensor levels by calling `createInitialUniversePartitions`, which proceeds in three steps to generate the code labeled (1) in Figure 9b. First, it invokes `initUniversePartition` for each tensor accessed by the current index variable. Next, it emits a `for` loop over the current index variable, and infers symbolic upper and lower bounds on the index variable to pass to an invocation of `createUniversePartitionEntry` for each tensor. Lastly, it calls `finalizeUniversePartition` on each tensor. The algorithm then partitions each coordinate tree using `partitionCoordinateTrees` partitions each level above the initial level with `partitionFromChild` and partitions each level below the initial level with `partitionFromParent`. The result of `partitionCoordinateTrees` is demarcated with label (2) in Figure 9b.

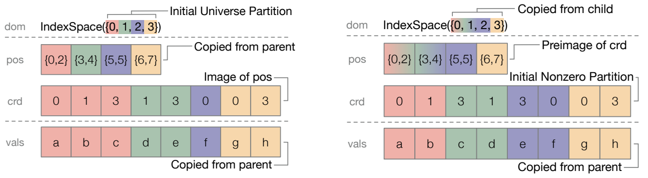
The partitioning process for coordinate position loops is similar to the process for partitioning coordinate value loops. The algorithm calls `createInitialNonZeroPartition`, which constructs an initial level partition of the position space tensor by invoking `initNonZeroPartition`, generating a loop with the result of `createNonZeroPartitionEntry` and completing the level partition with `finalizeNonZeroPartition`. Next, the algorithm partitions the full coordinate tree of the position space tensor as previously with `partitionNonZeroCoordinateTree`. Finally, the algorithm uses a universe partition derived from the top-level partition of the position space tensor's coordinate tree to use as an initial level partition for all other tensors in the statement (`partitionRemainingCoordinateTrees`).

```
def codegen(TINStatement s, IndexVar i):
  if not distributed(i):
    # Fall back to standard TACO code generation.
    codegenTACO(s, i)
    return

  if coordinateValueIteration(s, i):
    # Create initial partitions of each tensor.
    createInitialUniversePartitions(i, s)
    # Derive full coordinate tree partitions.
    partitionCoordinateTrees(i, s)
  else:
    # Create initial partition of non-zero split tensor.
    createInitialNonZeroPartition(i, s)
    # Partition the full coordinate tree of the
    # non-zero split tensor.
    partitionNonZeroCoordinateTree(i, s)
    # Using the partition of the non-zero split tensor,
    # partition all other accessed tensors.
    partitionRemainingCoordinateTrees(i, s)

  # Emit a distributed for loop over the index variable
  # and pass the partitions to each iteration.
  emitDistributedForLoop(i)
  # Codegen the next index variable as the loop body.
  codegen(s, next(i))
```

(a) Code generation algorithm.

```
void SpMV(Tensor a, Tensor B, Tensor c, int pieces) {
  // B1.initUniversePartition()
  Coloring BColoring = {};
  for (int io = 0; io < pieces; io++) {
    int iLo = io * (B[0].dim / pieces);
    int iHi = (io + 1) * (B[0].dim / pieces);
    // B1.createUniversePartitionEntry(io, {iLo, iHi})     (1)
    BColoring[io] = {iLo, iHi - 1}; }
  // B1.finalizeUniversePartition()
  auto B1Part = partitionByBounds(B[0].dom, BColoring);
  // B2.partitionFromParent(B1Part)
  auto B2PosPart = copy(B1Part, B[1].pos);
  auto B2CrdPart = image(B2PosPart, B[1].pos);               (2)
  auto BValsPart = copy(B2CrdPart, B.vals);
  // Execute each iteration on a different node.
  distributed for io in {0 ... pieces} {
    B = Tensor({B1Part[io],                                  (3)
         {B2PosPart[io], B2CrdPart[io]},
          BValsPart[io]});
    for (int ii = 0; ii < (B[0].dim / pieces); ii++) {
      int i = io * (B[0].dim / pieces) + ii;
      for (int jB = B[1].pos[i].lo;
               jB <= B[1].pos[i].hi; jB++) {                 (4)
        int j = B[1].crd[jB];
        a.vals[i] += B.vals[jB] * c.vals[j];
      }}}}
```

(b) Generated pseudo-code for a row-based SpMV. For simplicity, partitioning of a and c, iteration guards and bounds checks are omitted.



(c) Generated universe partition for SpMV.

(d) Generated non-zero partition for SpMV.

Figure 9: Code generation algorithm and examples of output.

After creating the necessary partitions of each tensor in the input statement, the algorithm emits a distributed for loop over the current index variable and passes the corresponding sub-region of each partition to each distributed loop iteration (`emitDistributedForLoop`), as seen in label (3) in Figure 9b.

