Compilation of Shape Operators on Sparse Arrays

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

We show how to build a compiler for a sparse array language that supports shape operators such as 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: our generated shape operators achieve geometric mean speed-ups of 1.66×–15.3× when compared to hand-written kernels in `scipy.sparse` and 1.67×–651× when compared to generic implementations in `pydata/sparse`. For operators that require data structure conversions in these libraries, our generated code achieves geometric mean speed-ups of 7.29×–13.0× when compared to `scipy.sparse` and 21.3×–511× when compared to `pydata/sparse`. Finally, our evaluation demonstrates that fusing shape and compute operators improves the performance of several expressions by geometric mean speed-ups of 1.22×–2.23×.

CCS Concepts: • **Software and its engineering → Source code generation**; **Domain specific languages**.

Additional Key Words and Phrases: sparse array programming, sparse iteration theory, sparse data structures

ACM Reference Format:
Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. 2024. Compilation of Shape Operators on Sparse Arrays. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 312 (October 2024), 27 pages. https://doi.org/10.1145/3689752

1 Introduction

Shape operators are the type casts of array programming. The shape of an array determines which operations are valid as 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 [30], reshaping tensors in neural networks [36], and slicing images in biomedical computing [8]:

Authors' Contact Information: Alexander J Root, Stanford University, Stanford, USA, ajroot@stanford.edu; Bobby Yan, Stanford University, Stanford, USA, bobbyy@stanford.edu; Peiming Liu, Google Research, Seattle, USA, peiming@google.com; Christophe Gyurgyik, Stanford University, Stanford, USA, cpg@stanford.edu; Aart J.C. Bik, Google Research, Mountain View, USA, ajcbik@google.com; Fredrik Kjolstad, Stanford University, Stanford, USA, kjolstad@stanford.edu.

ACM 2475-1421/2024/10-ART312
https://doi.org/10.1145/3689752

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

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

As a result, dense array programming systems [16, 19, 25] have ample support for shape 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 [2, 37] support some shape operators, the implementations reduce to a small set of hand-written kernels. The data structure conversions required by these reductions incur a significant performance cost. Sparse tensor algebra compilers, on the other hand, [3, 22, 38, 39] lack a complete set of shape operators.

Hand-engineering sparse shape operators faces a significant scaling problem: each shape operator must be implemented for each sparse data structure, and it is not feasible to implement the full Cartesian product. Prior work on sparse tensor algebra compilation proposes a promising direction. Kjolstad et al. [22] and Chou et al. [9] show how to decouple compute operators from sparse data structures by establishing a data representation interface that sparse tensor algebra is compiled to and that each sparse data structure implements. This separation linearizes the quadratic dependency between algorithm and data representation and lets users generate sparse tensor algebra kernels for any combination of input and output data structures. Such techniques have not yet been developed for sparse shape operators. The key challenge lies in establishing a compilation model that supports generating code that iterates over, and computes on, combinations of reshaped, concatenated, and sliced views of sparse arrays backed by irregular data structures.

While recent work on compilers for sparse tensor algebra [3, 5, 22] addresses the sparse compute problem, they do not support shape operators except for limited cases (e.g., Taco supports slicing unfused array operands [18] and Looplets [3] supports concatenation without fusion). The challenge is that shape operators result in significantly more complicated iteration patterns, requiring a general unified representation of both shape operators and computation. On the other hand, libraries of hand-written kernels, such as `scipy.sparse` [37] and `pydata/sparse` [2] 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. These libraries also do not support operator fusion, which is an important optimization to achieve high performance for complicated expressions.

We introduce a compiler-based approach to providing general support for sparse shape operators, extending 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 4), describe compilation to a new intermediate representation (IR) for iterating over sparse data structures (Section 5), and outline the compilation process from our IR to efficient CPU code (Section 6 and Section 7). 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 4) to a high-level loop language that expresses fusion across operators (Section 5).
(2) A state-driven algorithm for generating optimized `while` loops that coiterate through reshaped iteration spaces (Section 6).

[1]A Burrito is what you get when you *reshape* a Taco.

(3) A compilation approach for generating iteration code that abstracts both data structures and reshaped iteration spaces, based on a simple iterator model (Section 7).

Burrito generates code competitive with hand-written libraries on unfused shape operators, and generally outperforms these libraries when given sparse kernels that offer fusion opportunities. Across six shape operators hand-written in `scipy.sparse` and `pydata/sparse`, Burrito performs 1.66×–15.3× and 1.67×–651× faster, respectively. For shape operators whose library implementations convert between data structures, Burrito-generated code outperforms `scipy.sparse` by 7.29×–13.0× and `pydata/sparse` by 21.3×–511×. To evaluate the benefits of fusion, we perform a self-comparison and show that code that fuses shape and compute operators outperforms unfused code by 1.22×–2.23× on some benchmarks. The observed benefits are largely due to the removal of expensive intermediate tensor allocations, which we discuss further in Section 8.

		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)	✔	✔	✔	✔	✔	✔	✔	✔

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.

