Compilation of Dynamic Sparse Tensor Algebra

STEPHEN CHOU, MIT CSAIL, USA
SAMAN AMARASINGHE, MIT CSAIL, USA

Many applications, from social network graph analytics to control flow analysis, compute on sparse data that evolves over the course of program execution. Such data can be represented as dynamic sparse tensors and efficiently stored in formats (data layouts) that utilize pointer-based data structures like block linked lists, binary search trees, B-trees, and C-trees among others. These specialized formats support fast in-place modification and are thus better suited than traditional, array-based data structures like CSR for storing dynamic sparse tensors. However, different dynamic sparse tensor formats have distinct benefits and drawbacks, and performing different computations on tensors that are stored in different formats can require vastly dissimilar code that are not straightforward to correctly implement and optimize.

This paper shows how a compiler can generate efficient code to compute tensor algebra operations on dynamic sparse tensors that may be stored in a wide range of disparate formats. We propose a language for precisely specifying recursive, pointer-based data structures, and we show how this language can express many different dynamic data structures, including all the ones named above as well as many more. We then describe how, given high-level specifications of such dynamic data structures, a compiler can emit code to efficiently access and compute on dynamic sparse tensors that are stored in the aforementioned data structures.

We evaluate our technique and find it generates efficient dynamic sparse tensor algebra kernels that have performance comparable to, if not better than, state-of-the-art libraries and frameworks such as PAM, Aspen, STINGER, and Terrace. At the same time, our technique supports a wider range of tensor algebra operations—such as those that simultaneously compute with static and dynamic sparse tensors—than Aspen, STINGER, and Terrace, while also achieving significantly better performance than PAM for those same operations.

CCS Concepts: • **Software and its engineering** → **Abstraction, modeling and modularity**; **Source code generation**; *Domain specific languages*; • **Mathematics of computing** → *Mathematical software performance*.

Additional Key Words and Phrases: dynamic sparse tensors, sparse tensor formats, pointer-based data structures, sparse tensor algebra, sparse tensor algebra compilation, node schema language

1 INTRODUCTION

Sparse matrices and tensors (i.e., multidimensional arrays) are commonly used to represent data in many domains, including graph analytics [Mattson et al. 2013], machine learning [Park et al. 2016; Rajbhandari et al. 2017], and many others. Countless formats for efficiently storing sparse tensors in memory have been proposed [Baskaran et al. 2012; Dhulipala et al. 2019; Ediger et al. 2012; Li et al. 2018; Liu and Vinter 2015; Monakov et al. 2010; Pandey et al. 2021; Smith and Karypis 2015], and many of these formats are supported by widely used sparse linear/tensor algebra libraries like Intel oneMKL [Intel 2020] and graph processing frameworks like Ligra [Shun and Blelloch 2013].

Authors' addresses: Stephen Chou, MIT CSAIL, 32-G778, 32 Vassar Street, Cambridge, MA, 02139, USA, s3chou@csail.mit.edu; Saman Amarasinghe, MIT CSAIL, 32-G744, 32 Vassar Street, Cambridge, MA, 02139, USA, saman@csail.mit.edu.

175

(a) Original sparse matrix   (b) Original matrix stored in CSR   (c) Original matrix stored as BSTs

(d) Modified sparse matrix   (e) Modified matrix stored in CSR   (f) Modified matrix stored as BSTs
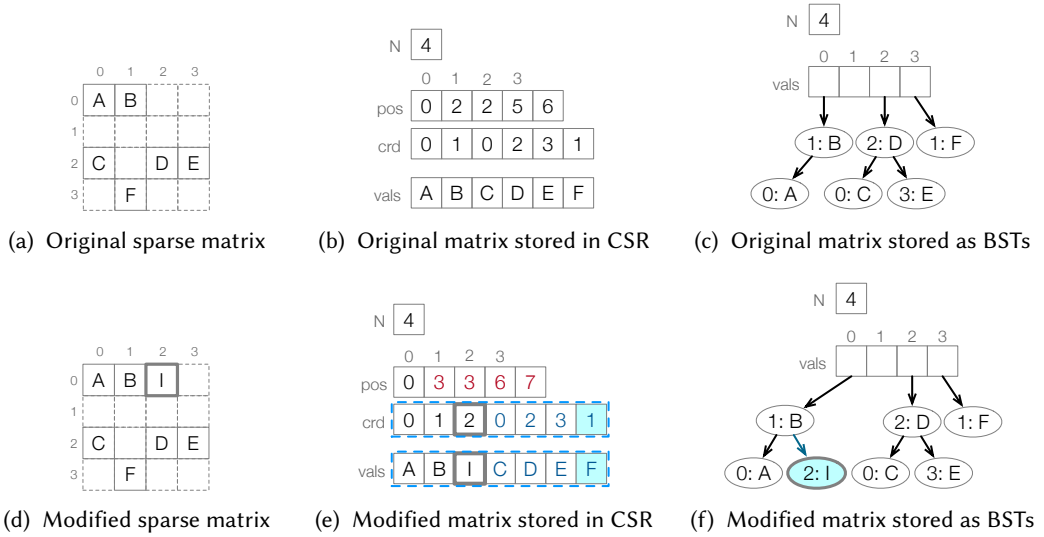
Fig. 1. Examples of the same sparse matrix stored in CSR and as BSTs. Inserting a new nonzero I into CSR at coordinates $(0, 2)$ requires shifting stored nonzeros in memory and may require reallocating the `crd` and `vals` arrays. By contrast, inserting the same nonzero into a BST only requires allocating a new node.

Being able to efficiently compute with sparse tensors is crucial since real-world applications often work with large data sets that can be represented as sparse tensors. As a result, lots of research effort has been devoted to optimizing the performance of sparse tensor computations [Azad and Buluç 2017; Bell and Garland 2008; Hong et al. 2019; Kjolstad et al. 2017; Smith et al. 2015; Zhang et al. 2017]. High-performance libraries like oneMKL can efficiently compute with sparse tensors by storing them in formats like compressed sparse row (CSR) that, as Figure 1b shows, use a fixed number of arrays to compactly store nonzeros in memory. Such array-based formats provide good cache spatial locality and are ideal for storing *static* sparse tensors that have fixed sparsity structures (i.e., they do not gain or lose nonzeros over the course of an application's execution). However, it is generally inefficient to modify a tensor that is stored in a static sparse tensor format, since this may require already-stored nonzeros to be moved around in memory.[1] As Figure 1e shows, for instance, inserting a nonzero into a CSR matrix requires all subsequent nonzeros to be shifted back so that space can be made for the new nonzero. This incurs significant performance overhead.

However, many real-world applications need to work with *dynamic* sparse tensors, which have constantly-evolving sparsity structures [Cheng et al. 2012; King et al. 2016]. For instance, a graph that encodes friendship relations in a social network can be represented by a sparse tensor (i.e., its adjacency matrix). Such a tensor may need to be regularly updated by inserting new nonzeros in order to reflect new friendships that are formed between users. To allow efficient modification of dynamic sparse tensors, specialized formats for storing such tensors typically use pointers to link together stored nonzeros. This makes it possible to, for instance, insert a new nonzero without having to move already-stored nonzeros in memory. As an example, Figure 1c shows a dynamic tensor format that stores each row of a tensor using a binary search tree (BST). Since nodes in a

---

[1]This problem could be avoided by storing the tensor in a dense array, which reserves space for all elements (including zeros) in the tensor. Unfortunately, many real-world applications work with large tensors that, in order to be stored as dense arrays, may require orders of magnitude more memory than is typically available. Additionally, for real-world tensors that are highly sparse, storing them in dense arrays can significantly reduce compute performance [Kjolstad et al. 2017].

BST do not have to be stored contiguously in memory, a new nonzero can be inserted by simply allocating a new node and attaching it to the BST without moving any existing node in memory, as Figure 1f shows. The main drawback of dynamic tensor formats is that they are typically less efficient to compute with than static, array-based tensor formats, since pointer-based data structures provide less spatial locality. Still, because converting a tensor between formats can incur significant overhead [Xie et al. 2018], an application that must compute on dynamic sparse tensors can often do so more efficiently by keeping the tensors stored in dynamic tensor formats.

There exist many distinct dynamic sparse tensor formats though, and they all possess different trade-offs. A format that uses BSTs to store nonzeros, for instance, can be efficiently modified but is also relatively inefficient to iterate over since its memory layout effectively provides no cache spatial locality. Conversely, a format that uses blocked data structures like B-trees to store nonzeros can be more efficiently accessed, since some nonzeros are stored close together in memory (which increases spatial locality). For the same reason, however, such a format cannot be as efficiently modified. Thus, to be able to support a wide range of applications that have different proportions of data modification and compute, a sparse tensor algebra system must be able to efficiently work with many disparate dynamic tensor formats.

1.1 Challenges of Supporting Dynamic Sparse Tensors

Unfortunately, existing libraries that work with dynamic sparse tensors (or dynamic graphs encoded as sparse adjacency matrices) each typically only supports storing such tensors in a limited number of formats. In order to effectively utilize any particular format, a library must be able to both efficiently in-place modify and efficiently compute on tensors stored in that format. To support modifying dynamic sparse tensors, a library essentially only needs a routine for inserting new nonzeros into a tensor and a routine for deleting nonzeros from a tensor. Thus, a library can support efficient in-place modification for a wide variety of formats by simply providing a bounded number of optimized routines for each format, which library developers can reasonably implement.

On the other hand, supporting efficient computation with a wide variety of formats is significantly more complicated. This is because there is effectively an unbounded number of operations that users might want to compute, and each of these operations not only can have an arbitrary number of operands but can also store those operands in distinct, arbitrary formats. As Figure 2 shows, efficiently computing with dynamic sparse tensors can require code that vary significantly—and that are all non-trivial to implement and optimize correctly—depending on both the operation being computed and the formats used to store dynamic sparse tensor operands. The result is a combinatorial explosion of potential kernels that a library developer would have to implement in order to support efficient computation with a wide range of dynamic sparse tensor formats. In turn, this limits the number of formats that hand-optimized libraries can reasonably support in practice.

This motivates the need for a compiler-based technique that can instead automatically generate efficient code to compute on dynamic sparse tensors stored in arbitrary formats. However, existing sparse linear and tensor algebra compilers like TACO [Chou et al. 2018, 2020; Kjolstad et al. 2019, 2017] and MLIR [Bik et al. 2022] cannot readily, if at all, generate code to efficiently compute with operands that are stored in a wide range of disparate dynamic sparse tensor formats.