The body of the loop is generated by a recursive call to the code generation algorithm (label (4) in Figure 9b).

V. Implementation

We implement SpDISTAL by extending DISTAL [13], which targets the Legion [16] runtime system, and use techniques from TACO [15] to generate sparse code for CPUs and GPUs.

A. Partitioning

To implement the partitioning strategy described in Section IV on the irregular data structures that store sparse tensors, SpDISTAL utilizes the *dependent partitioning* [14] infrastructure of Legion. We implement the index space, partition and region types discussed in Section III with Legion's `IndexSpace`, `Region` and `LogicalPartition` types. Legion supports distributed image and preimage operations, allowing for partitioning of sparse tensors without moving all program data into a centralized location. Legion manages moving the subregions defined by SpDISTAL's partitions between memories in the target machine through runtime analysis.

B. Sparse Output Tensors

Our prototype implementation of SpDISTAL has support for some cases of statements and output sparsity formats. Some tensor index notation statements (such as sparse tensor-times-vector $\mathbf{A}(i,j) = \mathbf{B}(i,j,k) \cdot c(k)$) preserve the sparsity pattern of the input tensor in the output tensor. SpDISTAL identifies situations where this is possible and emits code that copies the coordinate metadata from the output tensor into the input tensor and modifies the values of the output tensor only. For cases where the sparsity pattern of the output is unknown, we implement the two-phase parallel assembly approach described by Chou et al. [28]. SpDISTAL generates code that first symbolically executes the desired computation and records what locations non-zero outputs should be written into, and then uses the results of the symbolic execution to construct the output tensor without additional synchronization.

C. Tensor Distribution Notation

We follow DISTAL's [13] approach to implement tensor distribution notation. DISTAL translates a TDN statement into a scheduled TIN statement, using `divide` and `distribute` to partition the tensor according to the TDN statement. We extend this approach with coordinate fusion using `fuse`, and support non-zero partitioning by using the version of `divide` that strip-mines the non-zero coordinates. The resulting TIN statement is then compiled using the algorithm described in Figure 9a.

VI. Evaluation

Experimental Setup. We ran our experiments on the Lassen supercomputer [29]. Each Lassen node has a 40 core dual socket IBM Power9, four NVIDIA Volta V100s connected by NVLink 2.0 and an Infiniband EDR interconnect. All systems were compiled with GCC 8.3.1 and CUDA 11.1. Legion[3] was configured with GASNet-EX 2021.3.0 for communication.

Tensor name	Domain	Non-zeros
arabic-2005	Web Connectivity	6.39×10^8
it-2004	Web Connectivity	1.15×10^9
kmer_A2a	Protein Structure	3.60×10^8
kmer_V1r	Protein Structure	4.65×10^8
mycielskian19	Synthetic	9.03×10^8
nlpkkt240	PDE's	7.60×10^8
sk-2005	Web Connectivity	1.94×10^9
twitter7	Social Network	1.46×10^9
uk-2005	Web Connectivity	9.36×10^8
webbase-2001	Web Connectivity	1.01×10^9
freebase_music	Data Mining	1.74×10^9
freebase_sampled	Data Mining	9.95×10^7
nell-2	NLP	7.68×10^7
patents	Data Mining	3.59×10^9

Table II: Tensors and matrices considered in our experiments. The first group is matrices from SuiteSparse [17]. The second group are 3-tensors, where tensors prefixed with "freebase" are from the Freebase [19] dataset, and remaining tensors are from the FROSTT [18] dataset.

Comparison Targets. We compare against PETSc[4], the TPetra [10] package of Trilinos[5] and Cyclops Tensor Framework (CTF)[6]. Trilinos and CTF were configured with OpenMP. Trilinos and PETSc were built with CUDA support.

Dataset. We consider 14 real-world matrices and tensors from the SuiteSparse [17], FROSTT [18] and Freebase [19] datasets as described in Table II. For SuiteSparse, we chose the largest matrices that were representable in PETSc's and Trilinos's default configuration (32-bit indexing). For FROSTT, we chose all 3-tensors that satisfy the CTF limitation that tensor dimensions must multiply to less than the maximum 64 bit integer. For operations with multiple sparse inputs, we follow Henry and Hsu et al. [30] by shifting the last dimension of each tensor to construct additional sparse inputs.

Experimental Methodology. All experiments were run with 10 warm-up trials, 20 timed trials and a 90 minute timeout. For all tensors other than "patents" we use a format with a `Dense` outer level and all other levels `Compressed`; for "patents" we use two outer `Dense` levels and `Compressed` inner level. Thus, we use the same compressed format for matrices (CSR) as PETSc and Trilinos. SpDISTAL's kernels were run with one rank per node. For CPUs, we run PETSc and CTF with one rank per core, and one rank per socket for Trilinos. All CPU experiments utilize all cores on each node. For GPUs, PETSc and Trilinos were run with one rank per GPU.