2 Sparse Tensor Algebra Compilation Background

Burrito builds on a line of prior work on sparse tensor algebra compilation in the Taco compiler [22]. In this section, we give the background necessary to understand our contributions.

2.1 Tensor Index Notation

Taco decouples the language for expressing computation, known as *tensor index notation*, from the language that describes the physical layout of the sparse data structures backing those tensors, known as the format language [9]. Tensor index notation is a simple, declarative language for writing tensor algebra that supports element-wise addition, multiplication, broadcasts over tensor dimensions, and reductions (summations) over tensor dimensions. For example, matrix multiplication can be written as $A_{ij} = \sum_k B_{ik}C_{kj}$, where each component of A_{ij} is the inner product of a row of B and a column of C. We extend tensor index notation to also support shape operators in Section 4.

2.2 Format Language

Taco's language for expressing sparse data structures requires users to specify tensor types via combinations of per-dimension components [9, 22]. The popular coordinate (COO) format is simply a list of coordinate-value tuples, and can be expressed as a Compressed-Singleton matrix for the 2D case. As another example, the Compressed-Sparse-Row (CSR) [35] matrix format is expressed as Dense-Compressed, because the rows are densely stored but the columns of each row are compressed. We refer the reader to Chou et al. [9, Section 2] for a survey of tensor storage formats. Burrito uses this same format language for describing sparse data structures, with a slight change introduced by Kovach et al. [23], described in Section 7.1.

2.3 Concrete Index Notation

Tensor index notation is lowered to an intermediate representation (IR) called the Concrete Index Notation (CIN) [21], which describes tensor computation as per-dimension loops over intersections and unions of tensor coordinates. For example, a vector addition, $a_i = b_i + c_i$, requires a loop over the *union* of the non-zero coordinates in vectors b and c, because the output has a non-zero value when *either* operand is non-zero. This is denoted $i_b \cup i_c$, where i_b denotes the non-zero i coordinates in vector b. Likewise, a multiplication compiles to a loop over the *intersection* of non-zero coordinates, because the output has a non-zero only where *both* operands contain non-zero values. Generating a set expression corresponding to loop bounds is done on a per-dimension basis in Taco, and can be used to generate fused loops as well (e.g. multiply-add, $a_i = b_i c_i + d_i$ where a single loop iterates over the set expression $(i_b \cap i_c) \cup i_d$).

Consider an expression that adds a vector to the result of a matrix-vector multiplication, $y_i = b_i + \sum_j A_{ij} x_j$. This expression can be compiled to a loop over i coordinates that coiterates over b and A's i dimension, with a nested reduction loop over j coordinates that coiterates over A's j dimension and x, as shown in Figure 2. These loops iterate over all values of i such that b contains a non-zero element *or* A contains a non-empty row, and then iterates over all values of j such that A has a non-empty column and x contains a corresponding non-zero value, computing the SpMV reduction before computing and storing the sum of the two values. These loops are then lowered to loops that coiterate each of these three data structures, as described in the next sections. We refer the reader to Kjolstad et al. [21] for a more thorough description of CIN along with more examples.

```
forall i ∈ i_b ∪ i_A
  forall j ∈ j_A ∩ j_x
    t += A(i_A, j_A) * x(j_x)
  y(i) = b(i_b) + t
```

Fig. 2. CIN for $y_i = b_i + \sum_j A_{ij} x_j$.

Note that Taco supports loop scheduling operations on CIN such as loop reordering and insertion of temporary tensors [21] as well as more complicated operations such as loop tiling and parallelization [29]. While the prototype Burrito compiler supports loop reordering, it does not implement a full-fledged scheduling language, which we see as orthogonal to this work.

2.4 Iteration Lattice

Iteration lattices [18, 22] are ordered state machines that enable reasoning about multi-way merging of ordered coordinate sets. They consist of ordered points, where each point is distinguished by a set of coordinates. Following CIN, i_b represents the set of non-zero coordinates in the ith dimension of b, while $\emptyset$ represents the empty set, i.e., there are no non-zero coordinates remaining. A point may also have child points that represent simplified versions of the parent's sequence expression.

```
A: Compressed , Compressed
b: Compressed
x: Compressed
```

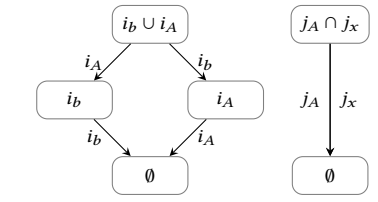


Fig. 3. Iteration lattices generated for $y_i = b_i + \sum_j A_{ij} x_j$ from Figure 2.

