Compilation of Shape Operators on Sparse Arrays

ALEXANDER J. ROOT, Stanford University, USA
BOBBY YAN, Stanford University, USA
PEIMING LIU, Google, USA
AART J.C. BIK, Google, USA
FREDRIK KJOLSTAD, Stanford University, USA

We show how to build a compiler for a sparse array language that supports shape operators (e.g. reshaping or concatenating arrays) in addition to compute operators. Existing sparse array programming systems implement generic shape operators for only some sparse data structures, reduce shape operators on other data structures to those, and do not support fusion. Our system compiles sparse array expressions to code that efficiently iterates over reshaped views of irregular sparse data structures, without needing to materialize temporary storage for intermediates. Our evaluation shows that our approach generates sparse array code competitive with popular sparse array libraries: shape operators perform on average 1.49× (0.408×–4.31×) better than hand-written kernels in `scipy.sparse` and 28× (0.476×–274×) better than generic implementations in `pydata/sparse`. For operators that require data structure conversions in these libraries, our generated code performs on average 4.19× (1.99×–8.36×) better than `scipy.sparse` and 54.6× (10.8×–259×) better than `pydata/sparse`. Finally, our evaluation also shows that fusing shape and compute operators improves the performance of several expressions (1.16×–2.79×).

1 INTRODUCTION

Shape operators are the type casts of array programming. The shape of an array is a key component of its type, and determines which operations on it are valid: vectors of different lengths cannot be added and matrices of incompatible dimensions cannot be multiplied. However, programmers often need to manipulate the shape of arrays, and do so via shape operators. Many important computations require explicitly manipulating array shapes, such as stacking constraint matrices in physical simulation [Solomon 2015], reshaping tensors in neural networks [Vaswani et al. 2017], and slicing images in biomedical computing [Boyse and Seidl 1996]:

$$y = \begin{bmatrix} A \\ B \\ C \end{bmatrix} x \qquad y = A \begin{bmatrix} X_{0,0} \\ \vdots \\ X_{n-1,m-1} \end{bmatrix} \qquad Y = A_{:,:,i}$$

Stacked matrix-vector multiply | Linear layer on a flattened matrix | 2D slice of 3D MRI data

As a result, dense array programming systems [Harris et al. 2020; Iverson 1962; Paszke et al. 2019] have ample support for such operators, and they often reduce to constant-time metadata edits.

Sparse array programming systems lag behind their dense counterparts, with incomplete support for shape operators. While sparse array libraries [Abbasi 2018; Virtanen et al. 2020] support some shape operators, the implementations reduce to a small set of hand-written kernels. The conversions to intermediate data structures required by these reductions incur a significant performance cost. Sparse compilers [Ahrens et al. 2023; Kjolstad et al. 2017; Ye et al. 2023; Zhao et al. 2022] lack a complete set of shape operators.

Compiler support for shape operators is necessary for performant sparse array systems. It is not enough to simply hand-engineer a library of sparse shape operators, nor is it feasible to

Authors' addresses: Alexander J. Root, Computer Science, Stanford University, USA, ajroot@stanford.edu; Bobby Yan, Computer Science, Stanford University, USA, bobbyy@stanford.edu; Peiming Liu, Google, USA, peiming@google.com; Aart J.C. Bik, Google, USA, ajcbik@google.com; Fredrik Kjolstad, Computer Science, Stanford University, USA, kjolstad@stanford.edu.

Table 1. Comparison of sparse shape operation support across several programming systems. Yellow circles indicate partial support: `scipy.sparse` and `pydata/sparse` have limited sparse data format support. TACO and Looplets support fusing compute operators, but not shape operators.

		Data Representation			Shape Operators			
Programming Model	Compilation	Dense	Sparse	Any dims	Slicing	Concatenation	Reshape	Fusion
`numpy`	✗	✓	✗	✓	✓	✓	✓	✗
`scipy.sparse`	✗	✓	●	✗	✓	✓	✓	✗
`pydata/sparse`	✗	✓	●	✓	✓	✓	✓	✗
TACO	✓	✓	✓	✓	✓	✗	✗	●
Looplets	✓	✓	✓	✓	✗	✓	✗	●
BURRITO (This Work)	✓	✓	✓	✓	✓	✓	✓	✓

implement the full Cartesian product of operations and sparse array data structures. Prior work on sparse compute operators (e.g. matrix multiplication) shows that compiler support is necessary for performance and programmer productivity, due to the scaling issues introduced with the large number of sparse data structures [Ahrens et al. 2023; Chou et al. 2018]. Compiler support for sparse shape operators enables generating fused code that is specialized to the data structures of the operands and the result. The key challenge in compiling sparse shape operators lies in generating code that iterates over, and computes on, combinations of reshaped, concatenated, and sliced views of irregular data structures representing sparse arrays.

While recent work on compilers for sparse tensor algebra [Ahrens et al. 2023; Bik et al. 2022; Kjolstad et al. 2017] addresses the sparse compute problem, they offer limited support for shape operators: TACO supports slicing unfused array operands [Henry et al. 2021] and Looplets [Ahrens et al. 2023] supports concatenation without fusion. These systems offer no unified representation of shape operators, and *no* compiler that we are aware of supports sparse reshape operations. Libraries such as `scipy.sparse` [Virtanen et al. 2020] and `pydata/sparse` [Abbasi 2018] are not feature complete, and are limited by the time required for the maintainers to implement different combinations of shape operators and sparse array data structures. Libraries also naturally do not support operator fusion.

We introduce a compiler-based approach for representing shape operators, building on ideas from TACO to support iterating over reshaped arrays. We implement these ideas in a prototype compiler named BURRITO[1]. We extend an existing array programming language to support shape operators (Section 3), describe compilation to a new intermediate representation (IR) for iterating over sparse data structures (Section 4), and outline the compilation process from our IR to efficient CPU code (Section 5 and Section 6). Our abstractions enable generation of specialized code for sparse array kernels containing shape operators, can generate code for a variety of data structures, and enable fusion across shape and compute operators. Table 1 summarizes how BURRITO compares to other array programming systems. Our technical contributions are:

(1) A lowering approach from an array language with shape operators (Section 3) to a high-level loop language that expresses fusion across operators (Section 4).
(2) A state-driven algorithm for generating optimized while loops that coiterate through reshaped iteration spaces (Section 5).
(3) A compilation approach for generating iteration code that abstracts both data structures and reshaped iteration spaces, based on a simple iterator model (Section 6).

BURRITO generates code competitive with hand-written libraries on unfused shape operators, and generally out-performs these libraries when given sparse kernels that offer fusion opportunities.

[1]A BURRITO is what you get when you *reshape* a TACO.

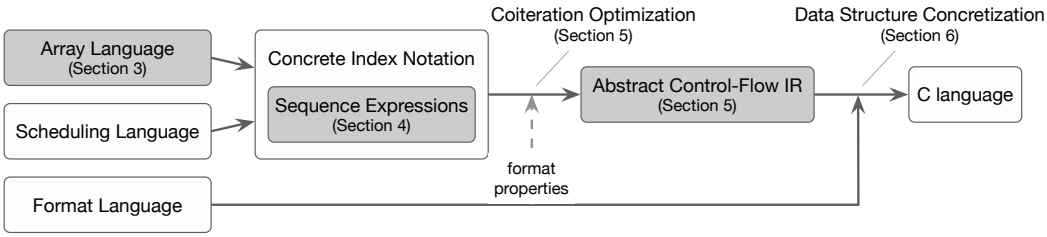


Fig. 2. The BURRITO compilation approach compiles an array language to C code in three steps through two loop-based intermediate representations over logical arrays. Section references mark our contributions.

Across five shape operators hand-written in `scipy.sparse` and `pydata/sparse`, BURRITO performs 1.11×–4.31× and 10.4×–274× (respectively) faster. A sixth shape operator performs 0.408× and 0.476× slower respectively due to vectorization, which BURRITO does not currently support. For shape operators whose library implementations convert between data structures, BURRITO-generated code out-performs `scipy.sparse` by 1.9×–8.36× and `pydata/sparse` by 6.29×–287×. To evaluate the benefits of fusion, we perform a self-comparison and show that code that fuses shape and compute operators out-performs unfused code by 1.16×–2.79× on some benchmarks.

2 OVERVIEW

The compilation approach we describe in this paper compiles an array language to C code. The array language is tensor index notation extended with shape operators, and operates on *logical* arrays that abstract away the details of the *physical* data structures. BURRITO separately accepts a format language that describes the sparse or dense physical data structures that implement the logical arrays, following prior work on separating algorithm from data structure [Chou et al. 2018; Kjolstad et al. 2017]. Finally, a separate scheduling language [Kjolstad et al. 2019] lets users or automated systems describe loop reordering and introduce temporaries.

Figure 2 shows an overview of our compilation approach, which we implement in the BURRITO compiler. Compilation consists of three lowering steps through two intermediate representations (IRs). The first step lowers the array language to nested forall loops over logical arrays. Scheduling directives (e.g. loop reordering) apply to this IR. Each loop iterates over an expression that describes the loop's iteration domain as an ordered set expression (sequence expression) over the coordinates of array dimensions. Prior work expresses compute operators as intersections and unions [Kjolstad et al. 2017] and further work adds complements and slicing [Henry et al. 2021]. In order to support the complete set of shape operators, our work adds three new set expression operators: set product, set projection, and disjoint union.

$$\underbrace{(\text{intersection} + \text{union})}_{\text{[Kjolstad et al. 2017]}} + \underbrace{(\text{complement} + \text{slicing})}_{\text{[Henry et al. 2021]}} + \underbrace{(\text{product} + \text{projection} + \text{disjoint union})}_{\text{[this work]}}$$

The forall loops are then lowered to an IR that expresses more complex control-flow. Each forall is turned into a sequence of while loops over sequence expressions that iterate over progressively simplified loop bodies as sparse arrays run out of non-zero coordinates. This lowering step requires knowledge of format properties to determine which arrays to iterate over and which arrays to randomly access into. The final step lowers the while loops to C code where logical arrays are replaced by the physical data structures described via the format language.