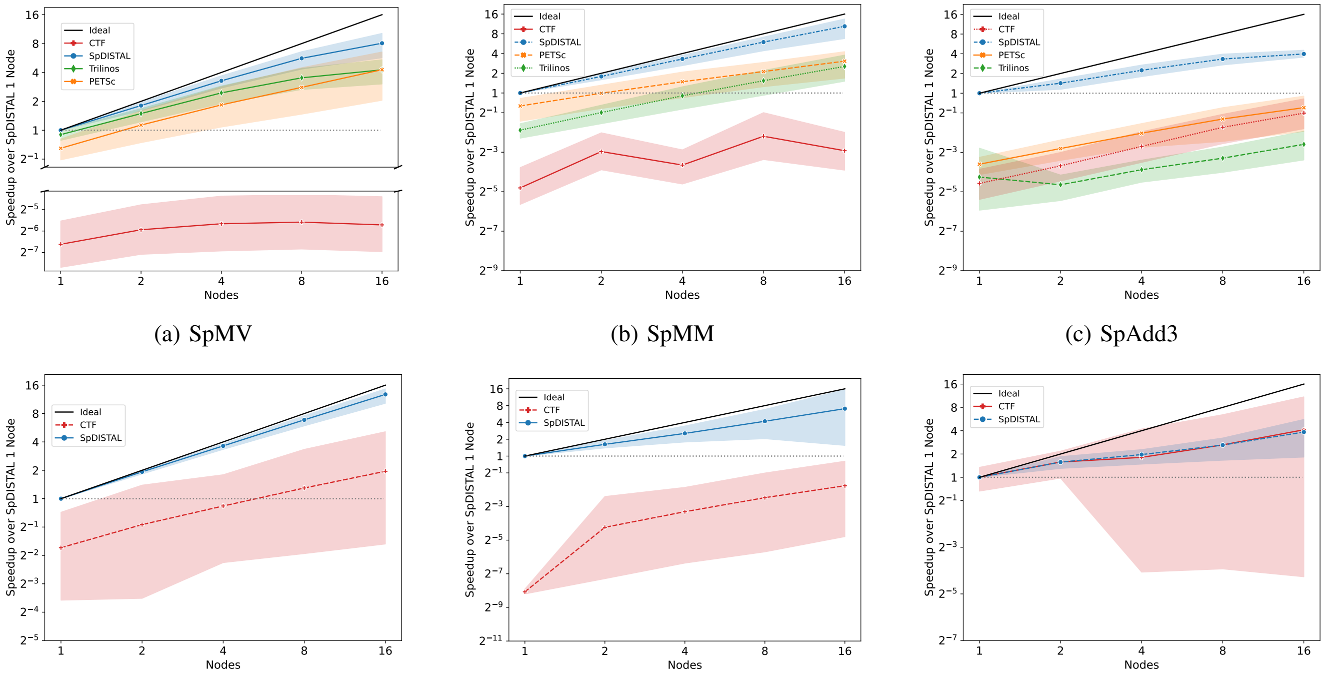
Overview. We evaluate the performance of SpDISTAL by comparing against hand-written systems (PETSc and Trilinos), and interpretation-based systems (CTF), and consider both strong scaling and weak scaling experiments. Our evaluation shows that 1) SpDISTAL achieves performance competitive with (specialized) expert-tuned kernels in hand-written systems and 2) SpDISTAL significantly outperforms (general) interpretation-based approaches to distributed sparse tensor algebra. These results indicate that a compilation-based ap-

[3] https://gitlab.com/StanfordLegion/legion/, commit excluded for review.

[4] https://gitlab.com/petsc/petsc, version 3.16.3, commit e27481de.

[5] https://github.com/trilinos/Trilinos, version 13.2.0, commit 4a5f7906.

[6] https://github.com/cyclops-community/ctf, commit 36b1f6de.



(a) SpMV

(b) SpMM

(c) SpAdd3

(d) SDDMM

(e) SpTTV

(f) SpMTTKRP

Figure 10: Strong scaling results for CPUs. On SpTTV, CTF OOM'ed on the "patents" tensor on 1 node. On SpMTTKRP, CTF OOM'ed on the "freebase_music" tensor on 1 and 2 nodes, and on the "freebase_sampled" tensor at all node counts.

proach that produces bespoke implementations can achieve both generality and high performance, and SpDISTAL is the first such approach for distributed sparse tensor algebra.