This data structure is used to implement an important component of sparse tensor algebra compilation, namely the *coiteration optimization*, which produces loops and control flow that handle the cases where some sparse tensors do not contain values for certain coordinates. For each loop in CIN, the set expression describing its iteration space is used to build an iteration lattice representing cases where some operands do not contain elements. Consider iteration over the union of two coordinate sets, generated from sparse vector addition. If one operand contains a coordinate that the other does not, then no addition needs to be performed, only a copy of the non-zero value from the source operand to the output vector. Likewise, if either vector runs out of elements, the program should move to code that only

```
1  while i ← i_b ∪ i_A
2   if i == i_b && i == i_A:
3    while j ← j_A ∩ j_x
4     if j == j_A && j == j_x:
5      t += A(i_A, j_A) * x(j_x)
6     y(i) = b(i_b) + t
7   elif i == i_A:
8    while j ← j_A ∩ j_x
9     if j == j_A && j == j_x
10     t += A(i_A, j_A) * x(j_x)
11   y(i) = t
12  elif i == i_b:
13   y(i) = b(i_b)
14 while i ← i_b
15  y(i) = b(i_b)
16 while i ← i_b ∪ i_A
17  while j ← j_A ∩ j_x
18   if j == j_A && j == j_x:
19    t += A(i_A, j_A) * x(j_x)
20  y(i) = t
```

(a) Pseudocode of the generated loops.

```
1  while (iA < iA_e) {
2   int i = A_crd0[iA]);
3   int jA = A_pos1[jA];
4   int jA_end = A_pos1[jA+1];
5   int jx = x_pos[0];
6   int jx_end = x_pos[1];
7
8   double t = 0.0;
9   while (jA < jA_end && jx < jx_end) {
10   int j = min(A_crd1[jA], x_crd[jx]);
11   if (j == A_crd1[jA] && j == x_crd[jx]) {
12    t += A_vals[jA] * x_vals[jx];
13   }
14   jA += (j == A_crd1[jA]);
15   jx += (j == x_crd[jx]);
16  }
17
18  y_vals[i] = t;
19  iA++;
20 }
```

(b) C code generated for lines 16-20 of **(a)**.

Fig. 4. Optimized loops generated for $y_i = b_i + \sum_j A_{ij}x_j$ from the lattices in Figure 3.

iterates over the non-empty vector, copying the non-zero values into the output vector. Similarly, for sparse vector multiplication, the multiplication only needs to be performed if both vectors contain the same non-zero coordinate. If either vector runs out of elements, then the multiplication is complete.

We show the iteration lattice for our running example, $y_i = b_i + \sum_j A_{ij}x_j$, in Figure 3. In the first lattice, over a union, if i_A runs out of elements (A runs out of non-empty rows), then control flow passes to a loop over only i_b. Likewise, if i_b runs out of elements (b runs out of non-zero values), then control flow passes to a loop over only i_A.

In this work, we extend iteration lattices to represent iteration over more complicated coiteration expressions than Taco, containing operations more complex than just union and intersection. We define these sequence expressions in Section 5, provide a construction algorithm for generalized iteration lattices, and show how to use them to compile a loop in CIN to loops that efficiently coiterate sparse data structures in Section 6.1.

2.5 Generated Code

While our compilation pipeline relies on many ideas from Taco, it uses a different code generation approach. Therefore, we will not describe how Taco generates C code. Instead, we provide an example of the code that both Taco and Burrito generate[2] for our running add-SpMV example. Figure 4a shows the coiterating loops for the example in pseudocode that abstracts away from concrete data structures. Figure 4b shows C code that implements the last nested while loop in the pseudocode, highlighting the complex nature of code that coiterates irregular data structures. Section 6 describes our algorithm for compiling iteration lattices to loops.

3 Overview

The compilation approach we describe in this paper compiles an array language to C code. The array language is tensor index notation, as used in many prior tensor algebra compilers, extended with shape operators. Like in Kjolstad et al. [22], our array language operates on *logical* arrays

[2]Burrito generates the same code as TACO for expressions that do not contain shape operators.

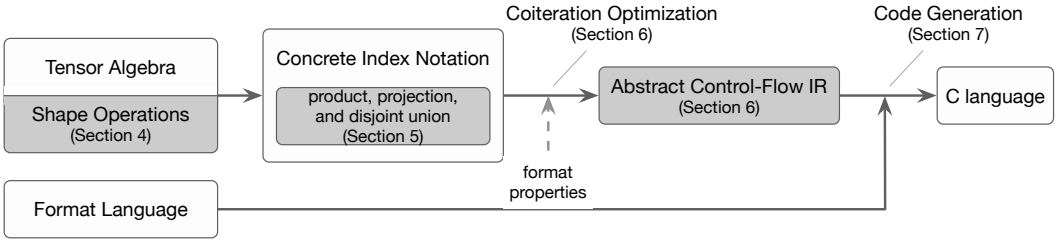


Fig. 5. The Burrito compilation approach compiles an array language to C code in three steps through two loop-based intermediate representations over logical arrays. Section references mark our contributions.

that abstract away the details of the *physical* data structures. Burrito separately accepts a format language as in Chou et al. [9] for describing the physical data structures (see Section 2.2).