1.2 Contributions

We propose the first technique that, given high-level specifications of a wide variety of dynamic sparse tensor formats, can automatically generate tensor algebra kernels that efficiently compute on sparse tensors stored in the aforementioned formats. In particular, we propose a language for precisely specifying how nonzeros can be stored in recursive, pointer-based data structures such as BSTs and linked lists, which compose to form many known dynamic tensor formats. We

```
void map_b(blist* b, double* c, double& a) {
  while (b) {
    for (int32_t p = 0; p < b->B; p++) {
      int32_t j = b->e[p].first;
      a += b->e[p].second * c[j];
    }
    b = b->n;
  }
}

void compute(...) {
  for (int32_t i = 0; i < N; i++) {
    double sum = 0.0;
    map_b(b[i]->h, c, sum);
    a[i] = sum;
  }
}
```

(a) SpMV with matrix stored as block linked lists

```
void map_b(bst* b, double* c, double& a) {
  if (b) {
    if (b->l)
      map_b(b->l, a, c);
    int32_t j = b->e.first;
    a += b->e.second * c[j];
    if (b->r)
      map_b(b->r, a, c);
  }
}

void compute(...) {
  for (int32_t i = 0; i < N; i++) {
    double sum = 0.0;
    map_b(b[i]->r, c, sum);
    a[i] = sum;
  }
}
```

(b) SpMV with matrix stored as BSTs

```
inline uint8_t
iter_bst(uint8_t state, bst*& n,
         call_stack<uint8_t,bst*>& cs,
         int32_t& c, double& v) {
  if (state == 1)
    goto iter_resume1;
  cs.emplace(0, n);
  while (!cs.empty()) {
    n = get<1>(cs.top());
    if (get<0>(cs.top()) == 1)
      goto call_resume1;
    while (n) {
      if (n->l) {
        get<0>(cs.top()) = 1;
        get<1>(cs.top()) = n;
        cs.emplace(0, n->l);
        goto call_end;
call_resume1:;
      }
      c = n->e.first;
      v = n->e.second;
      return 1;
iter_resume1:
      n = n->r;
    }
    cs.pop();
call_end:;
  }
  return 0;
}
```

```
inline uint8_t
iter_blist(uint8_t state, blist*& b,
           int32_t& p, int32_t& c, double& v) {
  if (state == 1)
    goto iter_resume1;
  while (b) {
    for (p = 0; p < b->B; p++) {
      c = b->e[p].first;
      v = b->e[p].second;
      return 1;
iter_resume1:;
    }
    b = b->n;
  }
  return 0;
}

void compute(...) {
  ...
  for (int32_t i = 0; i < N; i++) {
    bst* bn = b[i]->r;
    blist* cn = c[i]->h;
    uint8_t bs = iter_bst(0, bn, bstack, jb, bv);
    uint8_t cs = iter_blist(0, cn, cp, jc, cv);
    while (bs && cs) {
      int32_t j = min(jb, jc);
      if (j == jb && j == jc)
        a[pa++] = bv * cv;
      if (j == jb) bs = iter_bst(bs, ..., bv);
      if (j == jc) cs = iter_blist(cs, ..., cv);
    }
  }
}
```

(c) Element-wise multiplication of a matrix stored as BSTs and a matrix stored as block linked lists

Fig. 2. Examples of different dynamic sparse tensor algebra kernels with operands stored in disparate formats. As these examples demonstrate, efficiently computing with dynamic sparse tensors can require very different code depending on the format used to store the tensors ((a) vs. (b)) as well as the computation ((a)/(b) vs. (c)). Our technique is able to automatically generate all of these kernels.

show how a compiler can use these specifications to generate iterators and map functions for the aforementioned data structures, which can be invoked to efficiently compute on dynamic sparse tensors. Additionally, we propose an abstract interface that captures how dynamic data structures can be efficiently assembled, and we show how a compiler can use implementations of this interface to generate tensor algebra kernels that store the results of computations in dynamic tensor formats. In summary, our contributions include the

node schema language, which lets users precisely define a wide range of dynamic data structures that can be used to store dynamic sparse tensors (Section 3.1); an

assembly abstraction that captures how dynamic data structures can be efficiently constructed (Section 3.2); and

code generation techniques that, guided by the above abstractions, emit efficient code to compute tensor algebra operations on dynamic sparse tensors (Section 4).

We implement our technique as a prototype extension to the TACO sparse tensor algebra compiler, and our evaluation shows that our technique generates efficient dynamic sparse tensor algebra kernels (Section 5). Code that our technique generates has performance comparable to, if not better than, equivalent code that can be implemented using Aspen [Dhulipala et al. 2019], STINGER [Ediger et al. 2012], and Terrace [Pandey et al. 2021], which are three state-of-the-art dynamic graph processing frameworks. At the same time, our technique can generate code for many tensor algebra computations that are not readily supported by the other aforementioned frameworks, such as those that simultaneously compute with static and dynamic sparse tensors. For these other computations, code that our technique generates can significantly outperform PAM [Sun et al. 2018], which is a lower-level, parallel ordered key-value maps library that can be used to also implement the same kernels.

2 BACKGROUND

In this section, we give a brief overview of some of the many formats that have been proposed for storing dynamic sparse tensors. Additionally, we briefly describe the sparse tensor algebra compilation techniques of Kjolstad et al. [2019, 2017] and Chou et al. [2018], which generate efficient code that compute on static sparse tensors stored in array-based formats.

2.1 Dynamic Sparse Tensor Formats

There exist many formats for storing dynamic sparse tensors in memory, all of which possess different trade-offs. Figure 3 shows several representative examples of these formats for two-dimensional dynamic sparse tensors (i.e., matrices).

A standard way of storing a dynamic sparse matrix (such as the adjacency matrix of a dynamic graph) is as a collection of adjacency lists, each of which stores the nonzeros in a single row of a matrix. Each adjacency list can be stored as a linked list [Cormen et al. 2009], with each node in the linked list storing the column coordinate and value of a nonzero (Figure 3b). This representation enables new nonzeros to be efficiently added to a matrix by simply appending them to the appropriate adjacency lists, which can be done without moving any existing nonzero in memory. Additionally, the collection of adjacency lists may itself be stored as a linked list, forming the list of lists representation; this enables new rows to be efficiently added to a matrix as well.

One drawback with linked lists though is that, when iterating over stored nonzeros, each access can potentially incur a cache miss, since nodes in a linked list are typically not be stored contiguously in memory. This increases the overhead of accessing nonzeros, which reduces performance when computing with dynamic tensors that are stored as linked lists. To address this limitation, some high-performance graph processing frameworks such as STINGER [Ediger et al. 2012] instead

(a) A 6×6 tensor

(b) Adjacency list

(c) Block linked list

(d) Binary search tree

(e) C-tree

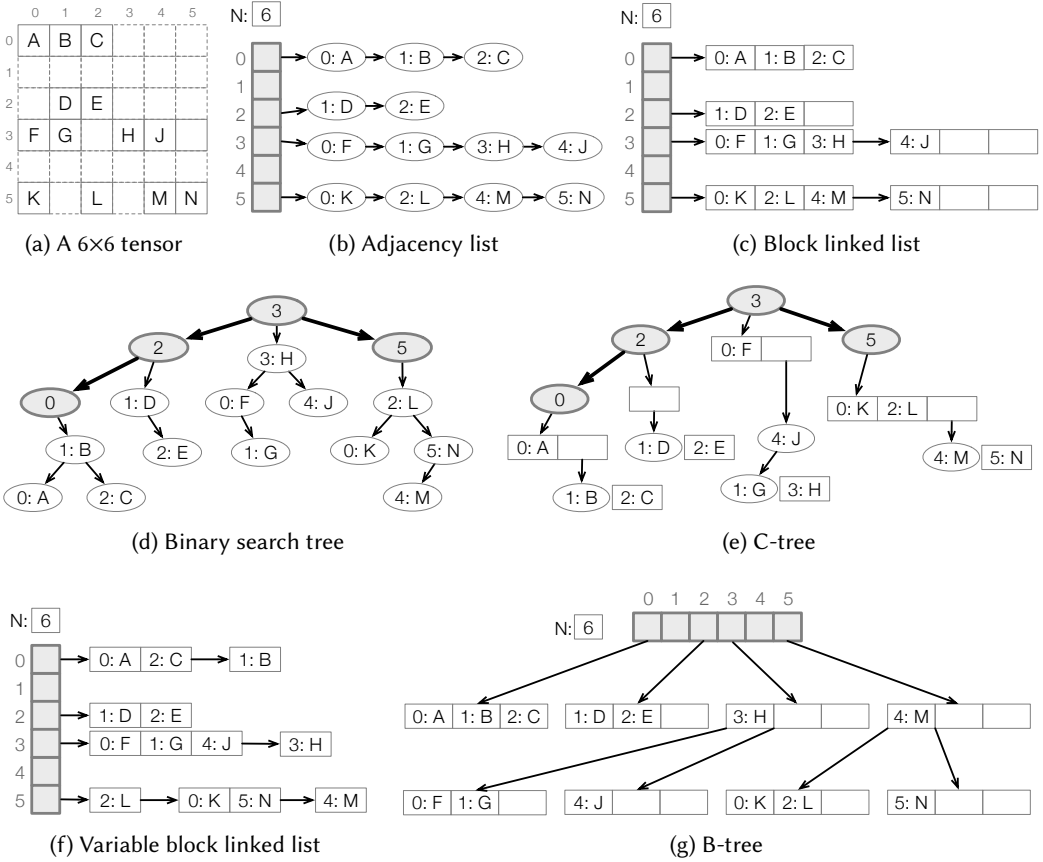(f) Variable block linked list

(g) B-tree

Fig. 3. Examples of disparate dynamic sparse tensor formats (b–g) storing the same tensor (a).

use block linked lists that store multiple nonzeros in each node (Figure 3c), which amortizes the overhead of each node access. In a typical block linked list, every node contains an array of the same size and is able to store the same maximum number of nonzeros. However, some frameworks such as GraphOne [Kumar and Huang 2019] use variable block linked lists that allow different nodes to have arrays of different sizes, enabling nodes to store different maximum numbers of nonzeros (Figure 3f). This allows updates to be efficiently batched, with every batch of new nonzeros—regardless of the size of the batch—inserted as just a single new node.

Another way of representing a dynamic sparse matrix is to store the set of non-empty rows using a (balanced) binary search tree and additionally store each row's nonzeros using a separate BST (Figure 3d) [Dhulipala et al. 2019]. A benefit of using BSTs to store nonzeros is that it enables new nonzeros to be efficiently inserted while also keeping the data structure sorted. This is useful for computations that require accessing nonzeros in order. Again though, to amortize the overhead of accessing nodes in a tree, many high-performance graph processing frameworks instead use block tree data structures that store multiple nonzeros in each node. For instance, Aspen [Dhulipala et al. 2019] represents each row of a dynamic graph's adjacency matrix using a C-tree, which stores only a subset of nonzeros (i.e., *heads*) directly in a BST (Figure 3e). The remaining nonzeros, meanwhile, are stored in either a *prefix* (which contains all nonzeros that have smaller coordinates than any head element) or in chunks (i.e., *tails*) that are each associated with a distinct head element.

Similarly, Terrace [Pandey et al. 2021] supports storing each row of an adjacency matrix using B-trees, which generalize BSTs in a different way by allowing each node to store more than two children in addition to storing multiple nonzeros (Figure 3g).

2.2 Sparse Tensor Algebra Compilation

Our technique builds on the techniques of Kjolstad et al. [2019, 2017] and Chou et al. [2018], which are implemented in the TACO sparse tensor algebra compiler. TACO's code generator takes as input a tensor algebra computation expressed in *concrete index notation*, which specifies how each element in the output tensor should be computed in terms of elements in the input tensors. (For example, matrix addition can be expressed in concrete index notation as $\forall_i \forall_j A_{ij} = B_{ij} + C_{ij}$, which specifies that each element in the output tensor A is the sum of the corresponding elements in the input tensors B and C.) Given such a concrete index notation statement, TACO's code generator can recursively lower it to imperative code by emitting one or more loops to iterate over each dimension. So to generate code that computes matrix addition, for instance, the code generator first emits one or more loops to iterate over all rows of B and C (i.e., dimension i). Then, within each emitted loop over the rows of B and C, the code generator emits one or more loops to iterate over all columns (i.e., dimension j) within a row in order to compute the element-wise sum of that row.

To generate code that computes with sparse tensors stored in specific formats though, TACO additionally requires the user to specify the format of each input and output tensor. Chou et al. show how a wide range of static, array-based sparse tensor formats can be expressed as compositions of *level formats*, each of which stores a dimension of a tensor. The CSR format shown in Figure 1b, for instance, can be expressed as a composition of the dense and compressed level formats, which store the row and column dimensions respectively. The dense level format uses a single scalar variable N to encode a dense set of rows with coordinates from 0 to $N - 1$, while the compressed level format uses a pos array and a crd array to store the column coordinates of each row's nonzeros. Both level formats—and all other level formats—implement snippets of imperative code that precisely describe how their underlying data structures can be accessed or assembled. This lets TACO's code generator emit efficient code to compute with tensors in specific formats by inlining the aforementioned code snippets into the generated loops.

3 DYNAMIC TENSOR FORMAT ABSTRACTIONS

In the same way that static tensor formats can be expressed as compositions of per-dimension level formats (as summarized in Section 2.2), dynamic tensor formats can also be expressed as compositions of per-dimension formats by generalizing level formats to support dynamic, pointer-based data structures. Assume, for instance, we can define new level formats like bst, ctree, and blist that use BSTs, C-trees, and block linked lists to store a tensor dimension, respectively. We can then express Aspen's adjacency matrix representation (Figure 3e) as (bst, ctree), indicating that the set of non-empty rows are stored using a BST while the set of non-zero columns for each row are stored using a C-tree. A tensor format may also be composed of level formats that use both static (array-based) and dynamic data structures. For example, the composition (dense, blist) describes a tensor format that stores a matrix as a dense array of block linked lists, each of which stores a row of the matrix (Figure 3c); this format is akin to STINGER's adjacency matrix representation.