3 SHAPE OPERATORS

BURRITO compiles a language that extends tensor index notation for tensor algebra to support shape operators. Tensor index notation is a declarative language for describing tensor algebra.

```
kernel ::= array(I) = expr
 expr  ::= array(I) | expr + expr | expr * expr | sum(idx, expr) | broadcast(idx, expr) |
           collapse(expr, (idx, idx) → idx) | concat((expr, expr), (idx, idx) → idx) |
           split(expr, idx → (idx, idx)) | slice(expr, idx → idx, (int, int, int))
```

Fig. 3. Our front-end language for expressing array algebra, which supports tensor index notation (element-wise addition and multiplication, summation along an axis, and broadcasting), as well as our additional shape operators. Indices (`idx`) label array dimensions.

An example is matrix multiplication, $A_{ij} = \sum_k B_{ik} C_{kj}$, where each component of A_{ij} is the inner product of a row of B and a column of C. Shape operators are common in libraries like `numpy` and `PyTorch` and let users change the shape of a tensor in various ways. For example, a user can reshape (flatten) a matrix into a vector with the same number of elements or concatenate two matrices with the same number of rows into a matrix that contains columns from each of the two operands.

Figure 3 provides the syntax of the algorithm language supported by BURRITO. We first provide a high-level semantics for the shape operators by defining how each operator maps a coordinate in the logical input array(s) to a coordinate in the constructed logical array, and then provide a shape inference algorithm for detecting the validity of an array program via typing rules.

3.1 Operator Definitions

The shape of a logical n-dimensional logical array represents an n-dimensional hyper-rectangle, where each dimension has a size. Tensor index notation labels each dimension via *index variables*. Shape operators are used to explicitly manipulate the shapes of arrays by combining and constructing index variables, and thereby the corresponding dimensions. We define each shape operator with respect to the coordinate mapping it induces from one or more input arrays to the constructed array. Note that each of our operators are given as binary operators (i.e. binary concatenation), as BURRITO supports *fusion by default*.

Collapse. The **`collapse`** operator flattens two dimensions into one, constructing a new index with a size equal to the product of the size of the flattened dimensions. The shapes of the input and the output contain the same number of discrete points, but the output shape has one fewer dimensions than the input shape. The ordering of the flattened index variables (the second argument to **`collapse`**) controls the coordinate space mapping: the ordering of coordinates in the constructed dimension is equivalent to the the lexicographic ordering of the flattened dimensions. For example, a matrix `B` with rows labeled `i` and columns labeled `j` can be row-major collapsed via `a(k) = collapse(B(i, j), (i, j) → k)`, or column-major collapsed via `a(k) = collapse(B(i, j), (j, i) → k)`. The coordinate remapping follows the standard strided offset formula.

Concatenate. The **`concat`** operator is used to stack two arrays along a dimension. The shape of the output has the same number of discrete points as the sum of the points of the shapes of the inputs, and the same number of dimensions. The ordering of arrays control the coordinate space mapping: each coordinate in the first input array exists in the same location in the output array, and each coordinate in the second is offset in the dimension being concatenated. For example, a program that horizontally concatenates two arrays can be expressed as: `A(i, l) = concat((B(i, j), C(i, k)), (j, k) → l)`. `A` contains every coordinate `B` contains, as well as every coordinate that `C` contains, with the second coordinate of each element offset by the size of the dimension labeled by `j`.

Split. The **`split`** operator is the inverse of **`collapse`**, as it divides one source dimension into two constructed dimensions[2]. The source dimension has a size equal to the product of the sizes of

[2] **`split`** and **`collapse`** are building blocks that can be used to support the *reshape* operation.

the constructed dimensions. The input and output shapes contain the same number of discrete points, but the output shape has one more dimension than the input shape. The ordering of the pair of constructed indices controls the coordinate space mapping: the lexicographic ordering of the constructed dimensions is equivalent to the ordering of the source dimension. For example, consider splitting a 1D array into 2D: `A(i, j) = split(b(k), k → (i, j))`. For this row-wise splitting, the coordinate 1 in `b` corresponds to $(0, 1)$ in `A`. For a column-wise splitting: `A(i, j) = split(b(k), k → (j, i))`, the coordinate 1 in `b` corresponds to $(1, 0)$ in `A`. This coordinate remapping is defined by the inverse of the strided offset formula for **collapse**.

Slice. The **slice** operator extracts a uniform (possibly strided) subsequence from a dimension via values for the start, end, and stride. The shape of the input contains more discrete points[3] but has the same number of dimensions as the output shape. A coordinate in the sliced dimension is discarded if: 1) it is less than the start value; 2) greater than or equal to the end value; or 3) not a whole-number multiple of the stride value from the start index. If a coordinate passes these filters, i.e. a coordinate c in the input dimension is sliced with a start index of s and a stride of r meets the constraint: $c = s + x \cdot r$ for $x \in \mathbb{N}$ and $c < e$, then it maps to the coordinate x in the output dimension. For example, if a user wished to select the even-indexed values of a vector, they could write the kernel `a(i) = slice(b(j), j → i, (0, J, 2))`, or if they wanted the first half of an array, `a(i) = slice(b(j), j → i, (0, J/2, 1))`. Note that Henry et al. [2021] introduced slicing on sparse array operands only, while BURRITO supports slicing intermediate computations, i.e. slicing the result of a multiplication, or a reshaped array.

3.2 Shape Inference

The shape of an expression in array algebra is an essential component of the *type*[4], and an array compiler requires a shape inference algorithm for checking the validity of programs. For tensor index notation, shapes are easy to infer; element-wise operations produce an output with the same shape as the inputs and reductions remove a dimension from the shape of the operand. Broadcasting, though implicit in tensor index notation, is technically a shape operator, as it inserts a new dimension into the logical shape of an array. For the new shape operators defined in the prior section, new dimensions are constructed that a compiler must reason about in order to check program correctness.

As described previously, the shape of an array describes the number of dimensions and the size of each dimension. We denote shape as S, an unordered set of indices, each of which labels a dimension with a name and size. The set cardinality operator $|i|$ is used to represent the size of the dimension indexed by i.

The full inference algorithm is provided in Figure 4, but as an illustrative example, the inference rule for **collapse** is:

$$\frac{\mathtt{a} : S \quad \mathtt{i} \in S \quad \mathtt{j} \in S \quad \mathtt{k} \notin S \quad |\mathtt{i}| \cdot |\mathtt{j}| = |\mathtt{k}|}{\mathbf{collapse}(\mathtt{a},\ (\mathtt{i},\ \mathtt{j}) \rightarrow \mathtt{k}) : (S - \{\mathtt{i}, \mathtt{j}\}) \cup \{\mathtt{k}\}}$$

The first two constraints, $\mathtt{i} \in S$ and $\mathtt{j} \in S$, require that the source indices that are being collapsed, `i` and `j`, are contained within the shape, S, of the array operand, `a`. Next, $\mathtt{k} \notin S$ requires that the constructed index, `k`, is not already in S, which is necessary for the uniqueness of indices in the output shape. The last constraint, $|\mathtt{i}| \cdot |\mathtt{j}| = |\mathtt{k}|$, requires that the size of the dimension of the constructed index, $|\mathtt{k}|$, is the product of the size of dimensions represented by the source indices, $|\mathtt{i}|$ and $|\mathtt{j}|$. This means that the output array of the **collapse** has the same number of discrete elements

[3]Assuming the slice is non-trivial: a trivial slice removes no elements, i.e. slicing i with a start of 0, an end of the size of i, and a stride of 1.

[4]The type of an array consists of the shape and the type of the scalar elements.

$$\frac{}{\texttt{a}(I) : I}$$

$$\frac{\texttt{a} : S \quad \texttt{i} \in S \quad \texttt{j} \in S \quad \texttt{k} \notin S \quad |\texttt{i}| \cdot |\texttt{j}| = |\texttt{k}|}{\textbf{collapse}(\texttt{a},\ (\texttt{i},\ \texttt{j}) \rightarrow \texttt{k}) : (S - \{\texttt{i}, \texttt{j}\}) \cup \{\texttt{k}\}}$$

$$\frac{\texttt{a} : S \quad \texttt{i} \notin S}{\textbf{broadcast}(\texttt{i},\ \texttt{a}) : S \cup \{\texttt{i}\}}$$

$$\frac{\texttt{a} : S_a \quad \texttt{b} : S_b \quad S_a - \{\texttt{i}\} = S_b - \{\texttt{j}\} \quad \texttt{k} \notin S_a \quad |\texttt{i}| + |\texttt{j}| = |\texttt{k}|}{\textbf{concat}((\texttt{a},\ \texttt{b}),\ (\texttt{i},\ \texttt{j}) \rightarrow \texttt{k}) : (S_a - \{\texttt{i}\}) \cup \{\texttt{k}\}}$$

$$\frac{\texttt{a} : S_a \quad \texttt{b} : S_b \quad S_a = S_b}{\texttt{a}\ \texttt{<op>}\ \texttt{b} : S_b}$$

$$\frac{\texttt{a} : S \quad \texttt{i} \in S \quad \texttt{j} \notin S \quad \texttt{k} \notin S \quad |\texttt{i}| = |\texttt{j}| \cdot |\texttt{k}|}{\textbf{split}(\texttt{a},\ \texttt{i} \rightarrow (\texttt{j},\ \texttt{k})) : (S - \{\texttt{i}\}) \cup \{\texttt{j}, \texttt{k}\}}$$

$$\frac{\texttt{a} : S \quad \texttt{i} \in S}{\textbf{sum}(\texttt{i},\ \texttt{a}) : S - \{\texttt{i}\}}$$

$$\frac{\texttt{a} : S \quad \texttt{i} \in S \quad \texttt{j} \notin S \quad \frac{e-s+(r-1)}{r} = |\texttt{j}|}{\textbf{slice}(\texttt{a},\ \texttt{i} \rightarrow \texttt{j},\ (\texttt{s},\ \texttt{e},\ \texttt{r})) : (S - \{\texttt{i}\}) \cup \{\texttt{j}\}}$$

Fig. 4. Shape inference rules and validity constraints for shape and compute operators. An array's shape is the set of indices that index it (note that the indices used to *read* an array are ordered, but the *shape* is unordered). Broadcasting inserts a new index into the shape. Element-wise operations produce an expression with the same shape as the inputs. Summation removes the reduced index from the shape. Collapsing flattens two indices into a single constructed index, while splitting divides a single source index into two constructed indices. Concatenation accepts a pair of expressions and pair of indices, concatenating the two expressions along the two indices to produce a single constructed index. Concatenation requires that the non-concatenated indices of each operand must match. Slicing re-labels a source index into a smaller constructed index.