Figure 5 shows an overview of our compilation approach, implemented in the Burrito compiler. Compilation consists of three lowering steps through two intermediate representations (IRs). The first step lowers the array language to Concrete Index Notation (CIN) (see Section 2.3). 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 expressed the iteration pattern of compute operators as intersections and unions [22] as well as complements and slicing [18]. In order to support the complete set of shape operators, our work adds three new sequence operators: set product, set projection, and disjoint union. Thus, the full set of operators is

$$\underbrace{(\text{intersection} + \text{union})}_{\text{Kjolstad et al. [22]}} + \underbrace{(\text{complement} + \text{slicing})}_{\text{Henry et al. [18]}} + \underbrace{(\text{product} + \text{projection} + \text{disjoint union})}_{\text{[this work]}}$$

CIN loops are then lowered to the control-flow intermediate representation (CFIR), a new IR that expresses complex control-flow while still abstracting away the concrete data structures. Each forall is turned into successive while loops over sequence expressions, each of which iterates over progressively simplified loop bodies as sparse arrays run out of non-zero coordinates. This is made possible through our generalization of the iteration lattices introduced in Kjolstad et al. [22], which represent iteration over combinations of sparse coordinate sets. The final step lowers CFIR to C code using a compile-time iterator model inspired by Kovach et al. [23], and replaces logical arrays by the physical data structures described in the format language.

4 Shape Operators

Burrito extends the tensor index notation described in Section 2.1 to support four primitive shape operators, which compose to express the shape operators supported by the popular `numpy` [16] array processing library. These primitives are collapsing and splitting (which together can implement `reshape`), concatenation, and slicing. Figure 6 provides the syntax of the complete algorithm language supported by Burrito. This section provides a high-level semantics for these operators by describing how array coordinates in the operands map to an array coordinate in the result of the expression. We then provide a shape inference algorithm for detecting the validity of an array program via typing rules.