In the rest of this section, we show how to precisely define level formats that store tensor dimensions using dynamic data structures. In particular, we propose a new language that we call the *node schema language*, and we show how a user can use this language to precisely specify how a dynamic data structure stores nonzeros (or non-empty subtensors) in memory (Section 3.1). We also show how, by implementing a common abstract interface that we define, a user can precisely specify how dynamic data structures are assembled (Section 3.2). As Section 4 will show, these

⟨*node_schema*⟩   ::= ⟨*supertype_def*⟩* ⟨*node_def*⟩⁺
⟨*supertype_def*⟩ ::= 'def' 'supertype' ⟨*name*⟩
⟨*node_def*⟩   ::= 'def' ⟨*name*⟩ [':' ⟨*name*⟩] '{' ⟨*field_def*⟩⁺ [⟨*sequence_def*⟩] '}'
⟨*field_def*⟩   ::= ⟨*name*⟩ ':' ⟨*type*⟩
⟨*type*⟩   ::= ⟨*elem_type*⟩ | ⟨*child_type*⟩ | ⟨*size_type*⟩ | ⟨*metadata_type*⟩ | 'parent'
⟨*elem_type*⟩   ::= 'elem' [⟨*array_type*⟩] ['nonempty']
⟨*child_type*⟩   ::= ⟨*name*⟩ [⟨*array_type*⟩] ['nonempty']
⟨*array_type*⟩   ::= '[' (⟨*name*⟩ | ⟨*const*⟩) ']'
⟨*size_type*⟩   ::= 'size' ['in' ⟨*array_size*⟩]
⟨*array_size*⟩   ::= '[' ⟨*const*⟩ ',' (⟨*const*⟩ | '*') ']'
⟨*metadata_type*⟩ ::= 'bool' | 'int8' | 'uint8' | 'int16' | 'uint16' ...
⟨*sequence_def*⟩   ::= 'seq' '=' ⟨*seq_entry*⟩ (',' ⟨*seq_entry*⟩)*
⟨*seq_entry*⟩   ::= ⟨*name*⟩ | '{' ⟨*name*⟩ (',' ⟨*name*⟩)* '}'

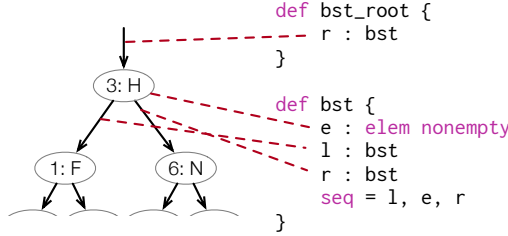Fig. 4. Syntax of the node schema language.

Fig. 5. The node schemas for a BST precisely specifies how nonzeros are stored in nodes of a BST and how these nodes are linked together.

specifications enable our technique to generate efficient code for computing on sparse tensors that are stored in dynamic tensor formats.

3.1 Node Schema Language

A wide range of dynamic tensor data structures, including all those described in Section 2.1, can be modeled as collections of nodes that are stored non-contiguously in memory, with each node storing a subset of nonzeros. To precisely define any dynamic data structure, our technique requires a user to provide *schemas* of the data structure's nodes, which specify how stored nonzeros are distributed amongst the nodes and how nodes are linked together. These schemas can be expressed using the node schema language, the syntax for which is provided in Figure 4.

The node schema language allows users to define nodes that can contain an arbitrary number of fields. Each field may store nonzeros (or, more generally, non-empty subtensors) or store references to other nodes. As an example, Figure 5 shows how binary search trees can be precisely defined using the node schema language. In particular, a binary search tree consists of two types of nodes: a bst_root node, which simply stores a reference to the root of the tree, and bst nodes, which actually contain the nonzeros. The schema for bst nodes specifies that each node stores one nonzero e as well as stores references to up to two child nodes l and r, both of which are of the same type. (The nonempty annotation specifies that each node must store exactly one nonzero and cannot be empty.) Furthermore, the schema contains a *sequence attribute* (seq) that specifies the ordering of nonzeros stored by all reachable nodes; in particular, all nonzeros reachable from l must have smaller coordinates than e, which in turns must have a smaller coordinate than all nonzeros

```
                         def blist {
 def list {                 e : elem[B] nonempty          def vblist {
   e : elem nonempty        n : blist                       e : elem[B] nonempty
   n : list                 B : size in [0, 3]              n : vblist
   seq = {e}, n             seq = {e}, n                    B : size
 }                        }                               }

 def list_head {          def blist_head {                def vblist_head {
   h : list                 h : blist                       h : vblist
 }                        }                               }
```

| (a) Linked list | (b) Block linked list | (c) Variable block linked list |

```
                              def rbtree {
 def ttree {                    e : elem nonempty                       def chunk {
   e : elem[B] nonempty         l : rbtree                                e : elem[N] nonempty
   l : ttree                    r : rbtree                                N : size
   r : ttree                    p : parent                                seq = {e}
   B : size in [1, 5]           c : bool          def tree {           }
   seq = l, {e}, r              seq = l, e, r       h : elem nonempty
 }                            }                     t : chunk             def ctree {
                                                    l : tree                p : chunk
 def ttree_root {             def rbtree_root {      r : tree                r : tree
   r : ttree                    r : rbtree           seq = l, h, t, r        seq = p, r
 }                            }                     }                     }
```

| (d) T-tree | (e) Red-black tree | (f) C-tree |

```
 def supertype btree             def btree_leaf : btree {
                                   e : elem[B] nonempty
 def btree_internal : btree {    B : size in [1, 3]
   e  : elem[B] nonempty         seq = {e}                      def hybrid {
   cf : btree nonempty         }                                  e : elem[B] nonempty
   cr : btree[B] nonempty                                         r : btree
   B  : size in [1, 3]         def btree_root {                   B : size in [0, 5]
   seq = cf, {e, cr}             r : btree                        seq = {e}, r
 }                            }                                 }
```

| (g) B-tree | (h) Fixed-size array/B-tree hybrid |

Fig. 6. Node schemas for a wide range of dynamic data structures, including all those in Figure 3.

reachable from r. Meanwhile, the schema for the bst_root node simply specifies that it stores a reference to the root node r, which may be null if the tree is empty.

The node schema language assumes, by default, that any data structure being defined is acyclic. This means that, for instance, while a bst node may store references to child nodes of the same type, it cannot have an ancestor node as its child. (That said, as Figure 6e demonstrates, one can define a variant of BSTs that stores a reference to each node's parent node in a parent field. As the example also highlights though, a node's sequence attribute may not constrain the ordering of nonzeros that are stored in the parent node.)

Nodes in a dynamic data structure may be defined to store more than one nonzero. For instance, Figure 6d shows how T-trees [Lehman and Carey 1986], which generalize BSTs by having each node store a bounded-size block of nonzeros, can be precisely defined. The schema for ttree nodes specifies that each node can store multiple nonzeros contiguously in an array e, with the exact number of nonzeros that e contains being stored in a separate field B. Different nodes may store different numbers of nonzeros, but the in clause (in the declaration of the B field) constrains each

```
def blist {
  e : elem[B]
  n : blist
  B : size in [0, 3]
  seq = {e}, n
}
```
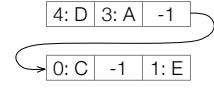
| 0: C | | |

| -1 | 4: D | |
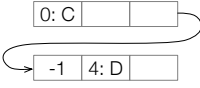
(a) With slots possibly being empty or storing nonzeros up to position B in each block

```
def blist {
  e : elem[3]
  n : blist
  seq = {e}, n
}
```

| 0: C | -1 | -1 |

| -1 | 4: D | -1 |
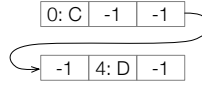
(b) With all slots possibly being empty or storing nonzeros

```
def blist {
  e : elem[3]
  n : blist
}
```

| 4: D | 3: A | -1 |

| 0: C | -1 | 1: E |

(c) With nonzeros unsorted (by coordinates)

Fig. 7. The node schema language can describe many variants of block linked lists. In the subfigures above, unlabeled slots are those that, at run time, can automatically be assumed to not store any nonzero (i.e., must be empty) and can therefore be skipped when iterated over. Slots labeled "-1", on the other hand, might store nonzeros (i.e., are only possibly empty) and must therefore be explicitly checked when iterated over.

node to contain between one to four nonzeros. As with BSTs, the sequence attribute specifies that all nonzeros stored in a node (in array e) have larger coordinates than all nonzeros stored in the left subtree l but smaller coordinates than all nonzeros stored in the right subtree r. Additionally though, the {e} term in the sequence attribute indicates that nonzeros are stored within e in increasing order of their coordinates. This means e[0] stores the nonzero with the smallest coordinate, e[1] stores the nonzero with the second-smallest coordinate, and so on. More generally, a sequence attribute term enclosed within braces may reference multiple arrays, indicating that the array elements are ordered in interleaving order. So, for instance, the {e, cr} term in the sequence attribute for internal nodes of B-trees (Figure 6g) indicates that e[0] has a smaller coordinate than all nonzeros stored in the subtree cr[0], which in turn all have smaller coordinates than e[1], and so on.

Annotations to node schemas and their fields—including nonempty annotations as well as sequence attributes—are strictly optional, making it possible to define many practical variants of a dynamic data structure. Figures 6b and 7, for instance, show how the node schema language can be used to define four variants of block linked lists, each of which pads blocks and orders stored nonzeros in a different way. Similarly, Figure 6f shows how a declaration of a size field can omit the in clause, indicating that the size of an array field is unconstrained. This makes it possible to precisely define a C-tree, which, unlike T-trees, does not strictly bound the number of nonzeros stored in each node.

A dynamic data structure may further be defined to consist of multiple types of nodes that store nonzeros in different ways. Figure 6f demonstrates, for instance, how C-trees can be expressed in the node schema language by defining two types of nodes—tree and chunk—for storing nonzeros. In particular, tree nodes organize all of the head elements into a BST, with each node storing a single head element in its h field. Meanwhile, each chunk node uses a single array e to store either the C-tree's prefix or all tail elements that correspond to a particular head element.

While different types of nodes may possess different sets of fields, they can nevertheless share a common supertype, which allows a single reference to point to a node that can be of one of several different types. For instance, B-trees consist of two types of nodes: internal nodes, which need to store references to child nodes, and leaf nodes, which can omit those references to reduce space usage. As Figure 6g shows, by defining internal nodes (btree_internal) and leaf nodes (btree_leaf) to be of the same supertype btree, a user can specify that each child of an internal node may itself be another internal node or, alternatively, be a leaf node.

Finally, the node schema language allows users to specify that nodes store additional metadata, which may not be strictly needed to store nonzeros but are useful for other purposes. For instance,

```
1   st = {
2     node : blist
3   };
4
5   append_first(elem, st, ret):
6     blist* node = new blist;
7     node->e[0] = elem;
8     node->B = 1;
9     node->n = null;
10    ret->h = node;
11    st->node = node;
12
13  append_rest(elem, st):
14    blist* node = st->node;
15    if (node->B == 2) {
16      node = new blist;
17      node->B = 0;
18      node->n = null;
19      st->node->n = node;
20      st->node = node;
21    }
22    node->e[node->B] = elem;
23    node->B += 1;
```

(a) Append for block linked lists

```
1   build_rbt(elems, s, e):
2     if (s > e)
3       return null;
4     rbtree* node = new rbtree;
5     uint64 m = (s + e) / 2;
6     node->e = elems[m];
7     node->p = null;
8     node->c = (s + 1 == e);
9     if (s == e) {
10      node->l = null;
11      node->r = null;
12    } else if (s + 1 == e) {
13      node->l = build_rbt(elems, s, s);
14      node->r = null;
15      node->l->p = node;
16    } else {
17      node->l = build_rbt(elems, s, m - 1);
18      node->r = build_rbt(elems, m + 1, e);
19      node->l->p = node;
20      node->r->p = node;
21    }
22    return node;
23
24  build(elems, sz, ret):
25    ret->r = build_rbt(elems, 0, sz - 1);
```

(b) Bulk assembly for red-black trees

Fig. 8. Examples of how our assembly abstraction can be implemented for various dynamic data structures.

Figure 6e shows how a node in a red-black tree can be defined to store a reference to its parent (in field p) as well as store another field c that represents the node's color; these fields are needed to support efficient insertions into a red-black tree while keeping the tree balanced.