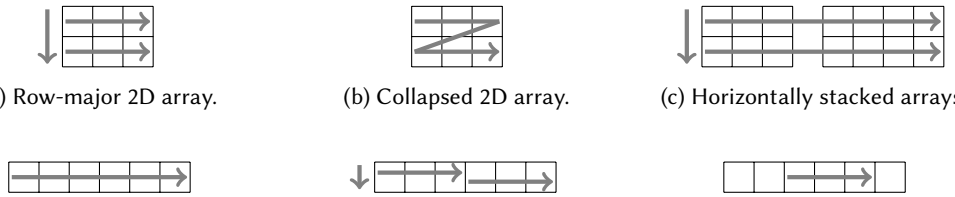
in space as the array being collapsed. The output array has a shape where dimensions other than `i` and `j` are unchanged, and `i` and `j` are replaced by `k`.

Consider the program `c(k) = a(k) * collapse(B(i, j), (i, j) → k)`, the element-wise multiplication of a vector with a flattened matrix. This program is well-typed: the shape of the argument to **collapse** is $S_{\texttt{B(i, j)}} = \{\texttt{i}, \texttt{j}\}$, and $\texttt{i} \in S_{\texttt{B(i, j)}}$ and $\texttt{j} \in S_{\texttt{B(i, j)}}$ are trivially true. Likewise, the constructed index is not in the source shape, $\texttt{k} \notin S_{\texttt{B(i, j)}}$. Lastly, the dimensionality constraint is satisfied if $|\texttt{i}| \cdot |\texttt{j}| = |\texttt{k}|$. Stepping up a level, both operands to the multiplication have a shape of $\{\texttt{k}\}$, so the multiplication is well-typed.

4 SEQUENCE EXPRESSIONS

The first compilation pass generates forall loops over logical arrays from the operators described in the previous section. The IR for representing forall loop extends TACO's concrete index notation (CIN) [Kjolstad et al. 2019] with a more advanced language for describing the iteration spaces of loops, termed *sequence expressions*. CIN is a language of loops over logical arrays that supports loop transformations such as loop reordering and the allocation of temporaries. These transformations are essential for avoiding inefficient *discordant traversals*, which iterate over a data structure in an order for which it was not designed [Bik et al. 1994; Gustavson 1972]. Collapsing a CSR matrix in column-row order or concatenating a CSR matrix with a CSC matrix are both examples of discordant traversals. It is generally more efficient to transpose an operand into a temporary matrix, and iterate over the temporary in concordant order in these cases, as discordant traversals require a dense loop with sparse look-ups. This section focuses on our novel contribution, sequence expressions, and we refer the reader to prior work for further discussion of CIN, loop transformations, and avoiding discordant traversals [Kjolstad et al. 2019].

Sequence expressions are closely related to the coordinate mappings described in Section 3.1, and can express both compute and shape operators. For example, addition corresponds to the union of sequences and **collapse** corresponds to the Cartesian product of sequences, where the coordinates in the resulted tuples are combined using a strided offset formula. Each shape operator corresponds to an operation on the sequences of non-zeros of its operands, and these operations fundamentally



(a) Row-major 2D array. (b) Collapsed 2D array. (c) Horizontally stacked arrays.

(d) 1D array. (e) Split 1D array. (f) Sliced 1D array.

Fig. 5. Various space filling curves.

change the iteration spaces of the loops. We provide semantics for sequence combinators and a construction algorithm for generating sequence expressions from the expression language described in Section 3.1.

4.1 Sequence Combinator Semantics

We provide the full grammar for CIN below, but this section focuses on our novel contribution: the grammar for sequence expressions, seq.

```
stmt  ::=  forall idx ∈ seq stmt | stmt; stmt | stmt where stmt | a(I) = expr | a(I) += expr
seq   ::=  idx_a | seq ∪ seq | seq ∩ seq | seq × seq | seq ⊔ seq | π_k( seq, (int, int) ) | seq[int:int:int]
```

Each sequence operator operates on a sequence of coordinates in a dimension of one or more arrays. Sequence operators corresponding to shape operators perform coordinate space transformations on their operand sequences. The semantics of these operators are important both for understanding how shape operators combine iteration spaces, and for compiling down to loops over irregular data structures. These operators are used to generate a sequence expression that represents the set of non-zero coordinates of the output array of a computation, for each dimension of the output array. Intuitively, each operator corresponds to an operation that changes the space filling curve of a hyperrectangle, and we provide examples in Figure 5. The semantics of these operators are as follows:

product: $A \times B$ is isomorphic to the Cartesian product, but applies a strided offset formula to the produced tuples to generate a whole number coordinate value. Intuitively, product corresponds to: for each element in A, step through each element in B. Figure 5b depicts the product space-filling curve of a collapsed matrix. Formally, product is expressed as

$$a \in A \text{ and } b \in B \iff (a \cdot |B| + b) \in A \times B$$

concatenation: $A \sqcup B$ is isomorphic to a disjunctive union, but produces a sequence with a range that is the sum of the input sequence ranges. Intuitively, this operator corresponds to first stepping through each element in A and then through each element in B. Figure 5c depicts the space filling curves of horizontally concatenated matrices. Formally

$$a \in A \iff a \in A \sqcup B \qquad b \in B \iff (b + |A|) \in A \sqcup B$$

projection: $\pi_k(A, (I, J))$ is isomorphic to the set projection operation, but first applies the inverse of the strided offset formula to elements of A to produce a tuple from a single coordinate, with the shape (I, J). The projection index, k, selects the kth element from the produced tuple. Because BURRITO's front-end allows users to express binary splitting, sequence expressions only need to support two-way projections, which can be chained if a dimension needs to be split more than once. Intuitively, a projection of a sequence produces two sequences that each iterate over sub-spaces of the original sequence. The first sequence $\pi_0(A, (I, J))$ is a sequence of size I, and the second, $\pi_1(A, (I, J))$ is a sequence of size J. Projection is the inverse to product, just as collapsing and

```
a(i) = b(i) * c(i)
```

$\downarrow$

$$\texttt{forall}\ \ i \in i_b \cap i_c \quad a(i) = b(i) * c(i)$$

```
A(i, j) = split(b(k), k → (i, j))
```

$\downarrow$

$$\texttt{forall}\ \ i \in \pi_0(k_b, [I, J]) \quad \texttt{forall}\ \ j \in \pi_1(k_b, [I, J]) \quad A(i, j) = b(k)$$

```
c(k) = collapse(A(i, j), (i, j) → k) + b(k)
```

$\downarrow$

$$\texttt{forall}\ \ k \in (i_A \times j_A) \cup k_b \quad c(k) = A(i, j) + b(k)$$

Fig. 6. CIN for various kernels.

splitting are inverses. Figure 5e depicts the space filling curves produced by splitting a vector into two dimensions. Formally

$$a \in A \iff \frac{a}{J} \in \pi_0(A, (I, J)) \text{ and } (a\%J) \in \pi_1(A, (I, J)).$$

slice: $A[s{:}e{:}r]$ is a simple filtering operation. Intuitively, slicing removes all elements from a sequence that are not a stride of r away from the start s, and any elements greater than e. Figure 5f depicts the space filling curve produced by slicing the 3rd through the 5th elements of a length 6 vector. Formally

$$a \in A \text{ and } s \leq a \leq e \text{ and } \exists x \in \mathbb{N}, a = s + r \cdot x \iff x \in A[s{:}e{:}r].$$

4.2 Sequence Combinator Construction

BURRITO's construction of sequence expressions from an array expression is defined by the function Υ`(e, i)`. Υ accepts an expression, `e`, from the grammar defined in Figure 3, and an index variable, `i`, and generates a sequence expression that represents the output coordinates in the `i`th dimension of `e`. The full definition of Υ is given in Figure 7, and we provide an example for intuition.