4.1 Operator Definitions

The shape of a logical n-dimensional array is an n-dimensional hyper-rectangle where each dimension has a size. Tensor index notation labels each dimension via *index variables*. Shape operators combine and construct index variables, thereby combining and constructing the corresponding dimensions those variables label. We define each shape operator with respect to the coordinate

```
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. 6. 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. Operators marked in red are our additions.

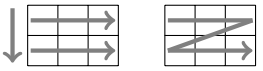
mapping it induces from its operands to its result. Note that our operators are all binary (e.g. binary concatenation), as Burrito supports *fusion by default*.

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. Consider an array expression E representing a three-dimensional array that is indexed by variables $\{i, j, k\}$. The following coordinate mapping collapses (flattens) the i and j index variables into an index variable l:

$$(x, y, z) \in E \implies (x * |j| + y, z) \in \texttt{collapse}(E, (i, j) \to l)$$

where $|j|$ is the size of the dimension labelled by index variable j (the dimension of coordinate y).

For example, a matrix `B` with rows labeled `i` and columns labeled `j` can be row-major collapsed via `c(k) = collapse(B(i,j),(i,j) → k)`, producing the iteration pattern in the figure to the right, or be column-major collapsed via `c(k) = collapse(B(i,j),(j,i) → k)`.



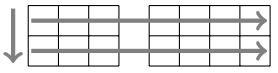
Concatenate. The `concat` operator is used to stack two arrays along a dimension. Consider two array expressions, E_0 and E_1, representing two-dimensional arrays that are indexed by $\{i, j\}$ and $\{i, k\}$, respectively. Concatenating these arrays along the j and k dimensions produces the following coordinate mapping:

$$(x, y) \in E_0 \implies (x, y) \in \texttt{concat}((E_0, E_1), (j, k) \to l)$$

$$(z, w) \in E_1 \implies (z, w + |j|) \in \texttt{concat}((E_0, E_1), (j, k) \to l)$$

where $|j|$ is the size of the dimension labelled by index variable j (the dimension of coordinate y).

For example, a program that horizontally concatenates two arrays can be expressed as: `D(i,l) = concat((B(i,j),C(i,k)), (j,k) → l)`. We provide a visual description of this operator in the figure to the right. `D` 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.[3] Consider an array expression E representing a two-dimensional array that is indexed by variables $\{i, j\}$. The following coordinate mapping splits j into index variables k and l:

$$(x, y) \in E \implies (x, \frac{y}{|l|}, y \bmod |l|) \in \texttt{split}(E, j \to (k, l))$$

where $|l|$ is the size of the dimension labelled by index variable l.

For example, consider splitting a 1-dimensional array into a 2-dimensional array: `C(i,j) = split(b(k), k → (i,j))`. For this row-wise splitting, the coordinate 1 in `b` corresponds to $(0, 1)$ in `C`. This produces

[3]The `split` and `collapse` operators are building blocks that can be used to support the common *reshape* operator.

the iteration pattern in the bottom of the figure to the right. For a column-wise splitting: `C(i,j) = split(b(k), k → (j,i))`, the coordinate 1 in `b` corresponds to $(1, 0)$ in `C`. 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. Consider an array expression E representing a two-dimensional array that is indexed by variables $\{i, j\}$. The following coordinate mapping slices into E's second dimension:

$$(x, y) \in E \;\; \& \;\; s \leq y < e \;\; \& \;\; \exists z \in \mathbb{N},\; y = s + r * z \implies (x, z) \in \texttt{slice}(E, j \to k, (s, e, r))$$

Unlike the other operators, slicing is used to discard elements at certain coordinates. 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, the whole-number multiple is the new coordinate in the output dimension.

For example, if a user wished to slice out the second through the fourth elements of a 1-dimensional array, they could write the slicing expression `c(i) = slice(b(j), j → i, (2,5,1))`, depicted on the right. If they wanted the first half of an array, they could write `a(i) = slice(b(j), j → i, (0,J/2,1))`. Note that Henry et al. [18] introduced slicing on sparse arrays only, while Burrito supports slicing any array expression, including intermediate computations, e.g., slicing the result of a multiplication or a flattened array.

4.2 Shape Inference

The shape of an expression in our array language is a part of its *type*, along with the type of the scalar elements. An array compiler requires a shape inference algorithm for type-checking programs. Shape inference from tensor index notation is quite straight-forward: element-wise expressions produce a result with the same shape as its operands 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 shape of an array. The new shape operators defined in the prior section construct new dimensions, and we provide inference rules for reasoning about these new dimensions in order to check for program correctness.

As described previously, the shape of an array specifies the number of dimensions and the size of each dimension. Let S denote the shape of an array as an unordered set of indices, each of which labels a dimension with a name and size. And let $|i|$ denote the size of the dimension indexed by i.

The inference rules are provided in Figure 9. For illustration, the `collapse` inference rule is:

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

The first two constraints, `i` $\in S$ and `j` $\in S$, require the source indices `i` and `j` that are being collapsed to be contained within the shape S of the array expression `a`. Next, `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, $|\texttt{i}| \cdot |\texttt{j}| = |\texttt{k}|$, requires that the size of the dimension of the constructed index, $|\texttt{k}|$, is the product of the size of dimensions represented by the source indices, $|\texttt{i}|$ and $|\texttt{j}|$. This means that the output array of the `collapse` has the same number of discrete elements 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 element-wise multiplication of a vector with a flattened matrix `c(k) = a(k) * collapse(B(i,j), (i,j) → k)`. This program is well-typed: the shape of the argument to `collapse` is $S_{\texttt{B(i,j)}} = \{\texttt{i},\texttt{j}\}$, and `i` $\in S_{\texttt{B(i,j)}}$ and `j` $\in S_{\texttt{B(i,j)}}$ are trivially true. Likewise, the constructed index is

$$\frac{}{\mathsf{a}(I) : I} \qquad \frac{E : S \quad i \in S \quad j \in S \quad k \notin S \quad |i| \cdot |j| = |k|}{\texttt{collapse}(E, (i,j) \to k) : (S - \{i,j\}) \cup \{k\}}$$

$$\frac{E : S \quad i \notin S}{\texttt{broadcast}(i, E) : S \cup \{i\}} \qquad \frac{E_0 : S_0 \quad E_1 : S_1 \quad S_0 - \{i\} = S_1 - \{j\} \quad k \notin S_0 \quad |i| + |j| = |k|}{\texttt{concat}((E_0, E_1), (i,j) \to k) : (S_0 - \{i\}) \cup \{k\}}$$

$$\frac{E : S_0 \quad E_1 : S_1 \quad S_0 = S_1}{E_0 \texttt{<op>} E_1 : S_0} \qquad \frac{E : S \quad i \in S \quad j \notin S \quad k \notin S \quad |i| = |j| \cdot |k|}{\texttt{split}(E, i \to (j,k)) : (S - \{i\}) \cup \{j,k\}}$$

$$\frac{E : S \quad i \in S}{\texttt{sum}(i, E) : S - \{i\}} \qquad \frac{E : S \quad i \in S \quad j \notin S \quad \frac{e - s + (r-1)}{r} = |j|}{\texttt{slice}(E, i \to j, (s, e, r)) : (S - \{i\}) \cup \{j\}}$$

Fig. 9. Shape inference rules for shape and compute operators. An array's shape is determined by the set of indices that index it. 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 index, while splitting divides a single source index into two indices. Concatenation accepts a pair of expressions and pair of indices, concatenating the two expressions along the two indices to produce a single index. Slicing relabels a source index into a smaller index.

not in the source shape, $\mathtt{k} \notin S_{\mathtt{B(i,j)}}$. Lastly, the dimensionality constraint is satisfied if $|\mathtt{i}| \cdot |\mathtt{j}| = |\mathtt{k}|$. It follows that both operands to the multiplication have shape $\{\mathtt{k}\}$, so it is also well-typed.

5 Sequence Expressions

The first step in our compilation model is to generate nested loops that each iterate over the coordinates in one or more dimensions of one or more logical arrays. We extend the Concrete Index Notation (CIN) of Kjolstad et al. [21] (see Section 2.3) with a more advanced language for describing the iteration domain of each loop, which we call a *sequence expression*. Our extensions go beyond the unions and intersections of the coordinates in array dimensions described in prior work, to support the complex iteration patterns introduced by shape operators. For the remainder of this work, we refer to the coordinates of array dimensions as *sequences*, and the expressions that represent combinations of these sets using set operators as *sequence expressions*.

In this section, we provide a construction algorithm for generating sequence expressions from the expression language described in Section 4.1 and a semantics for the sequence *combinators* in the sequence expression language. Sequence expressions are closely related to the coordinate mappings described by the high-level shape operators in Section 4.1, and express both compute and shape operators, allowing for a natural fusion of operators. However, where the high-level compute and shape operators describe how to combine whole arrays, sequence expressions describe the iteration domain of a single loop in the CIN. For example, prior work observed that a tensor addition leads to the union of two sequences. Similarly, we show that a `collapse` leads to the Cartesian product of two sequences, where the coordinates in the resulted tuples are combined using a strided offset formula. Each shape operator corresponds to a computation on the coordinates of non-zeros of its operands, and fundamentally change the iteration spaces of the loops.

5.1 Sequence Combinator Semantics

We provide the full grammar for CIN below. CIN includes `forall` loops, statement pairs, consumer-producer `where` statements, assignments, and reductions. The `forall` statement iterates over a sequence expression that may contain sparse as well as dense sequences. The `where` statement is similar to a let statement. It constructs an intermediate tensor (or scalar) on the right hand side that can be used on the left hand side, and is thus a construct for creating intermediate tensors. We refer the reader to Kjolstad et al. [21] for more details on CIN, as 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 | πk( seq, int ) | seq [ int:int:int ]
```