3.2 Assembly Abstract Interface

To support generating sparse tensor algebra kernels that store results in dynamic tensor formats, our technique additionally requires users to implement an abstraction that captures how dynamic data structures are efficiently assembled. Specifically, for any dynamic data structure, a user must specify how nonzeros can be individually *appended* to the data structure and/or specify how the data structure can be *bulk assembled* from a set of nonzeros.

Appends to a dynamic data structure are defined by two functions in our abstraction:

- append_first(elem, st, ret);
- append_rest(elem, st);

append_first defines how the first nonzero can be appended to the data structure, while append_rest defines how all subsequent nonzeros can be appended (in order of their coordinates, if the data structure is specified by sequence attributes to be sorted). Both functions take as arguments the nonzero to be appended (elem) as well as a reference to a user-defined object (st) that can be utilized to keep track of where exactly a nonzero was last appended in the data structure being assembled. Additionally, append_first takes as input a reference to a preallocated node (ret) that is intended to serve as a handle to the data structure being assembled. Figure 8a demonstrates how the append functions can be implemented for one specific dynamic data structure, namely block linked lists. To append the first new nonzero, append_first for block linked lists allocates a block, stores the nonzero at the beginning of the block, and initializes the root pointer (ret->h) to point to the block. For each subsequent new nonzero, append_rest then simply appends the new nonzero to the end

of the last allocated block (which is cached in st) unless the block is already full, in which case a new block is first allocated and attached to the end of the list.

Bulk assembly of a dynamic data structure, on the other hand, is defined by a single function:

- build(elems, sz, ret);

elems represents the sequence of nonzeros to be inserted, sz stores the size of elems, and ret is again a reference to a preallocated node that is intended to serve as a handle to the data structure being assembled. The argument elems implements an array interface, so any nonzero can be accessed by their position in the sequence. Additionally, if the data structure being assembled is specified to be sorted (i.e., if stored nonzeros are ordered by a sequence attribute), then the nonzeros in elems are guaranteed to be ordered by their coordinates. Figure 8b shows how a user can implement the build function for red-black trees. Bulk assembly can often be implemented more efficiently than appends. In the case of red-black trees, for instance, bulk assembly can be performed without needing to rebalance the tree after inserting each nonzero, which by contrast is needed when appending to red-black trees. Bulk assembly is also typically more amenable to parallelization; for example, the implementation of build in Figure 8b can be trivially parallelized by having recursive calls to build_rbt be spawned in parallel. However, bulk assembly requires the set of inserted nonzeros (elems) to be fully precomputed, which for some computations may incur additional overhead.

4 CODE GENERATION

In this section, we describe how we generalize the techniques of Kjolstad et al. [2019, 2017] and Chou et al. [2018] to generate efficient code that compute on tensors stored in arbitrary combinations of dynamic and static tensor formats. Like the technique of Kjolstad et al., which was summarized in Section 2.2, our technique takes as input a tensor algebra computation expressed in concrete index notation and recursively emits imperative (C++) code to iterate (or map) over each dimension of the operands. The remainder of this section will thus focus on how our technique emits code to efficiently compute on tensors along just one dimension. In particular, we show how our technique can use the abstractions we propose in Section 3 to generate code that may be optimized in very different ways for different computations and operand formats. The result is a system that reduces the effort needed to efficiently work with dynamic sparse tensors.

4.1 Generating Node Type Declarations

Before generating code to compute on dynamic sparse tensors, our technique first emits code to declare structs that represent nodes of dynamic data structures and that the generated compute code can actually work with. These structs are directly generated from node schemas, with one struct generated for each node schema. Figure 9 shows examples of structs that our technique generates for storing some of the dynamic data structures defined in Section 3.1.

Table 1 shows how our technique translates scalar fields in a node schema to fields in the corresponding struct. Array fields are translated in a similar way, except each emitted field is either an array member (e.g., int32_t f[4]) or a pointer into an array (e.g., int32_t* f). By default, array fields are translated to pointers into arrays that can be allocated separately from their containing struct. However, if an array field's size is either a constant N or upper-bounded by an in clause to be N, then the field is instead translated to an array member of size N. This reduces the overhead of accessing the array at run time by eliminating an indirection. Additionally, if a node only has one array field with an unbounded size, then our technique similarly translates the field to a zero-length array member[2] that stores its elements contiguously with the other fields of the node.

---

[2]Zero-length array members are not technically permitted in standard C++, though they are supported in practice by most C++ compilers (including GCC and LLVM for instance) as extensions.

```
struct bst {
  pair<int32_t,double> e;
  bst* r;
  bst* l;
};
```

(a) BST

```
struct blist {
  pair<int32_t,double> e[3];
  blist* n;
  int32_t B;
};
```

(b) Block linked list

```
struct btree {
  enum type { btree_internal, btree_leaf };
  type tp;
};

struct btree_internal : public btree {
  pair<int32_t,double> e[3];
  btree* cf;
  btree* cr[3];
  int32_t B;
};

struct btree_leaf : public btree {
  pair<int32_t,double> e[3];
  int32_t B;
};
```

(c) B-tree

Fig. 9. Examples of structs that our technique emits for storing various dynamic data structures.

Table 1. Translation of (scalar) fields in a node schema to fields in the corresponding emitted struct.

Field in Schema	Field in Emitted Struct	Notes
f : elem	pair<int32_t,V> f	First element of emitted pair stores coordinate of nonzero (or -1 if no nonzero is stored). Second element of emitted pair stores value of nonzero or pointer to data structure storing a subtensor.
f : node_type	node_type* f	
f : size	int32_t f	
f : parent	T* f	T is struct type being emitted (or its supertype, if applicable).
f : bool	bool f	
f : [u]intN	[u]intN_t f	$N \in \{8, 16, 32, 64\}$

Finally, to support having different types of node share a common supertype, our technique emits a struct for each supertype T, and all other emitted structs that correspond to T's subtypes inherit from T's struct. This struct contains a single member tp, which stores an enumeration that is intended for keeping track of a node's actual type at run time.

4.2 Generating Compute Code

From node schemas that describe how dynamic data structures store nonzeros, our technique can directly generate efficient code to compute on sparse tensors that are stored in those data structures. As we will see, such an approach enables our technique to generate code that are optimized in distinct ways for different computations and operand formats.

4.2.1 Generating Map Functions. If a computation on some tensor T is guaranteed to produce an output whose sparsity structure is a subset of T's sparsity structure, then the computation can be performed by simply mapping over and computing with each of T's nonzeros. This is always the case for multiplicative computations such as element-wise vector multiplication ($\forall_i a_i = b_i c_i$), since multiplication produces a non-zero result only if all operands are also non-zero. Thus, when one operand of a multiplicative computation is stored in a dynamic data structure while the rest are stored in formats (such as dense arrays) that support efficient random access of nonzeros, our technique emits code that map over the dynamic data structure to perform the computation.

```
emit_map(iv, expr, b):
  body := lower(expr, b)  # emit code to compute expr

  V := ...  # data type of elements in tensor b
  vars := ...  # variables referenced by body
  params := [", typeof(v) v" foreach v in vars].join()
  args := [", v" foreach v in vars].join()

  foreach schema in tensor b's node schemas:
    T := ...  # node type declared by schema
    emit        "void map_b(T* b params) {"
    emit          "if (b) {"
    if schema is a supertype:
      foreach aschema in tensor b's node schemas:
        S := ...  # node type declared by aschema
        if S is subtype of T:
          emit      "if (b->tp == T::S)"
          emit        "map_b((S*)b args);"
    else:
      foreach term in schema's sequence attribute:
        if term is a field:
          if term is an elem field:
            emit    "{"
            emit      "int32_t iv = b->term.first;"
            emit      "V v = b->term.second;"
            if term is not nonempty:
              emit    "if (iv != -1) {"
            emit        "body"
            if term is not nonempty:
              emit    "}"
            emit    "}"
          else:  # term is a child field
            if term is not nonempty:
              emit "if (b->term)"
            emit      "map_b(b->term args);"
        else:  # term is a set of array fields
          if size of array fields is constant N:
            bnd := N
          else:
            sz := ...  # field storing size of arrays
            bnd := "b->sz"
          emit        "for (int32_t p = 0; p < bnd; p++) {"
          foreach entry in term:
            if entry is an elem field:
              emit    "{"
              emit      "int32_t iv = b->entry[p].first;"
              emit      "V v = b->entry[p].second;"
              if entry is not nonempty:
                emit    "if (iv != -1) {"
              emit        "body"
              if entry is not nonempty:
                emit    "}"
              emit    "}"
            else:  # entry is a child field
              if entry is not nonempty:
                emit "if (b->entry[p])"
              emit      "map_b(b->entry[p] args);"
          emit        "}"
    emit          "}"
    emit        "}"
```

```
emit_iterator(b):
  V := ...  # data type of elements in tensor b

  foreach schema in tensor b's node schemas:
    T := ...  # node type declared by schema
    emit        "pair<int32_t,V> iter_T(T* b) {"
    emit          "if (b) {"
    if schema is a supertype:
      foreach aschema in tensor b's node schemas:
        S := ...  # node type declared by aschema
        if S is subtype of T:
          emit      "if (b->tp == T::S)"
          emit        "yield iter_S((S*)b);"
    else:
      foreach term in schema's sequence attribute:
        if term is a field:
          if term is an elem field:
            if term is not nonempty:
              emit "{"
              emit    "int32_t i = b->term.first;"
              emit    "if (i != -1)"
            emit        "yield b->term;"
            if term is not nonempty:
              emit "}"
          else:  # term is a child field
            if term is not nonempty:
              emit "if (b->term)"
            S := ...  # node type of term
            emit      "yield iter_S(b->term);"
        else:  # term is a set of array fields
          if size of array fields is constant N:
            bnd := N
          else:
            sz := ...  # field storing size of arrays
            bnd := "b->sz"
          emit        "for (int32_t p = 0; p < bnd; p++) {"
          foreach entry in term:
            if entry is an elem field:
              if entry is not nonempty:
                emit "{"
                emit    "int32_t i = b->entry[p].first;"
                emit    "if (i != -1)"
              emit        "yield b->entry;"
              if entry is not nonempty:
                emit "}"
            else:  # entry is a child field
              if entry is not nonempty:
                emit "if (b->entry[p])"
              S := ...  # node type of entry
              emit      "yield iter_S(b->entry[p]);"
          emit        "}"
    emit          "}"
    emit        "}"
```

Fig. 10. Algorithm for generating sequential code that maps over nonzeros in operand tensor b to compute the concrete index notation statement $\forall_{iv}\ expr$. The algorithm assumes that each node schema contains a sequence attribute; if there is not one, the algorithm emits code for each field in the order they are declared in the node schema. The algorithm also does not show tail call optimization applied.

Fig. 11. Algorithm for generating unoptimized iterators that enumerate nonzeros of any tensor stored in the same format as tensor b.

To generate sequential code that maps over a dynamic data structure and performs some specific computation on the stored nonzeros, our technique applies the algorithm shown in Figure 10. Our technique emits a map function for every type of node in the data structure. Figure 12a shows an example map function that our technique generates for mapping over nodes in a binary search

```
1   void map_b(bst* b, double* a, double* c) {
2     if (b) {
3       if (b->l)
4         map_b(b->l, a, c);
5       int32_t i = b->e.first;
6       a[i] = b->e.second * c[i];
7       if (b->r)
8         map_b(b->r, a, c);
9     }
10  }
```

(a) Sequential map over BST

```
1   void map_b(blist* b, double* a, double* c) {
2     while (b) {
3       for (int32_t p = 0; p < b->B; p++) {
4         int32_t i = b->e[p].first;
5         a[i] = b->e[p].second * c[i];
6       }
7       b = b->n;
8     }
9   }
```

(b) Sequential map over block linked list

```
1   void map_b(bst* b, double* a,
2              double* c, uint8_t d) {
3     if (b) {
4       if (d != 0) {
5         if (b->l)
6           #pragma omp task
7           map_b(b->l, a, c, d - 1);
8         if (b->r)
9           #pragma omp task
10          map_b(b->r, a, c, d - 1);
11        int32_t i = b->e.first;
12        a[i] = b->e.second * c[i];
13      } else {
14        map_b(b, a, c);
15      }
16    }
17  }
```

(c) Parallel map over BST

```
1   void map_b(blist* b, double* a, double* c) {
2     while (b) {
3       #pragma omp task
4       for (int32_t p = 0; p < b->B; p++) {
5         int32_t i = b->e[p].first;
6         a[i] = b->e[p].second * c[i];
7       }
8       b = b->n;
9     }
10  }
```

(d) Parallel map over block linked list

Fig. 12. Examples of map functions that our technique generates. Note that Cilk-parallelized code can be similarly generated by replacing OpenMP pragmas with Cilk keywords.

tree and computing element-wise vector multiplication. Each emitted function accesses all elem fields in the input node and, for each stored nonzero, performs the specified computation with the nonzero (lines 5–6 in Figure 12a). If the input node contains references to child nodes, then the emitted function must also compute on nonzeros that are stored in those child nodes (and their descendants). In the general case, this is done by (recursively) invoking the appropriate map function to process the child nodes (lines 3–4 and 7–8 in Figure 12a). By default, as demonstrated in Figure 12a, our technique emits sequential code that computes on stored nonzeros in coordinate order as specified by the input node's sequence attribute. However, if the input node does not provide a sequence attribute, then our technique instead simply emits code that processes the input node's fields in the order they are declared in the node schema.