Consider the compilation of the rightmost kernel in Figure 6, which generates a single loop over the output index, `k`. In order to derive the sequence expression that corresponds to `k`'s iteration space, the recursive construction algorithm applies the following rules from Figure 7:

$$\Upsilon(\texttt{a + b, i}) = \Upsilon(\texttt{a}, i) \cup \Upsilon(\texttt{b, i}) \qquad \Upsilon(\texttt{collapse(a, (j, k)} \rightarrow \texttt{l), i}) = \begin{cases} \Upsilon(\texttt{a}, j) \times \Upsilon(\texttt{a}, k) & i = l \\ \Upsilon(\texttt{a}, i) & i \neq j \end{cases}$$

These rules construct the sequence $(i_A \times j_A) \cup k_b$, the loop bounds in the rightmost loop of Figure 6. This sequence expression means that the output sequence of non-zeros is equivalent to the flattened sequence of `a`'s non-zeros unioned with the sequence of `b`'s non-zeros. This exactly represents the set of non-zeros in the output array.

$$\Upsilon(\texttt{t(I), i}) = \begin{cases} i_t & i \in I \\ \emptyset & i \notin I \end{cases} \qquad \Upsilon(\texttt{collapse(a, (j, k)} \rightarrow \texttt{l), i}) = \begin{cases} \Upsilon(\texttt{a}, j) \times \Upsilon(\texttt{a}, k) & i = l \\ \Upsilon(\texttt{a}, i) & i \neq j \end{cases}$$

$$\Upsilon(\texttt{a + b, i}) = \Upsilon(\texttt{a}, i) \cup \Upsilon(\texttt{b, i}) \qquad \Upsilon(\texttt{concat((a, b), (j, k)} \rightarrow \texttt{l), i}) = \begin{cases} \Upsilon(\texttt{a}, j) \sqcup \Upsilon(\texttt{b}, k) & i = l \\ \Upsilon(\texttt{a}, i) \cup \Upsilon(\texttt{b}, i) & i \neq l \end{cases}$$

$$\Upsilon(\texttt{a * b, i}) = \Upsilon(\texttt{a}, i) \cap \Upsilon(\texttt{b, i}) \qquad \Upsilon(\texttt{split(a, j} \rightarrow \texttt{(k, l)), i}) = \begin{cases} \pi_0(\Upsilon(\texttt{a}, j), (|k|, |l|)) & i = k \\ \pi_1(\Upsilon(\texttt{a}, j), (|k|, |l|)) & i = l \\ \Upsilon(\texttt{a}, i) & \end{cases}$$

$$\Upsilon(\texttt{bc(j, a), i}) = \begin{cases} \mathbb{U}_i & i = j \\ \Upsilon(\texttt{a}, i) & i \neq j \end{cases} \qquad \Upsilon(\texttt{slice(a, j} \rightarrow \texttt{k, R), i}) = \begin{cases} \Upsilon(\texttt{a, j)[R]} & i = k \\ \Upsilon(\texttt{a, i}) & i \neq k \end{cases}$$

Fig. 7. Υ maps an array algebra expression and an index to a *sequence expression*, which represents the non-zero elements being iterated over. For **slice**, `R` corresponds to the slice parameters (start, end, and stride). Reductions (**sum**) are not handled here because lowering rewrites reductions into loops with reduction temporaries when constructing forall loops.

5 COMPILING TO COITERATING LOOPS

We introduce a second intermediate representation that contains more complex control-flow constructs than CIN's forall loops: CFIR (Control Flow Intermediate Representation). To generate code that efficiently iterates over a sequence expression, BURRITO must reason about when a sparse sequence runs out of elements (i.e. when an array runs out of values), or does not contain a particular coordinate. CFIR allows BURRITO to represent these optimizations while still abstracting away the concrete details of the underlying physical data structures. We first define CFIR before showing how to construct it via a generalized form of iteration lattices [Henry et al. 2021; Kjolstad et al. 2017]. Iteration lattice construction reasons about *format properties* to determine which sequences should be iterated over, and which can be randomly-accessed into, but lattices and CFIR still abstract away details of the physical data structures underlying the logical arrays.

5.1 Control Flow Intermediate Representation

We provide the full grammar for CFIR below, whose key components are loops and conditional execution (switch statements).

```
cfir  ::=  while idx ← seq (with (seq = seq)*)?) cfir | switch idx (case seq: cfir)+ |
           cfir; cfir | a(I) = expr | a(I) += expr
```

5.1.1 Loops. The **while** loops of CFIR iterate over a sequence expression until an operand runs out of elements. This construct is useful because a sequence expression and a loop body can be simplified if any sub-sequence is empty, and that simplification produces simpler and more efficient code. Therefore, if a sub-sequence becomes empty, control flow should transition to a loop over a simpler sequence expression with a simpler loop body. For example, consider the example program in Figure 8, which adds a flattened 2D array to a compressed 1D array. If the flattened matrix runs out of non-zero values, then the program only needs to iterate over the compressed vector. Likewise, if the compressed vector runs out of elements, the program only needs to iterate over the flattened matrix. The single CIN loop in Figure 8d is compiled to three CFIR loops in Figure 8f, where the latter two have simplified loop bodies.

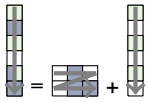
CFIR loops also support *locating* into sequences that support random-access (via the **with** operation) in order to access array values. In certain circumstances, a sequence does not need to be

```
c(k) = collapse(A(i, j),(i,j)→k)
       + b(k)
```

(a) Array algorithm description.

```
A: Dense , Compressed
b: Compressed
c: Dense
```

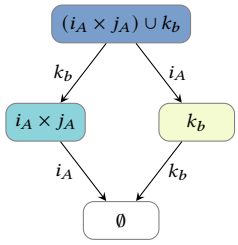
(b) Array format description.



(c) Logical iteration pattern.

```
forall k ∈ (i_A × j_A) ∪ k_b
  c(k) = A(i, j) + b(k)
```

(d) Concrete index notation.



(e) k's iteration lattice.

```
while k ← (i_A × j_A) ∪ k_b
  switch k
    case (i_A × j_A) ∩ k_b :
      c(k) = A(i, j) + b(k)
    case (i_A × j_A) :
      c(k) = A(i, j)
    case k_b :
      c(k) = b(k)
while k ← i_A × j_A
  c(k) = A(i, j)
while k ← k_b
  c(k) = b(k)
```

(f) Control-flow IR.

Fig. 8. Compilation of the fused collapse-and-addition in (a) with the formats in (b) produces the CIN in (d), visually represented by the space-filling curves in (c). Compilation of (d) generates the iteration lattice (e), which constructs the coiterating loops in (f). For the final compiled C output, refer to Figure 19.

```
C(i, j) = split(a(k), k→(i, j))
          * B(i, j)
```

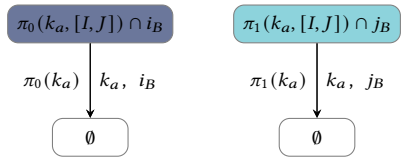
(a) Array algorithm description.

```
a: Compressed
B: Dense , Compressed
C: Dense , Dense
```

(b) Array format description.

```
forall i ∈ π0(k_a, [I,J]) ∩ i_B
  forall j ∈ π1(k_a, [I,J]) ∩ j_B
    C(i,j) = a(k) * B(i,j)
```

(c) Concrete index notation



(d) Iteration lattices for i and j.

```
1 while i ← π0(k_a, [I,J]) with i = i_B
2   while j ← π1(k_a, [I,J]) ∩ j_B
3     switch j
4       case π1(k_a, [I,J]) ∩ j_B :
5         C(i,j) = a(k) * B(i,j)
```

(e) Control-flow IR.

Fig. 9. Multiplication of a split compressed vector and a CSR matrix exposes an opportunity for the iterate/locate optimization: the i loop can iterate over the projection of the sparse vector's sequence, and use i to locate into a row of the CSR matrix, possibly skipping some rows. The j loop still coiterates the split sequence and the compressed columns of the CSR matrix, as both sequences are sparse. Refer to Figure 20b for the final generated C code from this example.

fully iterated. For example, if iterating over $a \cap b$, and $a \subseteq b$, it is sufficient to iterate over only a, and randomly access into b to get the value labeled by its coordinate. This is the iterate/locate optimization described by Kjolstad et al. [2017], because iterating over a sparse sequence and locating into a dense sequence is asymptotically optimal. BURRITO supports the same optimization, using *format properties* to detect which sub-sequences support fast ($O(1)$) location. In general, if a sequence is dense, or constructed from only dense sequences (i.e. the product of two dense sequences), it can and should be located into. For an example, in Figure 9, the sparse sequence $\pi_0(k_a, [I, J])$ can be used to locate into the dense rows of B. The final C code generated from Figure 9 is in Figure 20b, with line 4 performing the locate.

5.1.2 Conditionals. When iterating over a sequence, some sub-sequences may not contain a coordinate that other sub-sequences contain. Conditional execution is required to represent when a loop body should execute a simplified computation based on which sub-sequences contain a particular coordinate. In CFIR, this is handled by the **switch** construct, which redirects computation based on the value of a coordinate. For example, consider the loops in Figure 12d. When the **split** vector still has coordinates left, but does not contain a particular i coordinate that the CSR matrix does contain, the second loop only needs to iterate over the j values of the CSR matrix. This is reflected in lines 16–18 of Figure 12g.

5.2 Generalized Iteration Lattices

Iteration lattices [Henry et al. 2021; Kjolstad et al. 2017] are ordered state machines that enable reasoning about multi-way merging of coordinate sets. We extend iteration lattices to represent iteration over the sequence expressions defined in Section 4, provide a construction algorithm for iteration lattices, and show how to use iteration lattices to compile a loop in CIN to CFIR loops and conditionals.

An iteration lattice consists of ordered points. A point is labeled by a sequence expression, and represents iterating over that sequence. Each point has child points that represent simplified versions of the parent's sequence expression. Edges to children are labeled by a sub-sequence whose removal from the parent's sequence expression produces the simplified sequence expression in the

child point. Concretely, a point representing the sequence s has an edge labeled e to a child point representing the sequence t if removing e from s (replacing it with the empty set) and performing simplification produces the sequence t. This simplification is used to construct a lattice: we provide our algorithm for lattice construction in Figure 10, which follows a recursive top-down approach that produces progressively simpler sequence expressions to iterate over.