A. *Strong Scaling Performance.*

We evaluate SpDISTAL on the following sparse tensor kernels, all of which have been used in prior work [15], [20] to evaluate sparse tensor compilers. SpMV has applications in scientific computing, SpMM and SDDMM appear in sparse machine learning, and SpTTV and SpMTTKRP are used in tensor factorizations arising in data analytics. SpAdd3 is a synthetic benchmark intended to show the benefits of kernel fusion over binary kernels commonly used in libraries.

- SpMV: $a(i) = \mathbf{B}(i,j) \cdot c(j)$
- SpMM: $A(i,j) = \mathbf{B}(i,k) \cdot C(k,j)$
- SpAdd3: $\mathbf{A}(i,j) = \mathbf{B}(i,j) + \mathbf{C}(i,j) + \mathbf{D}(i,j)$
- SDDMM: $\mathbf{A}(i,j) = \mathbf{B}(i,j) \cdot C(i,k) \cdot D(k,j)$
- SpTTV: $\mathbf{A}(i,j) = \mathbf{B}(i,j,k) \cdot c(k)$
- SpMTTKRP: $A(i,l) = \mathbf{B}(i,j,k) \cdot C(j,l) \cdot D(k,l)$

Bolded tensors are sparse while all others are dense. For CPUs, we compare against PETSc, Trilinos and CTF for SpMV, SpMM and SpAdd3. The remaining operations are unsupported by PETSc and Trilinos. For GPUs, we restrict our comparison to PETSc and Trilinos, as CTF does not currently have GPU support that we could use[7]. Both PETSc and Trilinos have GPU support for the SpMV and SpMM kernels, while only Trilinos has GPU support for the SpAdd3 kernel when the output non-zero pattern is unknown. For the remaining higher order expressions we compare against SpDISTAL's CPU kernel.

1) CPU Results: Figure 10 shows speedup plots on CPUs. Each data point is normalized against SpDISTAL on 1 node, and each line is the average speedup over all tensors displayed with a 99% colored confidence interval (computed over all tensors). For all kernels other than SDDMM, we utilize an outer dimension-based distributed algorithm and initial data distributions. For SDDMM, we utilize a non-zero based distributed algorithm and initial data distribution. We experimented with non-zero based parallelization for SpMV, SpMM, SpTTV and SpMTTKRP but found that the extra synchronization required within the leaf kernel costed more than performance gained through load balance. SpAdd3 on CSR matrices is incompatible with the non-zero splitting scheduling transformation, so we used a row-based distribution and distributed algorithm.

For kernels that PETSc and Trilinos directly support (SpMV and SpMM), PETSc and Trilinos achieve performance competitive with SpDISTAL. SpDISTAL achieves median speedups of 1.8x and 1.2x on SpMV and 2.01x and 3.8x on SpMM over PETSc and Trilinos respectively. We attribute the slight performance improvement on SpMV to Legion's deferred execution model that avoids unnecessary synchronization. SpDISTAL achieves a larger speedup over PETSc due to PETSc's current lack of support for multi-threading: SpDISTAL uses OpenMP to dynamically load balance among threads, yielding an improvement. For SpMM, we implement the schedule used by Senanayake et al [20] as the leaf kernel, which appears to outperform the kernel used by PETSc and Trilinos, resulting in lower execution time but similar scaling.

[7] The CTF developers are in progress addressing issues with the GPU backend, but this work was not completed at the time of writing.