Each sequence combinator describes computation on ordered sets of coordinates. Sequence combinators corresponding to tensor computations (i.e. union and intersection) join two sequences, but sequence combinators corresponding to shape operators must additionally *transform* the coordinates from the sequences they combine. The semantics of these combinators are important both for understanding how shape operators combine iteration spaces, and for compiling down to loops over irregular data structures. These combinators 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. The remainder of this section describes the semantics of the sequence combinators.

product: $A \times B$ is similar to a Cartesian product on two sequence expressions. Intuitively, product corresponds to: for each element in A, step through each element in B, and for the tuple (a, b), apply a strided offset formula to product a single coordinate value. Formally:

$$A \times B = \{\ a \cdot |B| + b \ \mid \ a \in A \wedge b \in B\ \}$$

Note that this combinator looks quite similar to the semantics defined for `collapse`, and indeed it is used to represent the iteration induced by `collapse` operators.

concatenation: $A \sqcup B$ combines two sequence expressions as a union of the first with the elements of the second offset by the logical size of the first. This combinator is isomorphic to a disjunctive union, but produces a sequence with a logical size that is the sum of the logical sizes of the two input sequences. Intuitively, this combinator corresponds to first stepping through each element in A and then through each element in B (with an offset). Formally:

$$A \sqcup B = A \cup \{\ |A| + b \mid b \in B\ \}$$

This combinator has similar semantics as `concat`, and is used to compile that operator.

projection: $\pi_k(A, J)$ is the inverse of a product combinator. It applies the inverse of the strided offset formula to elements of A to produce a tuple from a single coordinate, with the shape $(\frac{|A|}{J}, J)$. It then produces *two* sequences, controlled by the projection index, k (which is either 0 or 1). This combinator is isomorphic to the set projection operation. Intuitively, a projection of a sequence produces two sequences that each iterate over sub-spaces of the original sequence. The first sequence $\pi_0(A, J)$ is a sequence of size $\frac{|A|}{J}$, and the second, $\pi_1(A, J)$ is a sequence of size J. Formally:

$$\pi_0(A, J) = \{\ a/J \ \mid \ a \in A\ \} \quad \pi_1(A, J) = \{\ a \bmod J \ \mid \ a \in A\ \}$$

This combinator corresponds to the `split` operator, and is used to compile that operator.

slice: $A[s{:}e{:}r]$ is a filtering combinator. Intuitively, slicing removes all elements from a sequence that are not a stride of r away from the start s, and elements greater than or equal to e. Formally:

$$A[s{:}e{:}r] = \{\ x \ \mid \ a \in A \wedge s \leq a < e \wedge a = s + r \cdot x\ \}$$

This combinator represents the semantics of the `slice` operator, and is used to compile it.

5.2 Lowering Shape Operators to Sequence Combinators

Burrito's lowers array expressions to sequence expressions using a recursive algorithm that pattern matches on the array expression to produce a sequence expression for the loop corresponding to each index variable. Figure 11 defines this algorithm, which accepts an array expression `expr` from the grammar defined in Figure 6, along with an index variable `i`. It then generates a sequence expression that represents the output coordinates in the `i`th dimension of `expr`.

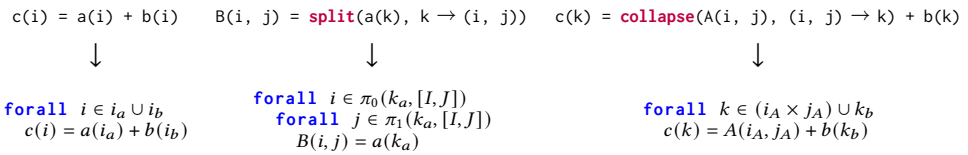


Fig. 10. CIN for various kernels.