Consider the iteration lattice in Figure 8e, which represents the iteration pattern produced from adding a collapsed CSR matrix, A, to a compressed vector, b. If b runs out of values (the sequence k_b becomes empty), then iteration only needs to be performed over the collapsed matrix. This is represented by the child $i_A \times j_A$ of the top node in the lattice, which is transitioned to when k_b runs out of elements. Likewise, if the matrix runs out of elements (the product becomes empty), iteration only needs to proceed over the vector, k_b. This results in the leftmost and rightmost nodes in the lattice, respectively. Note that the transition to the node represented by k_b is labeled only i_A, not j_A. This is due to the semantics of product: if j_A runs out of elements, i_A needs to be stepped forward, which means that a product *only runs out* when the first operand runs out of coordinates.

```
func ConstructLattice(seq):
  point = LatticePoint(seq)
  // Transition sub-sequences
  // (Section 5.2.1 and Figure 11)
  edges = χ(seq)
  for sub in edges:
    // Simplification
    // (Section 5.2.2)
    r = remove(sub, seq)
    s = simplify(r)
    l = ConstructLattice(s)
    point.add_child(sub, l)
  return point
```

Fig. 10. Top-down lattice construction.

5.2.1 Edge Sequences. Edges in an iteration lattice correspond to sequences that can introduce a state transition. These are often sequences produced by array levels, but can also correspond to slices or projections. This is because a slice (or projection) can run out before the sequence it is slicing (or projecting) runs out, and should induce a state transition when that happens.

We give an algorithm for collecting edges from a sequence expression in Figure 11, the function χ. It is a recursive algorithm that generates a set of sub-sequences that correspond to state transitions. Child lattice points are generated by replacing an edge sequence with an empty set and simplifying the sequence expression, which we discuss further below.

Consider the expression in Figure 12, which reshapes a compressed vector into a matrix and adds it to a CSR matrix. The j loop over the rows of the matrices changes state when: 1) the compressed vector runs out of elements; 2) the logical slice of the compressed vector that corresponds to a single row in the logical matrix runs out; and 3) the CSR matrix row runs out. These three cases are edges in Figure 12f, as each triggers a state transition.

5.2.2 Sequence Simplification. Sequence simplification follows simple set rules: e.g. if one side of an intersection is empty, the entire intersection is empty. Likewise, if one side of a union is empty, then return the other side of the union. Our algorithm treats sparse-and-empty and dense-and-empty differently. For example, if the set expression is $a \cup b$ and a is a dense sequence, when a becomes empty (dense-and-empty), the entire union is empty, because the sequence must have been fully iterated if a ran out of elements. This optimization can be seen in Figure 12e, where the dense

$$
\begin{array}{rcl@{\qquad}rcl}
\chi(i_a) & = & \{i_a\} & \chi(x \cup y) & = & \chi(x) \cup \chi(y) \\
\chi(x \cap y) & = & \chi(x) \cup \chi(y) & \chi(x \times y) & = & \chi(x) \\
\chi(x \sqcup y) & = & \chi(x) & \chi(\pi_k(x, S)) & = & \{\pi_k(x, S)\} \cup \chi(x) \\
\chi(x[\mathtt{R}]) & = & \{x[\mathtt{R}]\} \cup \chi(x) & \chi(s + x) & = & \chi(x)
\end{array}
$$

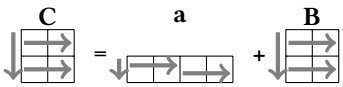
Fig. 11. χ generates a set of sub-sequences whose removal induces state transitions.

```
C(i, j) = split(a(k), k→(i, j))
      + B(i, j)
```

(a) Array algorithm description.

```
a: Compressed
B: Dense , Compressed
C: Dense , Dense
```

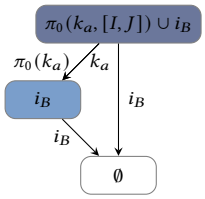
(b) Array format description.



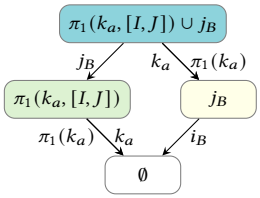
(c) Logical iteration pattern.

```
forall i ∈ π0(ka, [I,J]) ∪ iB
  forall j ∈ π1(ka, [I,J]) ∪ jB
    C(i,j) = a(k) + B(i,j)
```

(d) Concrete index notation.



(e) i's iteration lattice.



(f) One of the constructed iteration lattices for a j loop.

```
while i ← π0(ka, [I,J]) ∪ iB
 switch i
  case π0(ka, [I,J]) ∪ iB :
   while j ← π1(ka, [I,J]) ∪ jB
    switch j
     case π1(ka, [I,J]) ∪ jB :
      C(i,j) = a(k) + B(i,j)
     case π1(ka, [I,J]) :
      C(i,j) = a(k)
     case jB :
      C(i,j) = B(i,j)
   while j ← π1(ka, [I,J])
     C(i,j) = a(k)
   while j ← jB
     C(i,j) = B(i,j)
  case iB :
   while j ← jB
     C(i,j) = B(i,j)
while i ← iB
  while j ← jB
    C(i,j) = B(i,j)
```

(g) Control-flow IR.

Fig. 12. Addition of a split compressed vector and a CSR matrix has the same logical iteration pattern as the multiplication in Figure 9, but requires very different loops. The iteration lattice in (e) highlights the asymmetrical sequence simplification from treating sparse-and-empty and dense-and-empty as two different simplification results: when the dense i_B iterator runs out of coordinates, the sparse $\pi_0(k_a, [I, J])$ iterator must have run out of elements, so no lattice point is needed to represent iteration over solely $\pi_0(k_a, [I, J])$.

column of a CSR matrix is coiterated with a sparse sequence — when the dense sequence runs out, the entire union must run out.

5.2.3 Sub-points. Given a lattice point p that iterates over a sequence s, the sub-points of p are descendant points that represent sequences that are strict subsets of s. For example, in Figure 14e, the sequence $i_A \cup i_B$ has two sub-points, labeled i_A and i_B. Sub-points of p correspond to a sequence that represents a strictly *simpler* state than the state represented by p. Not all descendant points are sub-points due to the complexity of the concatenation operator. Consider the iteration lattice in Figure 14f; the concatenation operator produces a state transition when k_A runs out, indicating a transition to start iterating over the slice of l_B instead. However, this point is not considered a sub-point, because it represents a completely disjoint set (it requires k_A runs out, not just that k_A is missing an element). To compute the sub-points of a lattice point p, we gather all descendants that represent a strict subset of p's sequence.

5.3 Compiling Lattices to Loops

Iteration lattices are used to generate loops over progressively simpler loop bodies. A lattice always maintains a partial ordering, because each point has a progressively simpler sequence expression. The lattice can be compiled to loops by topologically sorting the points, laying them out in order, and generating a CFIR loop for each point that runs until an edge sequence runs out. A CFIR loop exits when an edge sequence runs out, as this indicates that the iteration should be passed to a loop over a simpler sequence expression. The body of a CFIR loop is generated as a conditional over the sub-points in the lattice, where the bodies of the conditional statements contain recursively simplified code. We provide the full compilation from CIN forall loops to CFIR loops in Figure 13.

Consider the CIN in Figure 12d, where the i loop iterates over a projection of a sparse iterator unioned with a dense iterator. This loop generates the iteration lattice in Figure 12e, which contains two states, the initial state (top), and a state that iterates over only the dense iterator (middle left), for when the sparse iterator runs out of elements. This iteration lattice compiles to the two outer loops in Figure 12g, one of which iterates over the union, and the other handles the case where the sparse iterator runs out of elements. Within the first outer loop, there is control flow to handle the case that the sparse iterator is at the same coordinate as the dense iterator, and the case where the dense iterator contains an element that the projection of the sparse iterator does not have. These cases within this first loop correspond to sub-states that the computation could be in, avoiding unnecessary work when the projection of the sparse iterator does not contain a coordinate.

```
func CompileForall(idx, seq, body):
  // Lattice construction in Figure 10
  lattice = ConstructLattice(seq)
  points = TopoSort(lattice)
  build = λ p: BuildLoop(p, idx, body)
  loops = map(build, points)
  return fold(Pair, loops)

func BuildLoop(point, idx, body):
  // Simplify loop body based on active
  // sequences.
  build = λ sp: Compile(simplify(sp, body))
  // Case for each sub-point (Section 5.2.3)
  bodies = map(build, point.subs)
  cond = Switch(idx, point.subs, bodies)
  return While(idx, point.seq, cond)
```

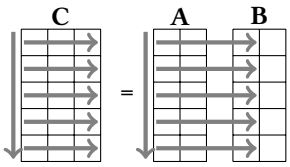
Fig. 13. Compilation of CIN forall loops to a chain of CFIR while loops over conditional statements.

```
C(i, j) = concat((A(i, k), slice(B(i, l), l → p, (0, L/2, 1))), (k, p) → j)
```

(a) Array algorithm description.

```
A: Compressed , Compressed
B: Compressed , Compressed
C: Dense , Dense
```

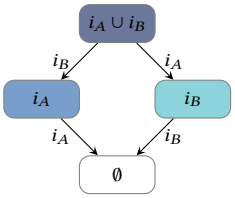
(b) Array format description.



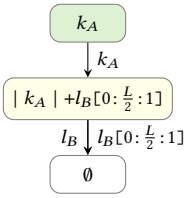
(c) Logical iteration pattern.

$$\begin{array}{l} \textbf{forall}\ \ i \in i_A \cup i_B \\ \quad \textbf{forall}\ \ j \in k_A \sqcup l_B[0:\frac{L}{2}:1] \\ \qquad C(i,j) = \textbf{concat}_{k,l}(A(i,k), B(i,l)) \end{array}$$

(d) Concrete index notation.



(e) i's iteration lattice.