The above approach generates correct code for any dynamic data structure that can be expressed using the node schema language. However, for data structures that do not exhibit any fanout (i.e., those comprised of nodes that have exactly one child each, such as linked lists), code that the above approach generates is prone to fail due to stack overflow when the input data structure contains many nodes. Thus, for any type of node that has exactly one child of the same type, our technique applies tail call optimization to instead emit a map function that uses a loop to iterate over all of the input node's descendants. (Our technique can trivially determine if a type of node has exactly one child of the same type by inspecting its schema.) Figure 12b shows an example of code that our technique emits for mapping over a block linked list using this approach.

Our technique also emits parallelized map functions in a similar way as sequential map functions. For any input node that has exactly one child of the same type though (e.g., block linked list nodes), our technique emits code to process the node and its descendants in parallel by spawning tasks that each computes on a single node's stored nonzeros (lines 3–7 in Figure 12d). Meanwhile, for any other type of input node, our technique emits code to process the node's descendants in parallel by spawning tasks that each maps over a single child node and its descendants (lines 5–10 in Figure 12c). To avoid spawning too many fine-grained tasks, the emitted code keeps track of the depth of recursion (parameter d in Figure 12c) and, once a certain depth has been reached, switches back to a sequential version of the map function (lines 13–14 in Figure 12c).

Finally, to support mapping over nodes that share a common supertype but whose actual types are not known at compile time, our technique additionally emits a map function for every supertype. Each such map function simply checks the input node's type at run time and then invokes the corresponding map function to actually compute on the input node. So to map over a child of a B-tree node, for instance, the generated code would invoke a map function that takes any instance of btree as input. This function would then invoke a second map function (which performs the actual computation) that only takes an instance of either btree_internal or btree_leaf as input, depending on if the child is an internal node (i.e., if tp == btree::btree_internal) or a leaf node.

4.2.2 Generating Iterators. In general though, computing a sparse tensor algebra operation may require simultaneously iterating over multiple operands that are all stored in dynamic data structures. Such computations are not readily supported by recursive map functions like those described previously. So to support these computations, our technique instead emits code that uses some set of loops to iterate over intersections or unions of the operands' nonzeros and compute on those nonzeros. Figure 2c shows an example of such code. Kjolstad et al. [2017] and Henry et al. [2021] describe how a compiler can generate loops that iterate over intersections or unions of sparse tensor operands, assuming it is possible to enumerate the stored nonzeros of each operand. However, while Chou et al. [2018] show how code that performs such enumeration can be emitted for operands that are stored in static, array-based formats, their technique does not support—and does not readily generalize to—dynamic, pointer-based sparse tensor formats.

To generate an iterator that efficiently enumerates stored nonzeros in a dynamic data structure, our technique first mechanically emits a (recursive) coroutine for every type of node that may be contained in the data structure. This is done by applying the algorithm in Figure 11. Figure 13a shows an example coroutine that our technique generates for iterating over a BST. Each emitted coroutine accesses all of the input node's stored nonzeros and child nodes in the order specified by the input node's sequence attribute. For each nonzero, the emitted code simply yields the coordinate and value of that nonzero (line 5 in Figure 13a). For each child node, on the other hand, the emitted code (recursively) invokes the appropriate coroutine to yield all nonzeros that are stored in the child node and its descendants (lines 3–4 and 6–7 in Figure 13a).

Our technique then applies a set of optimizations to each emitted coroutine in order to obtain an optimized iterator. First, our technique applies tail call optimization in order to reduce the number of recursive calls. Additionally, if the input node has child nodes of other types, all invocations of iterators for those nodes are inlined, yielding a coroutine that only has recursive calls to itself. Then, to eliminate the overhead of recursive calls to a coroutine, our technique rewrites the coroutine so that it emulates recursion using a loop with an explicit call stack. This stack stores the local variables and state of each recursive call. Finally, to obtain code that does not rely on language support for coroutines (and can thus be compiled with pre-C++20 compilers or even straightforwardly translated to C), our technique rewrites the coroutine to a function that, when invoked, yields the next nonzero's coordinate and value as output parameters. Figure 13 show how our technique

```
1  pair<int32_t,double> iter_bst(bst* n) {
2    if (n) {
3      if (n->l)
4        yield iter_bst(n->l);
5      yield n->e;
6      if (n->r)
7        yield iter_bst(n->r);
8    }
9  }
```

(a) Unoptimized iterator

```
1  pair<int32_t,double> iter_bst(bst* n) {
2    while (n) {
3      if (n->l)
4        yield iter_bst(n->l);
5      yield n->e;
6      n = n->r;
7    }
8  }
```

(b) After tail call optimization

```
1  pair<int32_t,double> iter_bst(bst* n) {
2    call_stack<uint8_t,bst*> cs;
3    cs.emplace(0, n);
4    while (!cs.empty()) {
5      n = get<1>(cs.top());
6      if (get<0>(cs.top()) == 1)
7        goto call_resume1;
8      while (n) {
9        if (n->l) {
10         get<0>(cs.top()) = 1;
11         get<1>(cs.top()) = n;
12         cs.emplace(0, n->l);
13         goto call_end;
14  call_resume1:;
15       }
16       yield n->e;
17       n = n->r;
18     }
19     cs.pop();
20  call_end:;
21   }
22  }
```

(c) After recursion elimination

Fig. 13. Steps involved in generating an optimized iterator for BSTs. The final code is shown in Figure 2c.

applies these optimizations to the unoptimized code in Figure 13a in order to generate an efficient iterator for BSTs, which is shown in Figure 2c (i.e., the function iter_bst). Then, by applying the techniques of Kjolstad et al. and Henry et al., our technique can emit code that uses the generated iterator to iterate over a BST simultaneously with any other dynamic—or even static—data structure, such as a block linked list (Figure 2c) or an array-based sparse vector (Figure 15).

4.2.3 Selecting Between Map Functions and Iterators. Unlike the recursive map functions described in Section 4.2.1, iterators that our technique generates can be used to compute any sparse tensor algebra operation. This includes all computations that are readily supported by recursive map functions (i.e., multiplicative operations with only one sparse tensor operand). However, we find that such computations can generally be performed more efficiently with recursive map functions that our technique generates. For one thing, our technique can often emit parallelized map functions to perform these computations, whereas iterators that our technique generates are inherently sequential. For another, as we show in Section 5.4, iterators that our technique generates may incur non-negligible performance overhead even when compared to sequential map functions. Thus, only for computations that cannot be readily implemented using recursive map functions (i.e., those that compute on multiple sparse operands) does our technique emit code that utilizes iterators.

4.3 Generating Assembly Code

In addition to generating code that compute on operands stored in dynamic tensor formats, our technique can emit code to store the results of those computations in the same dynamic tensor formats as well. This is achieved in several ways.

If a computation can be performed with a map function and if the result is stored in the same format as the input tensor being mapped over, then our technique emits code that assembles the result by essentially deeply copying the input data structure. This approach is valid since each nonzero in the result is computed from one nonzero in the input tensor being mapped over, so our technique can infer that the output data structure must have the same structure as the input data

```
1   bst* map_b(bst* b, double* c) {
2     if (b) {
3       bst* ret = new bst;
4       ret->l = NULL;
5       if (b->l)
6         ret->l = map_b(b->l, c);
7       int32_t i = b->e.first;
8       ret->e.first = i;
9       ret->e.second = b->e.second * c[i];
10      ret->r = NULL;
11      if (b->r)
12        ret->r = map_b(b->r, c);
13      return ret;
14    }
15    return NULL;
16  }
```

Fig. 14. Example emitted code that multiplies a sparse vector stored as a BST by a dense vector. The code stores the result in another BST by making a deep copy of the sparse input vector.

```
1   bool afirst = true;
2   uint8_t bs = iter_bst(0, bn, bstack, ib, bv);
3   int32_t pc = c_pos[0];
4   while (bs && pc < c_pos[1]) {
5     int32_t ic = c_crd[pc];
6     int32_t i = min(ib, ic);
7     if (i == ib && i == ic) {
8       double av = bv * c[pc];
9       if (afirst) {
10        aret = new blist_head;
11        append_first({i, av}, ast, aret);
12        afirst = false;
13      } else {
14        append_rest({i, av}, ast);
15      }
16    }
17    if (i == ib) bs = iter_bst(bs, ..., ib, bv);
18    pc += (i == ic);
19  }
```

Fig. 15. Example emitted code that multiplies a sparse vector stored as a BST by an array-based sparse vector. The code invokes append_first and append_rest to store result nonzeros. Our technique can further specialize this code for block linked list outputs by inlining implementations of the append functions for block linked lists.

structure. Figure 14 shows an example map function that our technique generates, which computes on an input tensor stored as a BST and which stores the result as another BST. Such map functions can be generated in largely the same way as described in Section 4.2.1. To make a deep copy of the input data structure though, each emitted map function additionally allocates and returns a new node that is of the same type as the input node (lines 3 and 13 in Figure 14). This new node is initialized by copying over the coordinates of nonzeros that are stored in the input node (line 8 in Figure 14), with the corresponding values initialized to be the results of the computation (line 9 in Figure 14). Furthermore, new output child nodes are allocated by invoking the augmented map function(s) on the input node's children (lines 6 and 12 in Figure 14).

In general though, a tensor algebra kernel may have to assemble a dynamic data structure from scratch to store the result. By using the abstraction we propose in Section 3.2, our technique can generate such code without needing to hard-code for any specific data structure. Specifically, to generate code that stores the result of a computation in a dynamic data structure, our technique first emits code that invokes the assembly functions described in Section 3.2 to store the result nonzeros. Then, the emitted code is specialized to a specific type of dynamic data structure by inlining its implementation of the assembly functions. So to generate code that stores the result of a tensor algebra computation in a block linked list, for instance, our technique emits code like what is shown in Figure 15, which invokes the append_first and append_rest functions to store the result nonzeros. The code generator can then inline implementations of append_first and append_rest for block linked lists (as shown in Figure 8a) into the emitted code, yielding code that is specialized for block linked list outputs. On the other hand, if a computation simply assigns an input tensor to the output and if the input is stored in an array-based format, our technique can emit code that invokes the build function (with a reference to the input as the argument elems) to bulk assemble the output tensor. The code generator can then inline any dynamic tensor format's implementation of build in order to obtain code that bulk assembles the output in the aforementioned format.

5 EVALUATION

We implement our technique as a prototype extension to the TACO sparse tensor algebra compiler, and we find it generates sparse tensor algebra kernels that efficiently compute on operands stored in dynamic tensor formats. Code that our technique generates have performance comparable to, if not better than, equivalent code that can be implemented using hand-optimized frameworks and libraries. At the same time, our technique can generate code for many tensor algebra computations that are not readily supported by most of the aforementioned frameworks, including kernels that simultaneously compute with static and dynamic sparse tensors for instance.