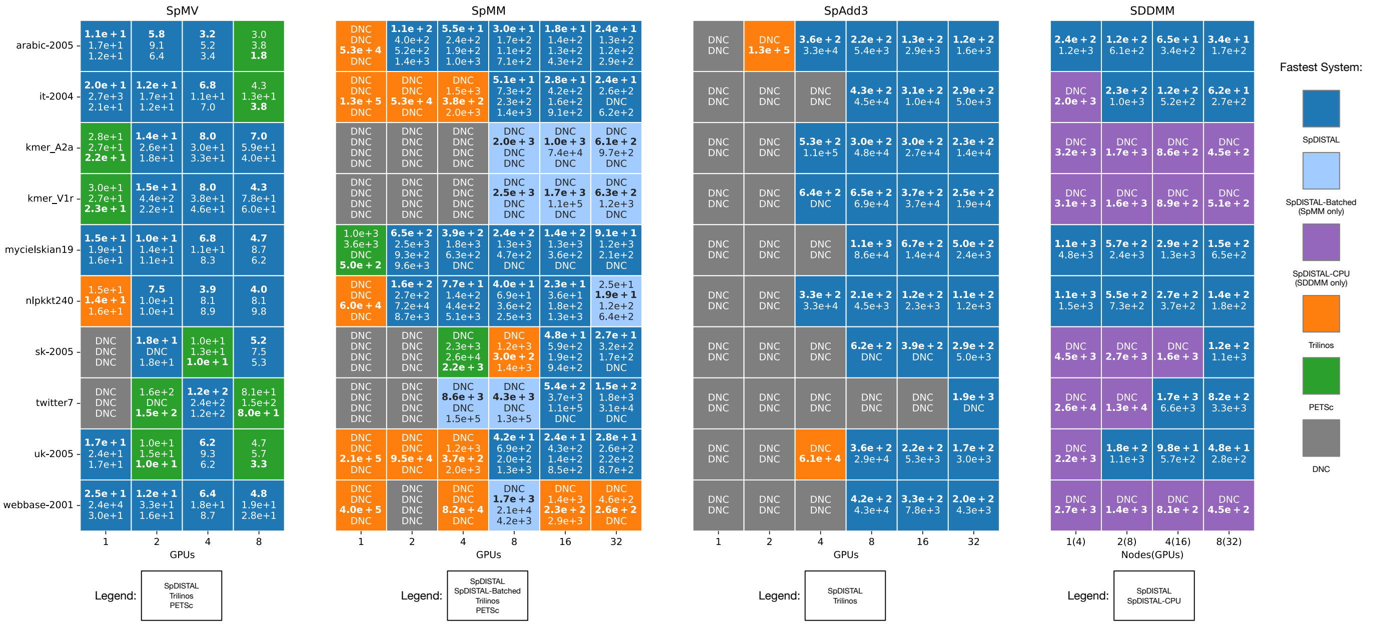


Figure 11: GPU strong scaling results for SpMV, SpMM, SpAdd3 and SDDMM. Each box contains the time in milliseconds by each systems' GPU kernels. DNC (does not complete) indicates that a system failed with an error (OOM, timeout, or other). SpDISTAL-Batched is a memory-conserving SpMM schedule and SpDISTAL-CPU is SpDISTAL's CPU SDDMM kernel.

The benefit of SpDISTAL's ability to fuse computation into a single kernel is demonstrated by SpAdd3. PETSc and Trilinos do not implement SpAdd3 and must use two matrix additions. This approach loses data locality and results in additional sparse matrix assembly operations, allowing SpDISTAL to achieve median speedups of 11.8x and 38.5x over PETSc and Trilinos.

Having shown that SpDISTAL can achieve competitive performance with hand-written libraries, we now compare against CTF's interpretation-based approach. CTF interprets tensor algebra expressions pair-wise by reducing them to distributed matrix-matrix multiplication, element-wise, and transposition operations. As can be seen in the SpMV and SpTTV speedup charts (Figure 10a and Figure 10e), interpretation leads to large slowdowns—SpDISTAL achieves median speedups of 299x, 161x and 19.2x on SpMV, SpTTV and SpAdd3. While interpretation leads to large constant-factor slowdowns for binary kernels, it leads to asymptotic slowdowns for kernels that require fusion, such as SDDMM and SpMTTKRP [15]. To address this slowdown, the CTF authors have developed hand-written, specialized kernels for SDDMM and SpMTTKRP [31]. For SDDMM, SpDISTAL achieves a median speedup of 15.3x over CTF, and achieves near perfect speedup due to its load balanced approach. For SpMTTKRP, SpDISTAL and CTF achieve similar performance (SpDISTAL achieves median 97% of CTF's performance) and CTF has wider range of performance. CTF outperforms SpDISTAL on the "patents" tensor, and completes the SpMTTKRP operation on "patents" significantly faster than on much smaller tensors.

2) GPU Results: GPU strong scaling results are shown in Figure 11 and Figure 12. We use a heatmap-based presentation for the GPU results to address that 1) on some kernel and tensor pairs, systems OOM or error out on different GPU counts and 2) some kernels have no distributed GPU comparison target. These heatmaps display the performance of each system on each input tensor and processor count pair.

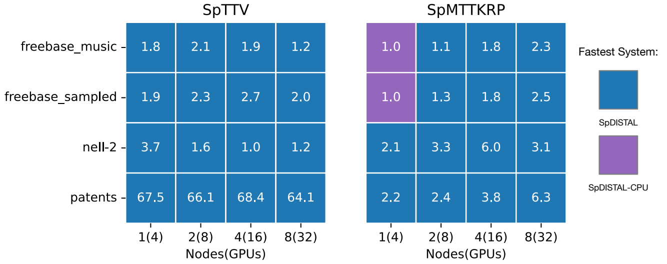


Figure 12: GPU strong scaling results for SpTTV and SpMTTKRP comparing SpDISTAL's GPU and CPU kernels. In each box is the speedup achieved by the faster system over the slower system on the same number of nodes.