(f) j's iteration lattice.

```
1  while i ← i_A ∪ i_B:
2   switch i
3    case i_A ∪ i_B:
4     while j ← k_A
5      C(i, j) = A(i, k)
6     while j ← |k_A| + l_B[0 : L/2 : 1]
7      C(i, j) = B(i, l)
8    case i_A:
9     while j ← k_A
10     C(i, j) = A(i, k)
11   case i_B:
12    while j ← |k_A| + l_B[0 : L/2 : 1]
13     C(i, j) = B(i, l)
14 while i ← i_A:
15  while j ← k_A
16   C(i, j) = A(i, k)
17 while i ← i_B:
18  while j ← |k_A| + l_B[0 : L/2 : 1]
19   C(i, j) = B(i, l)
```

(g) Control-flow IR.

Fig. 14. Concatenation produces a union over the shared dimensions, and iteration lattice construction for i generates loops specialized to when a coordinate in the sequence is both sequences or only one, producing simpler loop bodies in the latter case. Iteration lattice construction for j essentially fissions the loop over concatenated sequences into a pair of loops (lines 4-5 and 6-7 of (g)) that each coiterate a single sequence.

6 CODE GENERATION

In this section, we show how a simple set of primitives compose to produce iteration over any sequence expression. This final code generation pass uses a simple iterator model that builds on the indexed stream model of Kovach et al. [2023], a representation used for compiling fused sparse

```
Dense(String name, String idx,
      String size):
 Init():
  emit "int {idx}_{name} = 0;"

 Valid():
  emit "{idx}_{name} < {size}"

 Eval():
  emit "{idx}_{name}"

 Equals(i):
  emit "{idx}_{name} == {i}"

 Next():
  emit "{idx}_{name}++;"

 Locate(i):
  emit "{idx}_{name} = {i};"
```

Fig. 15. Iterator model for a dense dimension.

```
Compressed(String name, String idx,
           String lb, String ub):
 Init():
  emit "int {idx}p_{name} = {lb};"

 Valid():
  emit "{idx}p_{name} < {ub}"

 Eval():
  emit "{name}_crd[{idx}p_{name}]"

 Equals(i):
  emit "Eval() == {i}"

 Next():
  emit "{idx}p_{name}++;"

 Locate(i):
  emit "{idx}p_{name} = binary_search({i},
  {name}_crd, {lb}, {ub});"
```

Fig. 16. Iterator model for a compressed dimension.

tensor algebra that shows how to iterate over unions and intersections efficiently. We first describe the iterator model for sequence combinators, then for formats, and lastly provide the complete code generation algorithm from CFIR to C code.

6.1 Iterator Model

BURRITO relies on a small set of composable primitives to implement iteration over combinations of sequences: *initialize*, *valid*, *evaluate*, *equals*, *next*, and *locate*. We describe each below, giving examples of the iterator model for sequence products in Figure 17 and slicing in Figure 18. Note that Figure 18 shows a generalization of prior work [Henry et al. 2021], which supported slicing only array dimensions, while our algorithm can generate code that iterates over the slice of *any* sequence.

Initialize Initialization handles declaring any necessary iteration variables and locating the first non-zero element of a sequence.

Valid Valid is a check that the sequence has not run out of coordinates, meaning no subsequence has run out. This is a check that no iterators have gone out of bounds, and slices and projections are still within a valid range.

Evaluate Evaluation computes the current coordinate value of the sequence. For unions and intersections, this takes the minimum of the two sequences, but for sequence combinators, a sequence value (or values) must be re-mapped to the new space. For example, a product applies the strided offset formula to map two sequence coordinates to a single sequence coordinate. Likewise, a slice must re-map a sequence value to the sliced coordinate space.

Equals An equality test checks whether a sequence is currently at a coordinate. For unions and intersections, equality simply means that both sequence operands are equal to the given value. For combinators, we re-map the provided value into the spaces spanned by the operand sequences. This is used to compile CFIR's conditional statements (see Figure 20a).

Next The next method steps forward any sequences that are currently behind. For array levels, unions, and intersections, this performs the same conditional updates as TACO (i.e. line 25 of Figure 19). For sequence combinators, we define modular code generators for stepping forward a sequence.

Locate Location moves physical iterators to point to a specific logical coordinate. Section 5.1.1 described cases where random-access into a sequence is asymptotically preferable, and the locate interface is used to support this optimization. Many sequence operators are invertible

```
Product(Seq a, Seq b):
 Init():
  emit Init(a); Init(b)
  emit while(Valid(a) && !Valid(b))
  emit { Next(a); Init(b); }

 Valid():
  emit Valid(a) && Valid(b)

 Eval():
  emit (Eval(a) * |b|) + Eval(b)

 Equals(i):
  emit Equals(i/|b|,a) && Equals(i%|b|,b)

 Next():
  emit Next(b)
  emit while(Valid(a) && !Valid(b))
  emit { Next(a); Init(b); }

 Locate(i):
  emit Locate(i/|b|,a); Locate(i%|b|,b)
```

Fig. 17. Iterator interface for sequence products.

```
Slice(Seq a, Expr s, Expr e, Expr r):
 Init():
  emit Init(a)
  emit Locate(a, s)
  emit while(Valid() && ((Eval(a)-s)%r))
  emit { Next(a); }

 Valid():
  emit Valid(a) && Eval(a) < e

 Eval():
  emit (Eval(a) - s) / r

 Equals(i):
  emit Equals((i * r) + s, a)

 Next():
  emit do { Next(a) }
  emit while(Valid() && ((Eval(a)-s)%r))

 Locate(i):
  emit Locate((i * r) + s, a)
```

Fig. 18. Iterator interface for sequence slices.

```
int i_A = 0;                                                    // init i_A
int jp_A = A_pos[i_A];                                          // init j_A
while ((i_A < I) && !(jp_A < A_pos[i_A+1])) {                   // valid i_A && !valid j_A
  i_A++;                                                        // next i_A
  jp_A = A_pos[i_A];                                            // init j_A
}
int kp_b = b_pos[0];                                            // init k_b
while ((i_A < N) && (jp_A < A_pos[i_A+1]) && (kp_b < b_pos[1])) { // valid i_A × j_A ∪ k_b
  int k = min((i_A * M) + A_crd[jp_A], b_crd[kp_b]);            // eval (i_A × j_A) ∪ k_b
  if (((i_A == k / M) && (A_crd[jp_A] == k % M)) &&             // equals k, (i_A × j_A) ∪ k_b
      (k == b_crd[kp_b])) {
    c[k] = A.values[jp_A] + b.values[kp_b];
  } else if ((i_A == k / M) && (A_crd[jp_A] == k % M)) {        // equals k, i_A × j_A
    c[k] = A.values[jp_A];
  } else if (k == b_crd[kp_b]) {                                // equals k, k_b
    c[k] = b.values[kp_b];
  }
  if (k == ((i_A * M) + A_crd[jp_A])) {                         // next k, (i_A × j_A)
    jp_A++;                                                     // next j_A
    while ((i_A < N) && !(jp_A < A_pos[i_A+1])) {
      i_A++;                                                    // next i_A
      jp_A = A_pos[i_A];                                        // init j_A
    }
  }
  kp_b += (k == b_crd[kp_b]);                                   // next k, k_b
}
```

Fig. 19. Compilation of the first loop of Figure 8f, which iterates over a collapsed CSR matrix, A, and adds it to a compressed vector, b.

and this property can be used to locate through them, and base data structure formats often support fast random access (i.e. dense or hashed formats).

6.1.1 Formats. The iterator model for array formats is equivalent to the base case of the recursive algorithm for generating iteration code, as tensor dimensions are the main primitive in sequence expressions. As examples, we provide the implementations of dense and compressed formats in Figures 15 and 16, respectively. Extending BURRITO to support an additional sparse format only requires implementing the iterator interface for that format type. Our prototype compiler supports

TACO's dense, compressed, and singleton level formats [Chou et al. 2018], which compose to express many sparse data structures, such as CSR, CSC, COO, and variants.

6.1.2 Combinators. The iterator model fully composes to enable code generation for all sequence combinators. As examples, we provide the implementation of the iterator model for sequence products and slices in Figures 17 and 18, respectively. These implementations show the composability of this interface. We also illustrate a labeled example of the full C code generated from Figure 8 in Figure 19, with line labels corresponding to the interface that generated each line.

6.2 Compiling CFIR

We give the algorithm for compiling CFIR based on the iterator model in Figure 20a. Intuitively, each loop iterates until some sub-sequence runs out (the state is permanently changed), and after execution of a loop's body, steps the sequence forward. Conditional statements are used to evaluate which sub-state the iterator may currently be in, and generate if-else chains. The iterator model allows for a very simple code generation algorithm to compile the abstract while loops and switch statements of CFIR to the complex irregular loops over physical data structures required to support compute and shape operators.

7 EVALUATION

Our evaluation proves the following claims:

(1) Generated shape operators can match the performance of hand-written shape operators.
(2) Portability across data structures offers performance benefits over fixed-format kernels.
(3) Fusion of shape and compute operators can improve performance.

```
1  func CompileCFIR(stmt):
2   match stmt with
3   | While (idx, seq, locs, body) ->
4    emit Init(seq)
5    emit while (Valid(seq)) {
6    emit int idx = Eval(seq);
7    for a, b ∈ locs
8     emit Locate(Eval(a), b)
9    emit CompileCFIR(body)
10   emit Next(seq) }
11  | Switch (idx, seqs, bodies) ->
12   emit if (Equals(idx, seqs[0])) {
13   emit CompileCFIR(bodies[0]) }
14   for s, b ∈ zip(seqs, bodies)[1:]
15    emit else if (Equals(idx, s)) {
16    emit CompileCFIR(b) }
17  | Pair (cfir0, cfir1) ->
18   emit CompileCFIR(cfir0)
19   emit CompileCFIR(cfir1)
20  | Assign (array, idxs, expr) ->
21   emit CompileWrite(array, idxs)
22   emit CompileExpr(expr)
23  | Reduce (array, idxs, expr) ->
24   emit CompileReduction(array, idxs)
25   emit CompileExpr(expr)
```

(a) Recursive codegen via the iterator model.

```
4  int kp_a = a_pos[0];

5  while (kp_a < a_pos[1]) {
6    int i = a_crd[kp_a] / J;
8    int i_B = i;
4    int jp_B = B_pos[i];

     while ((kp_a < a_pos[1]) &&
5           (i == a_crd[kp_a] / J) &&
            (jp_B < B_pos[i_B+1])) {
       int j0 = a_crd[kp_a] % J;
6      int j1 = B_crd[jp_B];
       int j = min(j0, j1);
12     if ((j == j0) && (j == j1)) {
21-22    C[i*J + j] = a[kp_a] * B[jp_B];
13     }
       kp_a += (j == j0);
10     jp_B += (j == j1);
     }

     while ((kp_a < a_pos[1]) &&
            (i == a_crd[kp_a] / J)) {
10     kp_a++;
     }
   }
```

(b) Generated code from the CFIR in Figure 9e.