5.1 Experiment Setup

We evaluate code that our technique generates against four state-of-the-art frameworks and library: Aspen [Dhulipala et al. 2019], Terrace [Pandey et al. 2021], STINGER [Ediger et al. 2012], and PAM [Sun et al. 2018]. Aspen and Terrace are graph processing frameworks that let users compute on dynamic graphs by invoking a fixed set of hand-optimized primitives for mapping over and applying user-defined functions on edges and vertices. Internally, Aspen stores adjacency matrices of graphs using C-trees, while Terrace instead uses a combination of fixed-size arrays, packed-memory arrays, and B-trees. STINGER is another graph processing framework that supports computations on dynamic graphs; STINGER stores graphs using block linked lists and provides a set of macros that programmers can use to iterate over (and compute on) edges and vertices. PAM, by contrast, is a lower-level, parallel C++ library that implements a fixed set of primitives for operating on ordered key-value maps stored as self-balancing BSTs. While PAM does not directly implement any tensor algebra kernel, the primitives that PAM exposes can be utilized to compute on sparse tensors that are stored using BSTs.

We run our experiments on a two-socket, 12-core/24-thread 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache per socket and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS. We compile all code using GCC 7.5.0 with `-O3 -march=native -mtune=native -ffast-math` optimizations enabled. To ensure apples-to-apples comparisons of the actual algorithms that are implemented by the (generated and hand-optimized) kernels, we modify code generated by our technique so that they operate on identical data structures in memory as the libraries we compare against. This only requires minor changes to how the fields of input data structures are accessed and does not entail any algorithmic change. Additionally, we use the same parallel programming APIs as the libraries we compare against (i.e., OpenMP for STINGER and Cilk for the others), and all memory allocations are done using `jemalloc`. We run each experiment 100

Table 2. Statistics about real-world matrices used in our experiments.

	Matrix	Dimensions	NNZ
1	belgium_osm	1.44M × 1.44M	3.10M
2	cit-Patents	3.77M × 3.77M	16.5M
3	coAuthorsCiteseer	227K × 227K	1.63M
4	coPapersDBLP	540K × 540K	30.5M
5	com-Orkut	3.07M × 3.07M	234M
6	delaunay_n24	16.8M × 16.8M	101M
7	indochina-2004	7.41M × 7.41M	194M
8	rgg_n_2_24_s0	16.8M × 16.8M	265M
9	roadNet-CA	1.97M × 1.97M	5.53M
10	road_central	14.1M × 14.1M	33.9M
11	road_usa	23.9M × 23.9M	57.7M
12	ship_003	122K × 122K	3.78M
13	soc-LiveJournal1	4.85M × 4.85M	69.0M
14	webbase-1M	1.00M × 1.00M	3.11M

times under cold cache conditions and report median execution times. Each experiment is run using 24 threads, with execution restricted to a single socket using `numactl`.

We run our experiments with real-world sparse matrices of varying sizes from the SuiteSparse Matrix Collection [Davis and Hu 2011]. These matrices, which Table 2 describes in more detail, represent graphs and other data that arise in disparate application domains.

5.2 Support for Disparate Formats

Our technique generates efficient code to compute on dynamic sparse tensors that are stored in a wide range of disparate formats. To demonstrate this, we evaluate the performance of code that our technique generates for performing

- the **PageRank** kernel ($\forall_i \forall_j \ y_i \ +\!= A_{ij} x_j d_j^{-1}$) and
- **sparse-dense matrix multiplication** ($\forall_i \forall_j \forall_k \ C_{ik} \ \|\!= A_{ij} B_{jk}$, or **SpMM**)

where A is a dynamic sparse binary matrix (representing a graph's adjacency matrix), B and C are dense binary matrices, x and y are dense floating-point vectors, and d is a dense integer vector. In particular, the first kernel corresponds to the main kernel in each iteration of the PageRank algorithm [Page et al. 1998], while the second kernel can be used to implement algorithms such as multi-source breadth-first search [Acer et al. 2016]. For each kernel, we measure the performance of code that our technique generates for A stored in various dynamic tensor formats (specified in terms of level formats that are defined in Figures 5 and 6), including

- using only **BSTs** in the (bst, bst) format,
- using **C-trees** in the (bst, ctree) format,
- using **block linked lists** (**BLLs**) in the (dense, blist) format, and
- using a **hybrid of fixed-size arrays and B-trees** in the (dense, hybrid) format;

these correspond to formats that are supported by PAM, Aspen,[3] STINGER, and Terrace[4] respectively. We then compare the generated code against the aforementioned frameworks and library. In particular, Aspen and Terrace both implement an edgeMap primitive that can compute the PageRank and SpMM kernels by mapping over edges in the input graph (i.e., nonzeros in A) and performing some (user-defined) computation on each edge/nonzero. Similarly, STINGER provides macros that can be used to efficiently iterate over incident edges of each vertex (i.e., nonzeros in each row of A) in order to perform the same computations. Additionally, PAM can compute the same kernels in similar ways by mapping over the rows of A (using the map_void primitive) and performing either a map-reduce operation (for PageRank, using the semi_map_reduce primitive) or another map operation (for SpMM) over the nonzeros in each row.

Tables 3 and 4 show the results of our experiments. Our technique is the only one that supports all of the dynamic sparse tensor formats we consider; the other frameworks and library we evaluate each only supports a single format. Nevertheless, as Tables 3 and 4 demonstrate, our technique is able to achieve comparable, if not better, performance as all the hand-optimized frameworks and library. In particular, code that our technique generates for computing on C-trees and block linked lists have similar performance as Aspen and STINGER, with the generated code being 1.144–1.177× faster than Aspen and 1.002–1.124× faster than STINGER on average. Additionally though, code that our technique generates for computing on B-trees outperform Terrace by 1.214–1.551× on average, while code that our technique generates for computing on BSTs outperforms PAM by 1.202–1.355× on average. It is not surprising that our technique achieves similar performance as Aspen and STINGER, since code that our technique generates essentially implement the same high-level algorithms as Aspen and STINGER. Meanwhile, Terrace is slower than our technique since its implementation of edgeMap traverses B-trees using a sequential iterator that has more complicated control flow, which increases the cost of accessing each nonzero. By contrast, for

---

[3]While Aspen also supports C-trees that use difference encoding to compress the coordinates stored in each block, we only evaluate our technique and Aspen on C-trees that do not use difference encoding, since difference encoding is not supported by our technique as we have described it.

[4]Terrace supports packed-memory arrays (PMAs) in addition to fixed-size arrays and B-trees, though we omit PMAs from our evaluation since they are not supported by our technique.

Table 3. Performance of code implemented using existing libraries and generated by our technique (TACO) for computing the PageRank kernel on inputs stored in disparate dynamic sparse tensor formats. For each format, we show execution times of code implemented using an existing library that supports the format, execution times of code generated by our technique, and speedups achieved by the generated code. Each test matrix is identified by its label as shown in Table 2.

Matrix	BSTs			C-trees			BLLs			Array/B-tree Hybrid		
	PAM (ms)	TACO (ms)	$\frac{PAM}{TACO}$	Aspen (ms)	TACO (ms)	$\frac{Aspen}{TACO}$	STINGER (ms)	TACO (ms)	$\frac{STINGER}{TACO}$	Terrace (ms)	TACO (ms)	$\frac{Terrace}{TACO}$
1	9.876	5.508	1.793	6.970	6.310	1.105	15.05	14.18	1.061	3.080	2.783	1.107
2	61.51	43.53	1.413	49.23	36.96	1.332	63.77	56.96	1.120	40.46	32.28	1.253
3	3.142	2.506	1.254	1.712	2.018	0.848	3.314	2.951	1.123	1.766	1.100	1.605
4	31.80	27.70	1.148	14.99	12.30	1.219	34.87	28.89	1.207	26.96	14.36	1.877
5	504.0	449.7	1.121	378.7	276.4	1.370	494.3	403.3	1.226	471.5	288.0	1.637
6	143.8	111.6	1.288	85.48	76.36	1.119	218.4	199.0	1.098	46.12	42.40	1.088
7	188.7	181.6	1.040	90.45	70.85	1.277	195.6	171.5	1.141	529.9	164.6	3.220
8	378.9	265.8	1.426	163.0	147.1	1.108	435.5	355.5	1.225	306.7	148.1	2.071
9	12.54	7.817	1.604	8.852	8.089	1.094	24.19	20.35	1.189	3.920	3.582	1.094
10	172.9	98.67	1.752	117.5	96.22	1.221	192.5	184.7	1.042	71.48	62.27	1.148
11	166.1	95.12	1.746	118.1	106.5	1.109	269.1	258.7	1.040	58.86	52.32	1.125
12	7.044	6.734	1.046	3.613	3.046	1.186	6.076	5.751	1.056	6.868	3.638	1.888
13	156.2	124.1	1.259	103.2	80.12	1.288	176.7	149.7	1.180	157.3	79.48	1.979
14	6.715	4.811	1.396	5.220	4.005	1.303	9.224	8.707	1.059	4.879	2.738	1.782
Geomean			1.355			1.177			1.124			1.551

both PageRank and SpMM, our technique can generate code that instead recursively traverses the B-trees, which reduces the overhead of accessing nonzeros.

5.3 Support for Disparate Computations

Not only does our technique support many disparate formats, our technique can generate efficient code to compute a wide range of operations (in addition to those evaluated in Section 5.2) on dynamic sparse tensors that are stored in those formats. To demonstrate this, we evaluate the performance of code that our technique generates for performing

- **sparse matrix-vector multiplication** ($\forall_i \forall_j \ y_i \mathrel{+}= A_{ij}x_j$, or **SpMV**)

where A is a dynamic sparse matrix stored in the (bst, bst) format, x is a dense vector, and y is a dynamic sparse vector stored as a BST, as well as

- **sparse matrix addition** ($\forall_i \forall_j \ D_{ij} = A_{ij} + C_{ij}$, or **SpAdd**),
- **sparse matrix element-wise multiplication** ($\forall_i \forall_j \ D_{ij} = A_{ij}C_{ij}$, or **SpElwiseMul**), and
- **row-wise inner product** ($\forall_i \forall_j \ z_i \mathrel{+}= A_{ij}B_{ij}$, or **RowInnerProd**)

where A and B are dynamic sparse matrices stored in the (dense, bst) format, C and D are static sparse matrices stored in CSR, and z is a dense vector.

None of the kernels above are readily supported by Aspen, STINGER, or Terrace, regardless of what format is used to store the dynamic sparse matrix operands. (More generally, the three frameworks do not readily support computations that have dynamic sparse tensor outputs, that simultaneously work with dynamic and static sparse tensors, or that perform non-element wise operations on multiple dynamic sparse tensors.) By contrast, PAM, which is a lower-level library, implements a number of primitives that can be utilized to compute all of the kernels above. In particular, PAM can be used to compute sparse matrix-vector multiplication in a similar way as

Table 4. Performance of code implemented using existing libraries and generated by our technique (TACO) for computing SpMM on inputs stored in disparate dynamic sparse tensor formats. For each format, we show execution times of code implemented using an existing library that supports the format, execution times of code generated by our technique, and speedups achieved by our generated code. Each test matrix is identified by its label as shown in Table 2.

Matrix	BSTs			C-trees			BLLs			Array/B-tree Hybrid		
	PAM (ms)	TACO (ms)	$\frac{PAM}{TACO}$	Aspen (ms)	TACO (ms)	$\frac{Aspen}{TACO}$	STINGER (ms)	TACO (ms)	$\frac{STINGER}{TACO}$	Terrace (ms)	TACO (ms)	$\frac{Terrace}{TACO}$
1	23.40	16.95	1.380	20.19	17.06	1.184	27.19	29.33	0.927	16.23	14.60	1.112
2	163.4	153.3	1.066	138.3	126.2	1.095	147.9	150.2	0.985	146.8	124.6	1.178
3	11.21	9.839	1.140	8.825	8.374	1.054	11.85	11.47	1.033	9.378	8.408	1.115
4	122.5	99.96	1.225	87.24	78.24	1.115	108.4	102.0	1.062	109.8	90.58	1.213
5	1056	910.1	1.161	769.0	676.8	1.136	947.3	928.2	1.021	900.0	759.1	1.186
6	846.6	704.8	1.201	721.3	660.1	1.093	759.3	753.6	1.008	703.0	652.7	1.077
7	771.8	691.4	1.116	585.1	559.9	1.045	674.6	627.3	1.075	2295	963.1	2.383
8	1823	1634	1.115	1425	1346	1.058	1601	1558	1.028	1620	1388	1.167
9	43.82	33.83	1.295	37.20	32.73	1.136	48.09	51.74	0.930	33.53	30.65	1.094
10	344.4	282.0	1.222	302.4	269.9	1.121	337.8	368.4	0.917	252.1	227.9	1.106
11	457.1	338.3	1.351	391.8	329.5	1.189	477.9	516.1	0.926	333.5	300.4	1.110
12	30.13	25.35	1.189	21.68	20.40	1.063	28.93	26.61	1.087	28.24	22.62	1.248
13	413.6	366.0	1.130	320.3	288.9	1.109	374.0	370.8	1.009	401.2	307.1	1.307
14	17.55	13.70	1.281	20.21	11.46	1.764	17.92	17.29	1.037	18.70	17.07	1.096
Geomean			1.202			1.144			1.002			1.214

the PageRank kernel, except that the map operation (map) over A's rows also constructs a new BST to store the nonzeros of the output vector.[5] Meanwhile, sparse matrix addition and element-wise multiplication can be computed row by row by having PAM first convert each row of A to a BST, then compute the union/intersection of this BST with the corresponding row in C (using map_union/map_intersect), and finally map over the result (which is also stored in a BST) to store result nonzeros in D (using foreach_index). Additionally, row-wise inner product can be computed by having PAM first compute the intersection of each row of A with its corresponding row in B and then perform a map-reduce over the result in order to compute the corresponding element in z. We therefore limit our comparisons to PAM and do not consider the other frameworks.