We use a row-based algorithm and data distribution for SpMV on GPUs and utilize CuSPARSE to execute the SpMV at the leaves on a single GPU. We found that this approach outperformed the distributed version of the non-zero based schedule utilized by Senanayake et al [20] on the considered dataset and GPU. Due to the short runtime of SpMV on the dataset (order of 10ms), we strong scale only to 8 GPUs. As seen in Figure 11, SpDISTAL outperforms PETSc and Trilinos on 28/38 configurations, and achieves median speedups of 1.07x and 1.65x over PETSc and Trilinos.

SpMM on GPUs had large variability in both the fastest system and the number of systems to successfully complete problem instances. We implement two SpDISTAL schedules for GPU SpMM. The first distributes the non-zeros of the computation equally over all GPUs at the cost of replicating the C matrix, leading to OOMs on some matrix shapes. The other schedule (denoted "SpDISTAL-Batched") conserves memory by distributing the i and j dimensions of the computation to also partition the C matrix at the cost of potential load imbalance and extra communication. Per personal communication with the PETSc developers, the PETSc's current GPU SpMM implementation experiences a significant performance penalty when moving from one to multiple GPUs. Trilinos utilizes CUDA-UVM, allowing it fit some problem instances into GPU memory that SpDISTAL cannot, at the cost of paging this data in and out of the GPU. As Figure 11 shows, the SpDISTAL's load-balanced kernel performs the best once data fits into memory (24/49 configurations), and the memory-conserving kernel wins in 10/49 more configurations when data does not, for a total of 34/49. There are 13/49 configurations where Trilinos beats SpDISTAL's memory conserving schedule when both fit into GPU memory. Based on inspection of Trilinos source code, Trilinos performs a single communication operation to gather the necessary components of the input matrix to each GPU, and overflowing into CUDA-UVM. In contrast, SpDISTAL's memory-conserving algorithm communicates chunks of the dense input matrix in multiple rounds between nodes to fit data within GPU memory. We hypothesize that this choice allows for Trilinos to send fewer messages over the network than SpDISTAL, leading to faster runtimes on some configurations.

Similar to SpAdd3 on CPUs, we utilize a row-based strategy for SpAdd3 on GPUs. PETSc does not support GPU sparse matrix addition when the sparsity pattern of the output matrix is unknown, so we compare against Trilinos for SpAdd3 and find that SpDISTAL significantly outperforms Trilinos due to the ability to fuse computation and avoid allocation of intermediate results. Figure 11 shows that SpDISTAL outperforms Trilinos on 32/34 cases, where Trilinos succeeds in 2/34 cases by fitting matrices into a smaller GPU count with CUDA-UVM.

For SDDMM, SpTTV and SpMTTKRP we compare SpDISTAL's GPU kernels to SpDISTAL's CPU kernels using all the resources on a node. We use a non-zero based algorithm and data distribution for each kernel. This differs from the CPU algorithms, as on GPUs, the additional synchronization within the leaf kernel is outweighed by the load balance over all GPU threads across the machine. Figure 11 and Figure 12 show that SpDISTAL's GPU kernels achieve a median 4.3x speedup for SDDMM, 2.0x speedup for SpTTV and 2.2x speedup for SpMTTKRP when data fits into GPU memory. On SpMTTKRP, SpDISTAL's GPU kernels achieve increasing speedup due to the better load balance offered by the GPU schedule.

B. Weak Scaling Performance.

Figure 13 shows the weak-scaling performance of SpDISTAL's SpMV kernel on synthetic banded matrices up to 64 nodes (256 GPUs). The initial problem size for a single node of CPUs and a single GPU was 700 million non-zeros. We compare against PETSc, and exclude Trilinos due to difficulties reading large matrices from disk. SpDISTAL is configured to use CuSPARSE for local SpMV computations. We find that PETSc achieves perfect weak-scaling performance on both CPUs and GPUs. SpDISTAL's CPU kernels achieves between 90% and 92% of PETSc's performance. SpDISTAL's GPU kernel achieves slightly higher performance, between 1.05x and 1.29x, than PETSc's GPU implementation and has some more performance variability due to millisecond variations in execution time caused by network effects. We attribute the slight improvement to Legion's asynchronous execution model.

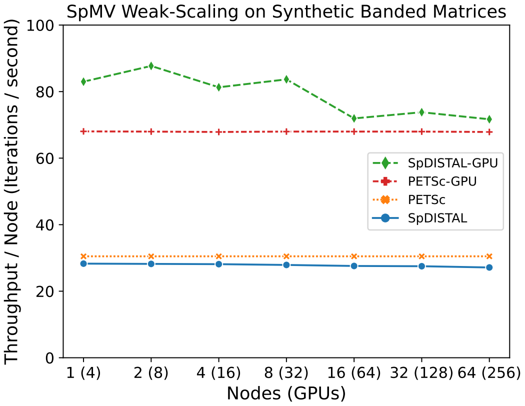


Figure 13: SpMV weak scaling results.