Fig. 20. (a) C code generation from CFIR. (b) shows the generated code for `C(i, j) = split(a(k), k → (i, j)) * B(i, j)`, where a is a compressed vector and B is a $I \times J$ CSR matrix. The labels to the left in (b) correspond to the line numbers in (a) that generated that particular line of code.

We first describe the existing state-of-the-art libraries that we compare to in Section 7.1, provide our benchmarking methodology in Section 7.2, and then provide evidence for the above claims in the following sections.

7.1 Baselines

We compare BURRITO-generated code to `scipy.sparse` and `pydata/sparse`, the two state-of-the-art libraries that have the most complete set of shape and compute operators. For shape operators, both of these libraries follow a simple reduction approach. Each shape operator is implemented for one or a few sparse data structures, and calling a shape operator on an unsupported sparse data structure incurs a conversion cost to convert to a supported format. For example, to *reshape* a CSR matrix with `scipy`, the library first converts to the CSR matrix to COO (the COOrdinate list data structure) and then calls `COO.reshape`. Likewise, when a user requests a non-standard output format (i.e. *reshape* a COO matrix into a CSR matrix), the library will perform the operation with a supported implementation and then convert the output to the requested output format. Note that for sparse tensor algebra alone, BURRITO's compilation technique is equivalent to TACO's, and achieves the same performance.

7.2 Methodology and Benchmark Notation

We evaluate on an Apple M1 Pro (3.2 GHz, 8 cores) with 16 GB of RAM. All benchmarks are single-threaded. We use the 12 largest SuiteSparse matrices that have indices fitting in 32 (unsigned) bits. These matrices span multiple domains, including computer vision, structural engineering, economics, graph analytics, and computational fluid dynamics. Each benchmark record is the minimum time of 5 iterations. For benchmarks that require multiple matrices (i.e. concatenation), we first split the SuiteSparse matrix in half, and use the halves as operands to concatenation. For fusion benchmarks, we use the same matrix shifted by a small number (10), due to the need for shape compatibility, as in prior evaluations of sparse tensor algebra [Kjolstad et al. 2017]. Note that BURRITO's compile times are interactive, so compilation does not introduce noticeable overhead.

We label benchmarks with short descriptions of their compute kernels. `vstack` means *vertical* stacking (concatenation along the first axis), `hstack` means *horizontal* stacking (concatenation along the second axis), and `reshape` is a `collapse` followed by a `split`. Benchmarks label operands with their array types (i.e. CSR or COO), and single letter labels ("C" or "D") correspond to randomly-generated vectors ("Compressed" and "Dense", respectively).

7.3 Comparison to Hand-written Kernels

We compare BURRITO's generated shape operators to shape operators that `scipy.sparse` and `pydata/sparse` whose implementations *do not* perform data structure conversions. This comparison shows that a compilation-based approach can match the performance of hand-written code.

The comparison to `scipy.sparse` is shown in Figure 21. On stacking COO matrices and slicing CSR matrices, performance is matched. On reshaping COO and horizontally stacking CSR, BURRITO out-performs `scipy`. Both of these performance benefits come from the fact that `scipy`'s implementations allocate a small number of temporary `numpy` arrays to perform conversions, while BURRITO allocates only the output array. On vertical stacking CSR matrices, `scipy` out-performs BURRITO: that particular operation is little more than a few `memcpy`s, and `scipy` has a highly optimized (vectorized) implementation of it, while the underlying C compiler used to compile BURRITO's generated code did not generate the same optimized `memcpy`s. However, in general, these benchmarks show that generated code can match or exceed the performance of hand-written reshape kernels.

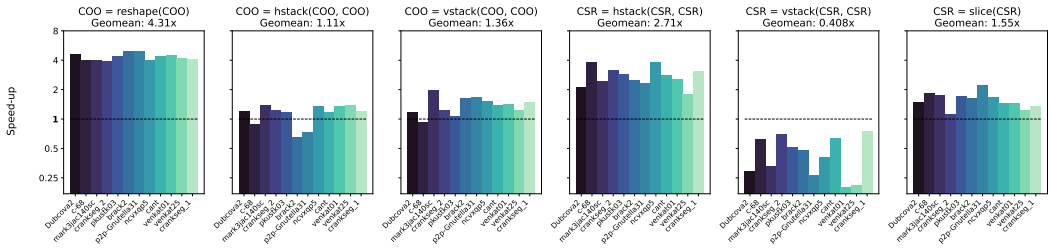


Fig. 21. Runtime speed-up over `scipy.sparse` of shape operators applied to SuiteSparse matrices, for kernels that do not perform data structure conversions.

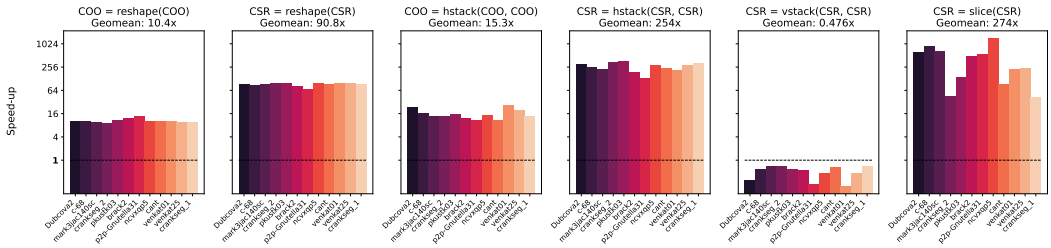
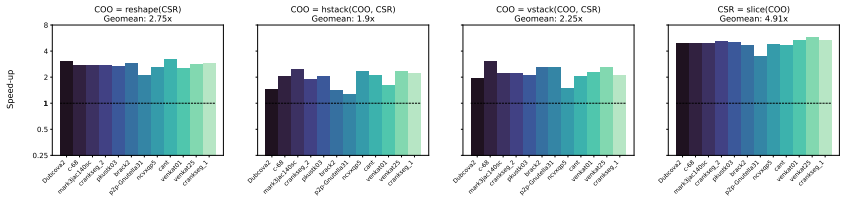
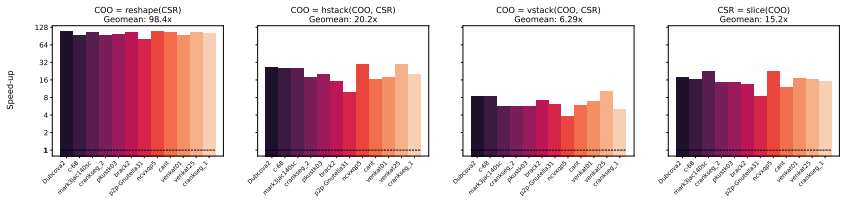


Fig. 22. Runtime speed-up over `pydata/sparse` of shape operators applied to SuiteSparse matrices, for kernels that do not perform data structure conversions.



(a) Runtime speed-up over `scipy.sparse`.



(b) Runtime speed-up over `pydata/sparse`.

Fig. 23. Evaluation of shape operators where existing SOTA techniques perform data structure conversions on at least one input array in order to perform the shaping operation.

We also compare to `pydata/sparse` is shown in Figure 22. BURRITO's generated code generally severely out-performs `pydata` because `pydata`'s implementions are dimensionality-agnostic (i.e. not specialized to a 2D COO, but works for an nD COO matrix), which is beneficial for reducing overall code size but not beneficial for code performance. This highlights the cost of generality in these systems. Like `scipy`, `pydata` out-performs BURRITO on vertically stacking CSR matrices, due to the same `memcpy` optimization.

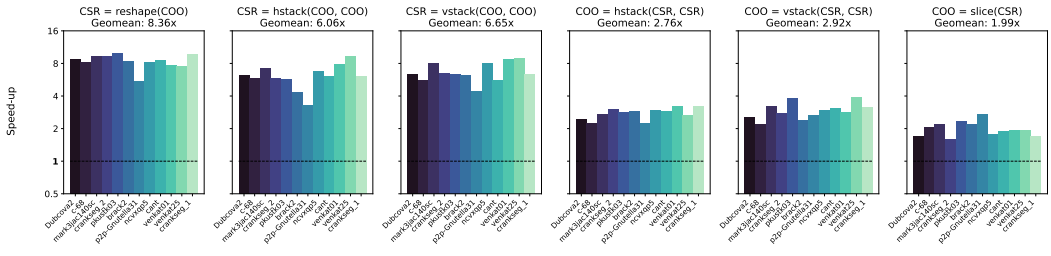


Fig. 24. Runtime speed-up over `scipy.sparse` of shape operators applied to SuiteSparse matrices, for kernels that `scipy.sparse` requires data structure conversions on the output of the shape operator.