Consider compiling the rightmost kernel in Figure 10, 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 in Figure 11 first applies the rule on line 4, then the rule on line 9, and lastly a series of base cases (line 3). These rules construct the sequence $(i_A \times j_A) \cup k_b$, the loop bounds in the rightmost loop of Figure 10. This sequence expression means that the resulting 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, and the iteration pattern required to produce the result.

```
1  func Seq(expr, i):
2   match expr with
3   | array(I) ↦ i_array
4   | a + b ↦ Seq(a, i) ∪ Seq(b, i)
5   | a * b ↦ Seq(a, i) ∩ Seq(b, i)
6   | sum(j, a) ↦ Seq(a, i)
7   | broadcast(i, a) ↦ 𝕌_i
8   | broadcast(j, a) ↦ Seq(a, i)
9   | collapse(a, (j, k) → i) ↦ Seq(a, j) × Seq(a, k)
10  | collapse(a, (j, k) → l) ↦ Seq(a, i)
11  | concat((a, b), (j, k) → i) ↦ Seq(a, j) ⊔ Seq(b, k)
12  | concat((a, b), (j, k) → l) ↦ Seq(a, i) ∪ Seq(b, i)
13  | split(a, j → (i, k)) ↦ π0(Seq(a, j),|k|)
14  | split(a, j → (k, i)) ↦ π1(Seq(a, j),|i|)
15  | split(a, j → (k, l)) ↦ Seq(a, i)
16  | slice(a, j → i, (s, e, r)) ↦ Seq(a, j)[s:e:r]
17  | slice(a, j → k, (s, e, r)) ↦ Seq(a, i)
```

Fig. 11. Generation of a sequence expression from an array expression expr, for a dimension labelled by index i.

6 Lowering to Coiterating Loops

To generate code that efficiently iterates over a sequence expression, Burrito reasons about when a sparse sequence runs out of elements and whether it contains a particular coordinate. In order to represent this reasoning, we introduce a second intermediate representation, CFIR (Control Flow Intermediate Representation), which contains more complex control-flow constructs than CIN's forall loops. CFIR allows Burrito to represent coiterating optimizations while still abstracting away the concrete details of the underlying physical data structures. CIN loops are compiled to a sequence of progressively simpler CFIR *while loops* that each exits when a sequence (e.g. an array or reshaped view of an array) runs out of values. We first define CFIR before showing how to construct it via a generalized form of iteration lattices [18, 22]. 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 both abstract away details of the physical data structures underlying the logical arrays.

6.1 Control Flow Intermediate Representation

We provide the grammar for CFIR below. Its 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 | alloc a(I)
```

6.1.1 Loops. The `while` loops of CFIR iterate over a sequence expression until a sub-sequence runs out of elements. The drop-out semantics is useful because a sequence expression and loop body can be simplified to discard sub-expressions that have run out of values in earlier loops, thus

producing 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, a kernel that collapses a 2-dimensional array into a 1-dimensional array produces a single loop that iterates over the product of the 2D array's dimensions, as shown on the right.

```
while k ← i_A × j_A
  b(k) = A(i_A, j_A)
```

6.1.2 Conditionals. The second control-flow construct in CFIR is the conditional `switch` construct. When iterating over a sequence expression, some sub-expressions may not contain a coordinate that other sub-expressions 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, in sparse vector addition, the addition should only be performed when *both* vectors contain a coordinate, and the `switch` statement in the code to the right represents that guard.

```
while i ← i_a ∪ i_b
  switch i
    case i_a ∪ i_b :
      c(i) = a(i_a) + b(i_b)
    case i_a :
      c(i) = a(i_a)
    case i_b :
      c(i) = b(i_b)
```

6.1.3 Location. CFIR loops also support *locating* into sequences that support random-access (via the `with` operation) in order to randomly access array values. This construct is useful because, in certain circumstances, a sequence does not need to be fully iterated. For example, vector multiplication iterates over $i_a \cap i_b$. If $i_a \subseteq i_b$, it is sufficient to iterate over only i_a, and randomly access into i_b to get the value labeled by its coordinate. This is the iterate/locate optimization described by Kjolstad et al. [22], and is illustrated on the right. Iterating over a sparse sequence and locating into a dense sequence is asymptotically optimal, and the `with` operator allows CFIR to represent this optimization. The `with` construct allows Burrito to represent this pattern, as `with` $a = b$ means that b is being iterated over, once a value is computed, use that value to locate into sequence a. Burrito uses *format properties* to detect which sub-sequences support fast $O(1)$ locates. 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.

```
while i ← i_a with i_b = i_a
  c(i) = a(i_a) * b(i_b)
```

6.1.4 Pairs, Assignments, and Allocations. CFIR also supports sequences of statements (sequential computation) through the pair construct, assignments, compound assignments for reductions, and allocations of intermediate temporaries. These statements are placed inside `while` and `case` bodies to perform computation and data structure allocation.

6.2 Generalized Iteration Lattices

Iteration lattices [22] consist of an ordered set of lattice points and are used to generate the drop-out while loops in the previous subsection. In Burrito, a lattice point is labeled by a sequence expression, and represents iterating over that sequence. Each lattice point has children 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.