C. Discussion

The scheduling and data distribution languages of SpDISTAL enable concise descriptions of distributed algorithms for sparse tensor computations that achieve high performance across the full range of tensor algebra expressions. Without scheduling and data distribution languages, users would not be able to describe how their data and computation map onto distributed machines.

Our experiments show many instances of SpDISTAL outperforming state-of-the-art systems, enabled by the chosen schedules and data distributions. For some computations, we replicated algorithms used by existing systems (e.g. a row-based distribution for SpMV), resulting in SpDISTAL at least equalling the performance of those systems. In cases where SpDISTAL exceeded the performance of existing systems using the same algorithm, we attribute the performance improvement to the efficiency and optimizations of the Legion runtime.

In some cases, we used SpDISTAL's capabilities to adapt known algorithms to new tensor expressions, such as SpADD3. Our schedule for SpAdd3 extends the row-based algorithm for adding two sparse matrices to a row-based algorithm that fuses the addition across the three input sparse matrices. This fusion yields speedups over other systems that perform pairwise additions and allocate temporary results, as seen in prior works [15], [30].

Finally, scheduling and data distribution primitives for non-zero partitioning allow SpDISTAL to express algorithms not found in existing systems. For SDDMM, GPU SpMM, GPU SpTTV and GPU SpMTTKRP using these primitives yields algorithms that are statically load-balanced across all processors. The perfect load balancing enables high performance at scale, regardless of the input tensor's sparsity structure.

Finally, SpDISTAL's compiler for the scheduling and data distribution languages allows specializing an implementation to a particular sparse tensor computation, in contrast to interpretation. Figure 10 shows that specialized systems can deliver large speedups over interpreted approaches on specific kernels. Significant overheads of interpretation, up to an order of magnitude, have been observed in prior work [13], [31]. SpDISTAL provides generality without sacrificing performance.

VII. Related Work

SpDISTAL is a compiler that implements all of tensor algebra for sparse tensors and targets distributed machines. Prior work includes libraries that implement subsets of sparse tensor algebra on distributed machines, dense compilers that can target single node and distributed systems, and sparse compilers that target single node systems.

SpDISTAL and Cyclops Tensor Framework [11], [12] (CTF) are the only two distributed systems we know of that support all of tensor algebra on sparse tensors. CTF is an interpreter for tensor algebra. We show in our evaluation that SpDISTAL significantly outperforms CTF and achieves this performance through novel compilation techniques that specialize the generated code to the input computation, data layout, data distribution and computation distribution.

PETSc [6] and Trilinos [9] are distributed sparse linear algebra libraries that support different sparse matrix data structures and algorithms for sparse linear algebra. These systems achieve high performance for the subset of tensor algebra implemented within the library, but often have sub-optimal performance for computations that compose multiple library functions. Our evaluation shows that SpDISTAL can achieve performance competitive with these specialized systems on computations that they natively implement. Herault et al. [32] develop a multi-GPU system for binary contractions between block-sparse tensors, but do not support all of tensor algebra or sparsity structures that are not block-sparse.

Sparse compilation techniques for single-node systems have seen attention from researchers. Bik et al. [33], developed an early compiler that transformed dense loops over matrices into sparse loops over the non-zero coordinates. The TACO [15] compiler was the first to describe how to compile all of sparse tensor algebra to CPUs. Chou et al. [27] showed how to extend TACO with an abstraction for definition of new formats without changing the code generation algorithm. Kjolstad et al. [22] and Senanayake et al. [20] extend TACO with sparse iteration space transformations and GPU code generation. Finally, Henry and Hsu et al. [30] generalize TACO beyond addition and multiplication to arbitrary functions. SpDISTAL sets itself apart from these work by showing how to compile sparse tensor algebra to distributed machines containing CPUs and GPUs. However, SpDISTAL utilizes components from these prior works for its programming model and implementation.

Finally, dense compilers for both single-node and distributed systems have been developed. Single-node dense compilers include Halide [21], TVM [24], Tensor Comprehensions [25] and Tiramisu [23]. Tiramisu and Distributed Halide [34] allow for targeting distributed backends. These compilers express computations in high-level DSLs and optimizing transformations through scheduling languages, similar to SpDISTAL. DISTAL [13] is a dense tensor algebra compiler that supports separate specifications of data and computation distribution. The key difference of SpDISTAL from these works is that SpDISTAL enables (distributed) computation over sparse data.

VIII. Conclusion

We introduce SpDISTAL, a system that for distributed sparse tensor algebra through independent specifications of computation, sparse data structures, data distribution and computation distribution. SpDISTAL realizes this programming model through novel compiler techniques for data partitioning, and shows that programming distributed sparse tensor algebra can be general, high performance and productive.