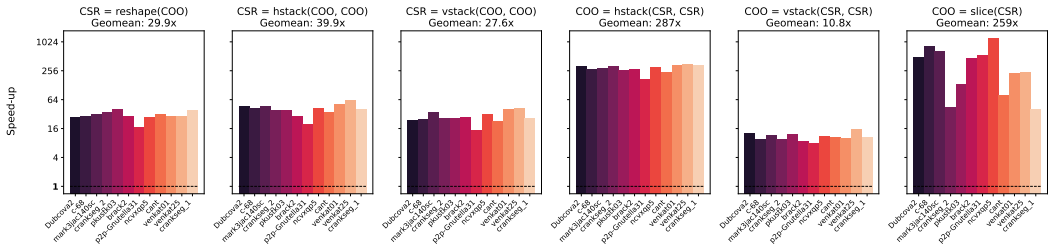


Fig. 25. Runtime speed-up over `pydata/sparse` of shape operators applied to SuiteSparse matrices, for kernels that `pydata/sparse` requires data structure conversions on the output of the shape operator.

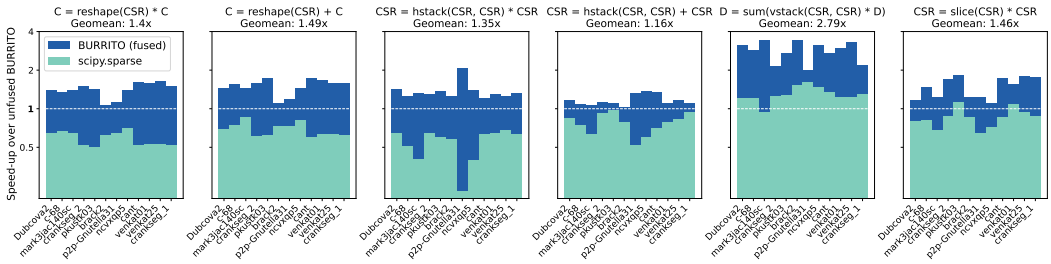


Fig. 26. Runtime speed-up over BURRITO-generated *unfused* code. The labeled geomeans show the speed-up of BURRITO-generated *fused* code.

7.4 Comparison to Reduced Kernels

To show the importance of allowing code to specialize to particular input and output data structures, we evaluate shape operators with data structures that the state-of-the-art libraries perform data structure conversions to compute. In Figure 23, we evaluate BURRITO generated code against implementations that first perform a data structure conversion on an input array before applying the hand-written shape operator. Figure 24 and Figure 25 show the performance difference of applying a shape operator with an output format that is different from the standard library implementation, requiring a data structure conversion following the shape operator. In each of these benchmarks, we see that generating code specialized to particular input and output data structures (avoiding data structure conversions) achieves performance benefits. BURRITO generated code also uses less memory, as it only allocates the output array, but not any temporary arrays.

7.5 Shape and Compute Operator Fusion

To evaluate the performance benefits of fusing shape with compute operators, we perform a series of self-comparisons on kernels with both shape and compute operators. For relevance to the

state-of-the-art, we also compare to `scipy.sparse`, the faster of the two libraries that we evaluate against in the prior sections. Figure 26 provides a performance evaluation of the speed-up gained by fusing shape and compute kernels. The benefit of fusing such operators is largely a result from reduced memory allocation, as there is no need to allocate expensive intermediate temporaries. There are some cases where `scipy.sparse` out-performs the *unfused* BURRITO-generated code (e.g. with the fused SpMV, where `scipy.sparse`'s fast `vstack`, discussed in Section 7.3, out-performs BURRITO's `vstack`), but *fused* BURRITO-generated code performs best across these benchmarks.

8 RELATED WORKS

Sparse Shape Operator Compilation. As discussed in the introduction, most prior work in sparse array compilation focuses on compiling compute operators [Bik et al. 2022; Kjolstad et al. 2017; Zhao et al. 2022], with the exceptions of two compilers: Henry et al. [2021] extended TACO to support iterating over slices of array operands, and allows computing over a slice, but does not support slicing intermediate computation, which limits fusion options; Looplets [Ahrens et al. 2023] can be used to express the concatenation of arrays, but does not express concatenation as an operator in the front-end language. BURRITO explicitly expresses shape operators in the front-end language, and has no restrictions on mixing compute and shape operators.

Abstracting Sparse Iteration. There are decades worth of work in abstracting sparse iteration. The database community relied on the iterator model [Graefe 1994] to implement many relational operators, and more recently, Kovach et al. [2023] introduced the stream model for iterating over sparse arrays, which supports a similar iterator interface based on a model equivalent to *init-valid-next*. Our iterator model builds on Kovach et al. [2023], and we introduce additional primitives to support shape operators. Chou et al. [2018], Looplets [Ahrens et al. 2023], and SparseTIR [Ye et al. 2023] provide various abstractions over sparse data formats, but do not use these abstractions to compile shape operators.

Avoiding Discordant Traversals. Sparse tensor algebra has a long history of shaping computation to avoid discordant traversals [Bik 1996; Bik et al. 1994; Duff et al. 1990; George and Liu 1981; Gustavson 1972; Pissanetsky 1984; Tewarson 1973; Zlatev 1991], and a compiler for efficient sparse shape operators must do the same. Kjolstad et al. [2019] introduced a simple loop IR and scheduling rewrites that allow for avoiding discordant traversals, and BURRITO uses that IR (CIN) and supports the same transformations, for the same reasons.

Sparse Array Libraries. There are a number of sparse array libraries [Abadi et al. 2016; Paszke et al. 2019; Virtanen et al. 2020] that implement some number of shape operators. We compare to the most complete of these, `scipy.sparse` and `pydata/sparse`, in Section 7. These array libraries are feature-incomplete, generally only supporting a small number of array formats and a small number of operators. BURRITO can be used to generate custom shape operator implementations for each of these libraries, or replace them entirely.

Array Languages. There are decades of work in dense array programming languages [Bader and Kolda 2008; Harris et al. 2020; Henriksen et al. 2017; Iverson 1962; Liu et al. 2022; Ragan-Kelley et al. 2013; Steuwer et al. 2017], many of which support shape operators, but only for dense arrays. For many of these languages, shape operators correspond to zero-cost array metadata edits, and do not require iteration over the data like sparse shape operators do.

Staged Compilation. While the database community typically uses the iterator model as a runtime technique, recent work [Tahboub et al. 2018; Tahboub and Rompf 2020] applies ideas from partial evaluation to enable using the iterator model for code generation. BURRITO's code generation can be seen as an application of the same idea to compiling iteration over sequence expressions.

9 CONCLUSION

We describe the first compiler for a sparse array programming language with both shape and compute operators. We show how a simple declarative array language can be compiled to imperative loops over sequences, how to generate optimized loops that coiterate these sequences, and lastly, how to generate data-structure-specific code via a simple iterator model. With these ideas, sparse array programming moves one step closer to the completeness that dense array programming systems have long since achieved.

ACKNOWLEDGMENTS

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

Hameer Abbasi. 2018. Sparse: a more modern sparse array library. In *Proceedings of the 17th python in science conference*. 27–30.

Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3579990.3580020

Brett W. Bader and Tamara G. Kolda. 2008. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.

Aart 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.* 19, 4, Article 50 (sep 2022), 25 pages. https://doi.org/10.1145/3544559

Aart J.C. Bik. 1996. *Compiler Support for Sparse Matrix Computations*. Ph. D. Dissertation. Department of Computer Science, Leiden University. ISBN 90-9009442-3.

Aart J.C. Bik, Peter M.W. Knijenburg, and Harry A.G. Wijshoff. 1994. Reshaping Access Patterns for Generating Sparse Codes. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC '94)*. Springer-Verlag, Berlin, Heidelberg, 406–420.

William E. Boyse and Andrew A. Seidl. 1996. A Block QMR Method for Computing Multiple Simultaneous Solutions to Complex Symmetric Systems. *SIAM Journal on Scientific Computing* 17, 1 (1996), 263–274.

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. https://doi.org/10.1145/3276493

Iain S. Duff, A.M. Erisman, and J.K. Reid. 1990. *Direct Methods for Sparse Matrices*. Oxford Science Publications, Oxford.

Alan George and Joseph W.H. Liu. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Englewood Cliffs, New York.

G. Graefe. 1994. Volcano— An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (feb 1994), 120–135. https://doi.org/10.1109/69.273032

Fred G. Gustavson. 1972. Some Basic Techniques for Solving Sparse Systems of Linear Equations. In *Sparse Matrices and Their Applications*, Donald J. Rose and Ralph A. Willoughby (Eds.). Plenum Press, New York, NY, 41–52.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

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

Kenneth E. Iverson. 1962. A Programming Language. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference* (San Francisco, California) *(AIEE-IRE '62 (Spring))*. Association for Computing Machinery, New York, NY, USA, 345–351. https://doi.org/10.1145/1460833.1460872

Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, New York, NY, USA, 180–192.

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

Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (jun 2023), 25 pages. https://doi.org/10.1145/3591268

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., Red Hook, NY, USA.

Sergio Pissanetsky. 1984. *Sparse Matrix Technology*. Academic Press, London.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Justin Solomon. 2015. *Numerical Algorithms: Methods for Computer Vision, Machine Learning, and Graphics*. A. K. Peters, Ltd., USA.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, New York, NY, USA, 74–85. http://dl.acm.org/citation.cfm?id=3049841

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 307–322. https://doi.org/10.1145/3183713.3196893

Ruby Y. Tahboub and Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2103–2118. https://doi.org/10.1145/3318464.3389701

Reginal P. Tewarson. 1973. *Sparse Matrices*. Academic Press, New York, NY.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. https://doi.org/10.1145/3582016.3582047

Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (dec 2022), 26 pages. https://doi.org/10.1145/3566054

Zahari Zlatev. 1991. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publishers, Dordrecht.