Table 5 shows the results of our experiments. As these results demonstrate, our technique generates code that significantly outperforms PAM for all of the kernels we evaluate. In particular, code that our technique generates for adding a dynamic sparse matrix to a static sparse matrix outperforms PAM by 6.975× on average. Meanwhile, code that our technique generates for element-wise multiplying a dynamic sparse matrix by a static sparse matrix outperforms PAM by 7.224× on average. PAM incurs significant overhead for these computations since the library only supports computing unions and intersections of BSTs. As a result, performing these computations using PAM requires allocating new nodes in order to convert the input matrix C to BSTs and also to actually perform the union/intersection operations. Moreover, PAM incurs additional overhead in order to copy nonzeros that are computed by the union/intersection operations over to the output matrix D. By contrast, our technique emits code that directly performs the computation without needing to convert to and from BST temporaries, thereby reducing memory traffic. This shows the

---

[5]While PAM uses a custom pool allocator to allocate new BST nodes by default, we modify PAM for our experiments so that it simply uses malloc to allocate new nodes. We find that, for our benchmarks, this slightly improves PAM's performance and also yields more repeatable performance results.

Table 5. Performance of code implemented using PAM and generated by our technique (TACO) for computing disparate sparse tensor algebra operations on inputs stored using BSTs. For each operation, we show execution times of code implemented using PAM, execution times of code generated by our technique, and speedups achieved by our generated code. Each test matrix is identified by its label as shown in Table 2.

Matrix	SpMV			SpAdd			SpElwiseMul			RowInnerProd		
	PAM (ms)	TACO (ms)	$\frac{PAM}{TACO}$	PAM (ms)	TACO (ms)	$\frac{PAM}{TACO}$	PAM (ms)	TACO (ms)	$\frac{PAM}{TACO}$	PAM (ms)	TACO (ms)	$\frac{PAM}{TACO}$
1	8.78	7.411	1.185	94.65	11.50	8.233	79.46	8.642	9.195	20.95	5.929	3.533
2	54.29	44.30	1.225	390.6	60.65	6.440	304.2	43.62	6.974	113.1	28.82	3.925
3	3.101	2.734	1.134	39.11	5.302	7.377	34.29	4.278	8.016	12.68	3.048	4.161
4	29.73	27.58	1.078	606.5	79.46	7.633	534.1	70.08	7.622	233.0	51.73	4.503
5	375.4	344.9	1.088	5097	653.1	7.805	3516	443.9	7.920	1547	385.1	4.017
6	156.1	132.0	1.182	2162	322.1	6.712	1761	261.6	6.733	614.1	169.1	3.633
7	193.9	188.4	1.029	4262	827.1	5.153	4018	790.5	5.083	1540	497.1	3.098
8	355.1	271.0	1.310	5057	769.2	6.574	3732	511.6	7.294	1592	428.8	3.713
9	11.86	10.48	1.132	148.9	22.55	6.601	123.2	18.24	6.754	35.02	10.13	3.457
10	169.9	115.4	1.473	964.8	144.0	6.698	804.6	111.2	7.234	220.1	61.95	3.553
11	180.8	132.5	1.365	1629	240.5	6.774	1364	192.9	7.073	371.4	105.3	3.528
12	7.040	6.944	1.014	174.4	19.19	9.092	169.1	18.38	9.201	71.26	13.38	5.327
13	132.0	110.9	1.190	1501	207.5	7.233	1137	154.9	7.345	463.4	115.5	4.012
14	6.256	5.511	1.135	100.6	16.28	6.180	81.95	14.06	5.829	25.07	8.645	2.900
Geomean			1.175			6.975			7.224			3.770

benefits of a compiler technique such as ours that can generate efficient code to directly compute on both static and dynamic sparse tensors.

Furthermore, code that our technique generates to compute the row-wise inner product of two dynamic sparse matrices outperforms PAM by 3.770× on average. PAM again incurs significant overhead since it does not support directly performing a map-reduce operation over the output of an intersection operation. As a result, computing an inner product requires performing the intersection and the map-reduce separately. By contrast, our technique can generate more efficient code that effectively fuses the intersection and the map-reduce. This demonstrates the benefits of a system that does not rely on hand-optimized code to perform a bounded set of operations.

5.4 Analysis of Generated Code

Table 6 reports the size of code that our technique generates as well as the time required to compile each generated kernel using GCC (with optimizations enabled).

In addition, we run several experiments to evaluate the effectiveness of optimizations that our technique applies to generated code. We focus on the PageRank kernel with input matrices stored

Table 6. Number of lines in the generated code (including 21 lines of shared boilerplate), size of the corresponding compiled object file, and compilation time for each kernel evaluated in Sections 5.2 and 5.3.

Kernel	LOC	Object File Size (bytes)	Compilation Time (ms)
PageRank (BSTs)	87	7496	290.5
PageRank (C-trees)	118	20376	509.8
PageRank (BLLs)	54	2544	236.6
PageRank (Hybrid)	109	23184	578.6
SpMM (BSTs)	88	8072	317.2
SpMM (C-trees)	123	7192	292.2
SpMM (BLLs)	61	3304	245.9
SpMM (Hybrid)	123	6824	331.0
SpMV	111	7184	284.8
SpAdd	186	5024	311.7
SpElwiseMul	153	4416	292.1
RowInnerProd	168	3368	273.0

Table 7. Performance of PageRank kernels that map over input matrices out of order (by performing pre-order traversals) and in order. We show execution times of both kernels and speedups achieved by the in-order approach. Each test matrix is identified by its label as shown in Table 2.

Table 8. Performance of PageRank kernels that rely on generated iterators and generated recursive map functions. We show execution times of both kernels and speedups achieved by the recursive map function approach. Each test matrix is identified by its label as shown in Table 2.

Matrix	Unordered (ms)	Ordered (ms)	$\frac{\text{Unordered}}{\text{Ordered}}$
1	6.233	5.508	1.132
2	44.48	43.53	1.022
3	2.735	2.506	1.092
4	29.03	27.70	1.048
5	441.5	449.7	0.982
6	119.8	111.6	1.073
7	186.2	181.6	1.026
8	332.2	265.8	1.250
9	8.605	7.817	1.101
10	104.3	98.67	1.057
11	106.0	95.12	1.114
12	6.774	6.734	1.006
13	127.6	124.1	1.028
14	4.698	4.811	0.977
Geomean			1.063

Matrix	w/ Iterator (ms)	w/ Map Func. (ms)	$\frac{\text{w/ Iterator}}{\text{w/ Map Func.}}$
1	6.688	5.508	1.214
2	48.35	43.53	1.111
3	2.727	2.506	1.088
4	27.87	27.70	1.006
5	449.5	449.7	1.000
6	112.7	111.6	1.010
7	181.8	181.6	1.001
8	266.9	265.8	1.004
9	8.491	7.817	1.086
10	113.4	98.67	1.149
11	105.6	95.12	1.110
12	6.850	6.734	1.017
13	129.0	124.1	1.040
14	4.833	4.811	1.005
Geomean			1.058

using BSTs as a representative example. In particular, we compare code that our technique generates, which maps over nonzeros in each row of the input matrix in coordinate order, against code that instead pre-order traverses the BSTs to perform the computation. As Table 7 shows, generating code to compute on stored nonzeros in coordinate order can yield speedups of 1.063× on average, since this reduces cache misses when accessing the input vectors. Furthermore, we compare code that our technique generates, which computes on input matrices using only recursive map functions, against code that instead iterates over input matrices using iterators like those shown in Figure 2c. As Table 8 shows, by generating recursive map functions to perform the computation where possible, our technique can yield speedups of 1.058× on average. This emphasizes the need for a compiler to be able to emit code that access dynamic sparse tensors in different ways for different computations.

Finally, we evaluate the overhead of having recursive function calls in generated code by measuring the performance of code that our technique generates for computing the PageRank kernel, compiled with and without GCC performing recursive function inlining. We find that recursive function inlining has negligible impact on performance, with the generated code being only 0.56% faster on average when recursive function inlining is enabled. This suggests that recursive function calls are not the main bottlenecks in code our technique generates.

5.5 Benefits of Supporting Disparate Formats

To be able to effectively support many different types of applications, a general-purpose system for computing with sparse tensors must be able to efficiently work with tensors that are stored in a wide variety of formats. To demonstrate this, we compare the cost of computing on and modifying sparse tensors that are stored using BSTs in the (bst, bst) format, using C-trees in the (bst, ctree) format, and in the CSR format. To quantify the cost of computing on sparse tensors that are stored in the aforementioned formats, we measure the performance of code that our technique generates for computing the PageRank kernel. To quantify the cost of modifying sparse tensors that are
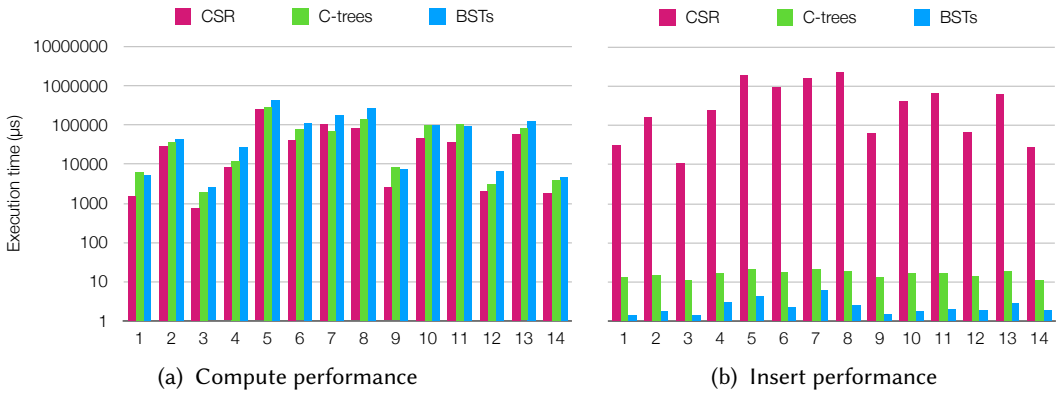
Fig. 16. Time required to (a) compute the PageRank kernel on matrices stored in various formats and (b) insert a new nonzero into matrices stored in the same formats. Labels along the horizontal axis identify the input matrices as listed in Table 2.

stored in the same formats, we adapt the method used by Dhulipala et al. [2019] and randomly sample 1% of nonzeros in each test matrix to treat as new nonzeros that need to be inserted. We then measure the performance of optimized routines that are implemented in PAM, Aspen, and Eigen (a widely-used sparse linear algebra library that supports CSR [Guennebaud et al. 2010]) for inserting the sampled nonzeros individually into the tensor.