Acknowledgements

We thank our anonymous reviewers for their valuable comments that helped us improve this manuscript. We thank Scott Kovach, Elliot Slaughter, Axel Feldmann, Shiv Sundram and Olivia Hsu for their comments and discussion on early stages of this manuscript. We thank the Legion team for their feedback and support during the development of SpDISTAL. We thank the PETSc, Trilinos and CTF authors for their assistance in setup and benchmarking of their software. Rohan Yadav was supported by an NSF Graduate Research Fellowship. This work was supported in part by the Advanced Simulation and Computing (ASC) program of the US Department of Energy's National Nuclear Security Administration (NNSA) via the PSAAP-III Center at Stanford, Grant No. DE-NA0002373 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration; the U.S. Department of Energy, Office of Science under Award DE-SCOO21516. This work was also supported by the Department of Energy Office of Science, Office of Advanced Scientific Computing Research under the guidance of Dr. Hal Finkel.

References

[1] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Training pruned neural networks," *CoRR*, vol. abs/1803.03635, 2018. [Online]. Available: http://arxiv.org/abs/1803.03635

[2] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *CoRR*, vol. abs/1902.09574, 2019. [Online]. Available: http://arxiv.org/abs/1902.09574

[3] A. Anandkumar, R. Ge, D. J. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *CoRR*, vol. abs/1210.7559, 2012. [Online]. Available: http://arxiv.org/abs/1210.7559

[4] B. W. Bader, M. W. Berry, and M. Browne, *Discussion Tracking in Enron Email Using PARAFAC*. London: Springer London, 2008, pp. 147–163. [Online]. Available: https://doi.org/10.1007/978-1-84800-046-9_8

[5] J. C. Kolecki, *An Introduction to Tensors for Students of Physics and Engineering*, 2002.

[6] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc Web page," https://petsc.org/, 2021. [Online]. Available: https://petsc.org/

[7] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc/TAO users manual," Argonne National Laboratory, Tech. Rep. ANL-21/39 - Revision 3.16, 2021.

[8] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[9] T. Trilinos Project Team, *The Trilinos Project Website*.

[10] T. Tpetra Project Team, *The Tpetra Project Website*.

[11] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S074373151400104X

[12] E. Solomonik and T. Hoefler, "Sparse tensor algebra as a parallel programming model," *CoRR*, vol. abs/1512.00066, 2015. [Online]. Available: http://arxiv.org/abs/1512.00066

[13] R. Yadav, A. Aiken, and F. Kjolstad, "DISTAL: the distributed tensor algebra compiler," ser. PLDI '22, 2022. [Online]. Available: https://arxiv.org/abs/2203.08069

[14] S. Treichler, M. Bauer, R. Sharma, E. Slaughter, and A. Aiken, "Dependent partitioning," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 344–358. [Online]. Available: https://doi.org/10.1145/2983990.2984016

[15] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133901

[16] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

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

[19] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 1047–1058.

[20] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A sparse iteration space transformation framework for sparse tensor algebra," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428226

[21] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[22] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor algebra compilation with workspaces," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 180–192.

[23] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," *CoRR*, vol. abs/1804.10694, 2018. [Online]. Available: http://arxiv.org/abs/1804.10694

[24] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018. [Online]. Available: http://arxiv.org/abs/1802.04799

[25] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018. [Online]. Available: http://arxiv.org/abs/1802.04730

[26] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276491

[27] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276493

[28] ——, "Automatic generation of efficient sparse tensor format conversion routines," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 823–838. [Online]. Available: https://doi.org/10.1145/3385412.3385963

[29] LLNL, "Lassen," 2021. [Online]. Available: https://hpc.llnl.gov/hardware/platforms/lassen

[30] R. Henry, O. Hsu, R. Yadav, S. Chou, K. Olukotun, S. Amarasinghe, and F. Kjolstad, "Compilation of sparse array programming models," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485505

[31] Z. Zhang, X. Wu, N. Zhang, S. Zhang, and E. Solomonik, "Enabling distributed-memory tensor completion in python using new sparse tensor kernels," *CoRR*, vol. abs/1910.02371, 2019. [Online]. Available: http://arxiv.org/abs/1910.02371

[32] T. Herault, Y. Robert, G. Bosilca, R. J. Harrison, C. A. Lewis, E. F. Valeev, and J. J. Dongarra, "Distributed-memory multi-gpu block-sparse tensor contraction for electronic structure," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 537–546.

[33] A. J. C. Bik, P. J. H, P. J. Brinkhaus, and H. A. Wijshoff, "The sparse compiler mt1: A reference guide."

[34] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851157