Figure 16 shows the results of our experiments. As these results illustrate, the performance of compute on sparse tensors that are stored in a particular format is often anti-correlated with the performance of modification on sparse tensors that are stored in the same format. In particular, as Figure 16a shows, computing the PageRank kernel on CSR matrices is 1.788× faster on average than computing the same kernel on matrices stored using C-trees, which in turn is 1.404× faster on average than computing on matrices stored using BSTs. On the other hand, as Figure 16b shows, inserting new nonzeros into matrices stored using BSTs is 6.893× faster on average than inserting new nonzeros into matrices stored using C-trees, which in turn is orders of magnitudes faster than inserting new nonzeros into CSR matrices. These results show how, depending on the relative proportion of data modification and compute, different applications can benefit from using different (dynamic or static) sparse tensor formats. This means a system intended to be general-purpose should ideally support a wide range of disparate formats.

6 RELATED WORKS

As Section 2.1 shows, there exists a long line of works on using dynamic data structures to efficiently represent adjacency matrices of dynamic graphs. In addition to the libraries and frameworks identified in Section 2.1, LLAMA [Macko et al. 2015] uses a data structure that resembles variable block linked lists to store elements of a dynamic graph's adjacency matrix, except each block stores elements corresponding to multiple rows. There are also various GPU libraries and frameworks that store dynamic graphs using variants of either the data structures shown in Figure 3 [Awad et al. 2020; Winter et al. 2017] or other data structures that can be expressed using our proposed abstractions [Busato et al. 2018]. Furthermore, a number of works [King et al. 2016; Sengupta and Song 2017] have explored using array-based data structures, including packed memory arrays [Pandey et al. 2021; Sha et al. 2017], to store dynamic graphs. However, all of these works, including those identified in Section 2.1, rely on hand-optimized kernels to compute on graphs that are stored in their data structure of choice. By contrast, our technique automatically generates such kernels.

Existing sparse linear and tensor algebra compilers cannot readily, if at all, generate code to efficiently compute on tensors stored in disparate dynamic sparse tensor formats. TACO [Chou et al. 2018, 2020; Kjolstad et al. 2019, 2017] generates efficient code for computing on static sparse tensors that are stored in a wide range of array-based formats like CSR and DIA [Saad 2003]. However, TACO cannot generate code to compute on dynamic sparse tensors that are stored in pointer-based data structures, since the sparse tensor format abstraction of Chou et al. [2018] cannot represent those data structures. In particular, the aforementioned abstraction requires users to provide snippets of imperative code (i.e., level functions) that fully define iteration over data structures for storing the coordinates of nonzeros. While such an approach suffices for supporting array-based data structures, the same approach does not generalize well to pointer-based data structures. For one thing, as the example in Figure 2c shows, efficient iterators for even relatively simple pointer-based data structures like BSTs can be very complex and difficult to correctly implement. For another, even if users can implement efficient iterators for pointer-based data structures, such iterators alone cannot support computing all sparse tensor algebra operations as efficiently as possible, as Figure 2b highlights for instance. Our technique addresses both issues by instead only requiring users to provide high-level specifications that describe how different pointer-based data structures organize nonzeros in memory. From these specifications, our technique can then automatically generate code that efficiently access pointer-based data structures in different ways (e.g., using sequential iterators or map functions) depending on the computation to be performed.

The Bernoulli compiler [Kotlyar 1999; Kotlyar et al. 1997; Stodghill 1997] similarly generates sparse linear algebra kernels using an abstraction for sparse vector and matrix formats called the black-box protocol. Kotlyar [1999] shows how array-based linked lists can be expressed using the black-box protocol, though they do not consider tree-based data structures such as BSTs. Furthermore, the black-box protocol requires users to manually implement low-level iterators for supported data structures. Again, such iterators would not only be difficult to implement for tree-based data structures but would also not be able to support computing all sparse tensor algebra operations as efficiently as possible.

COMET [Tian et al. 2021] and MLIR [Bik et al. 2022] are two additional examples of sparse tensor algebra compilers that support a wide range of static tensor formats by decomposing them into per-dimension formats. Like TACO without our extension though, COMET and MLIR do not support pointer-based data structures. MT1 [Bik 1996; Bik and Wijshoff 1993, 1994] and SIPR [Pugh and Shpeisman 1999], meanwhile, each only support a fixed set of array-based formats for storing sparse vectors and matrices and also do not support any pointer-based formats. More recently, Venkat et al. [2015] have shown how polyhedral techniques can be utilized to generate sparse linear algebra code by representing array-based sparse matrix formats as uninterpreted functions. Additionally, Arnold et al. [2010] have shown how computations on array-based sparse matrix formats can be expressed using a functional language they develop called LL. Again though, these techniques cannot generate the types of algorithms that are needed to compute with dynamic sparse tensors stored in recursive, pointer-based data structures.

There exists a separate line of works on synthesizing data structure operations from declarative specifications. Many techniques have been proposed for synthesizing imperative programs that modify pointer-based data structures like AVL trees and linked lists, given either user-specified invariants [Kurilova and Rayside 2013; Qiu and Solar-Lezama 2017] or graphical specifications of the desired programs' inputs and outputs [Singh and Solar-Lezama 2011]. Other techniques have also been proposed for synthesizing functional programs from declarative specifications, including programs that process and manipulate pointer-based data structures [Kneuss et al. 2013; Polikarpova et al. 2016]. None of these techniques consider block data structures like C-trees, and they do not generate parallel code. In addition, Rayside et al. [2012] show how Java iterators can

be synthesized for pointer-based data structures given specifications written in relational logic, though their technique does not generate map functions or any other code to actually compute on elements stored in those data structures. Finally, Hawkins et al. [2011] show how low-level data representations can be synthesized given declarative specifications of the required relational interface. However, their technique composes together data structures (e.g., linked lists) that are hand-implemented in the C++ Standard Template Library and Boost, whereas our technique automatically generates code to access and compute on those aforementioned data structures.

7 CONCLUSION

We have shown how a compiler can automatically generate efficient code to perform tensor algebra computations on dynamic sparse tensors that are stored in recursive, pointer-based data structures. In particular, by making the code generator agnostic to any particular format, our technique allows users to extend the compiler to support new formats without modifying the code generator itself.

Our technique thereby makes it possible to build general-purpose systems that can effectively work with dynamic sparse tensors in a wide variety of formats. Recall that, to effectively work with dynamic sparse tensors in any particular format, a system must be able to both efficiently modify and efficiently compute on tensors stored in that format. As mentioned in Section 1.1, supporting efficient modifications of dynamic sparse tensors in any particular format essentially only requires a routine for inserting new nonzeros and a routine for deleting existing nonzeros. So to support efficiently modifying dynamic sparse tensors in a wide variety of formats, a developer only needs to manually implement a bounded number of optimized routines that can be invoked by a general-purpose system. Meanwhile, to efficiently perform arbitrary computations on dynamic sparse tensors in a wide variety of formats, a general-purpose system can simply execute code that our technique generates, which are always optimized for the desired computations and operand formats. By enabling such a general-purpose system to be built, our technique can significantly reduce the amount of programmer effort that is needed to effectively work with dynamic sparse data in a broad range of application domains.

REFERENCES

Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. 2016. Improving Performance of Sparse Matrix Dense Matrix Multiplication on Large-Scale Parallel Systems. *Parallel Comput.* 59, C (nov 2016), 71–96. https://doi.org/10.1016/j.parco.2016.10.001

Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. 2010. Specifying and Verifying Sparse Matrix Codes. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/1863543.1863581

Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. 2020. Dynamic Graphs on the GPU. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 739–748. https://doi.org/10.1109/IPDPS47924.2020.00081

Ariful Azad and Aydın Buluç. 2017. A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 688–697. https://doi.org/10.1109/IPDPS.2017.76

M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. 2012. Efficient and scalable computations with sparse tensors. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–6. https://doi.org/10.1109/HPEC.2012.6408676

Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

Aart JC Bik. 1996. *Compiler Support for Sparse Matrix Computations.* Ph. D. Dissertation. Leiden University.

Aart J.C. Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* (jun 2022). https://doi.org/10.1145/3544559

Aart JC Bik and Harry AG Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing.* ACM, 416–424.

Aart JC Bik and Harry AG Wijshoff. 1994. On automatic data structure selection and code generation for sparse computations. In *Languages and Compilers for Parallel Computing.* Springer, 57–75.

Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC).* 1–7. https://doi.org/10.1109/HPEC.2018.8547541

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12).* Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/2168836.2168846

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages.

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 823–838. https://doi.org/10.1145/3385412.3385963

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011).

Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019).* Association for Computing Machinery, New York, NY, USA, 918–934. https://doi.org/10.1145/3314221.3314598

David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing.* 1–5. https://doi.org/10.1109/HPEC.2012.6408680

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11).* Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/1993498.1993504

Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of Sparse Array Programming Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (Oct. 2021), 29 pages. https://doi.org/10.1145/3485505

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* (Washington, District of Columbia) *(PPoPP '19).* Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

Intel. 2020. Intel oneAPI Math Kernel Library Developer Reference. https://www.intel.com/content/dam/develop/external/us/en/documents/onemkl-developerreference-c.pdf

James King, Thomas Gilray, Robert M. Kirby, and Matthew Might. 2016. Dynamic Sparse-Matrix Allocation on GPUs. In *High Performance Computing,* Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.). Springer International Publishing, Cham, 61–80.

Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. (2019), 180–192. http://dl.acm.org/citation.cfm?id=3314872.3314894

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13).* Association for Computing Machinery, New York, NY, USA, 407–426.

https://doi.org/10.1145/2509136.2509555

Vladimir Kotlyar. 1999. *Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs*. Ph. D. Dissertation. Cornell University.

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing*. Springer, 318–327.

Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 249–263. https://www.usenix.org/conference/fast19/presentation/kumar

Darya Kurilova and Derek Rayside. 2013. On the Simplicity of Synthesizing Linked Data Structure Operations. *SIGPLAN Not.* 49, 3 (oct 2013), 155–158. https://doi.org/10.1145/2637365.2517225

Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 294–303.

Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 19, 15 pages. https://doi.org/10.1109/SC.2018.00022

Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209

Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering*. 363–374. https://doi.org/10.1109/ICDE.2015.7113298

Tim Mattson, David Bader, Jon Berry, Aydın Buluç, Jack Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Michael Stonebraker, Steve Wallach, and Andrew Yoo. 2013. Standards for Graph Algorithm Primitives. In *IEEE High Performance Extreme Computing Conference*. IEEE, 1–2. https://doi.org/10.1109/HPEC.2013.6670338

Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *High Performance Embedded Architectures and Compilers*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–125.

L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. The PageRank citation ranking: Bringing order to the Web. In *Proceedings of the 7th International World Wide Web Conference*. Brisbane, Australia, 161–172. citeseer.nj.nec.com/page98pagerank.html

Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1372–1385. https://doi.org/10.1145/3448016.3457313

Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. arXiv:1608.01409 [cs.CV]

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

William Pugh and Tatiana Shpeisman. 1999. SIPR: A new framework for generating efficient code for sparse matrix computations. In *Languages and Compilers for Parallel Computing*. Springer, 213–229.

Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural Synthesis of Provably-Correct Data-Structure Manipulations. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 65 (oct 2017), 28 pages. https://doi.org/10.1145/3133889

Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. 2017. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/3037697.3037745

Derek Rayside, Vajihollah Montaghami, Francesca Leung, Albert Yuen, Kevin Xu, and Daniel Jackson. 2012. Synthesizing Iterators from Abstraction Functions. *SIGPLAN Not.* 48, 3 (sep 2012), 31–40. https://doi.org/10.1145/2480361.2371407

Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

Dipanjan Sengupta and Shuaiwen Leon Song. 2017. EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 97–119.

Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (sep 2017), 107–120. https://doi.org/10.14778/3151113.3151122

Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (feb 2013), 135–146. https://doi.org/10.1145/2517327.2442530

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 289–299. https://doi.org/10.1145/2025113.2025153

Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 5.

Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 61–70.

Paul Stodghill. 1997. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. Ph. D. Dissertation. Cornell University.

Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. 2018. PAM: Parallel Augmented Maps. *SIGPLAN Not.* 53, 1 (feb 2018), 290–304. https://doi.org/10.1145/3200691.3178509

Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High-Performance Sparse Tensor Algebra Compiler in Multi-Level IR. https://doi.org/10.48550/ARXIV.2102.05187

Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI 2015)*. 521–532.

Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091058

Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. CVR: Efficient Vectorization of SpMV on x86 Processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 149–162. https://doi.org/10.1145/3168818

Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. 293–302. https://doi.org/10.1109/BigData.2017.8257937