```
func ConstructLattice(seq):
  point = LatticePoint(seq)
  // Transition sub-sequences
  // (Section 6.2.1 and Figure 13)
  edges = Edges(seq)
  for sub in edges:
    // Simplification
    // (Section 6.2.2)
    r = remove(sub, seq)
    s = simplify(r)
    l = ConstructLattice(s)
    point.add_child(sub, l)
return point
```

Fig. 12. Top-down lattice construction.

While prior work [18, 22] provides a bottom-up lattice construction algorithm that requires filtering nodes to perform this simplification, we instead introduce a top-down construction algorithm

based directly on sequence simplification. We believe this algorithm is simpler and thus enables us to incorporate the additional complexity of our new sequence expressions. Figure 12 shows our algorithm for lattice construction, which follows a recursive top-down approach that generates progressively simpler sequence expressions to iterate over. For a given sequence expression, s, Burrito first finds all sub-sequences whose removal result in a simplification of s, called *edge sequences*. These sequences label the edges of the lattice from the point labelled by s. Then, for each edge sequence, remove it from s, simplify via set rules, and recursively construct an iteration lattice to point the edge to.

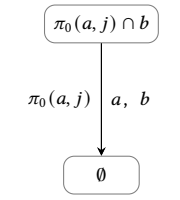
6.2.1 Edge Sequences. The algorithm for gathering edge sequences recurses on the structure of a sequence expression. Edge sequences are often sequences produced by array levels, but can also correspond to slices or projections. This is because a slice (or projection) can run out of coordinates before the sequence it is slicing (or projecting) runs out, and should thus induce a state transition. We give an algorithm for collecting edges from a sequence expression in the function `Edges` in Figure 13. It is a recursive algorithm that generates a set of sub-sequences that correspond to state transitions.

```
func Edges(seq):
 match seq with
 | i_array ↦ {seq}
 | a ∪ b ↦ Edges(a) ∪ Edges(b)
 | a ∩ b ↦ Edges(a) ∪ Edges(b)
 | a × b ↦ Edges(a)
 | a ⊔ b ↦ Edges(a)
 | π_k(a, j) ↦ {seq} ∪ Edges(a)
 | a[s:e:r] ↦ {seq} ∪ Edges(a)
```

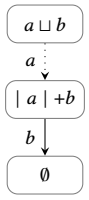
Fig. 13. Function for gathering a set of sub-sequences whose removal induces state transitions.

Consider a loop over the sequence $(a \cup b) \times c$, where all sequences are sparse. If a runs out of elements, the state transitions to a loop over $b \times c$. Likewise, if b runs out of elements, the state moves to a loop over $a \times c$. This iteration lattice is illustrated in Figure 14c. Notably, if c runs out of elements, $a \cup b$ must be stepped forward, and c reset, based on the semantics of product. This is why there is no edge corresponding to c running out, and why Figure 13 only grabs edges from the first operand of the product combinator.

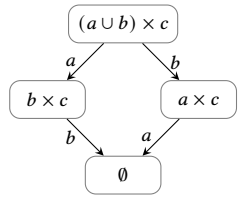
6.2.2 Sequence Simplification. Sequence simplification follows simple set rules, such as if one side of an intersection is empty, the entire intersection is empty. Likewise, if one side of a union



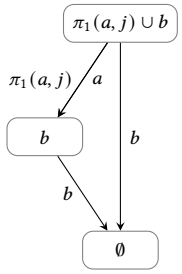
(a) Dense or sparse a and b



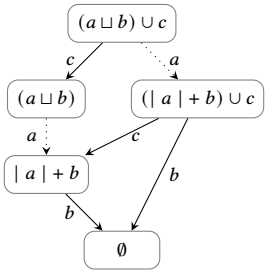
(b) Dense or sparse a and b



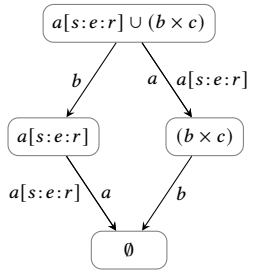
(c) Sparse a, b and c.



(d) Sparse a and dense b.



(e) Sparse a and c, dense b.



(f) Sparse a and b, dense c.

Fig. 14. Iteration lattices for several sequence expressions. Full lines are edges to sub-points, dotted lines are edges to non-sub-points (see Section 6.2.3 for the distinction).

is empty, then return the other side of the union. Our algorithm treats sparse-and-empty and dense-and-empty differently. For example, for a set expression $a \cup b$ with a as a dense sequence, the entire union must be empty when a becomes empty because the sequence must have been fully iterated when a dense sequence (which contains every coordinate) runs out of coordinates. This optimization can be seen in Figure 14d, where a dense sequence, b, is coiterated with a sparse sequence—when the dense sequence runs out, the entire union must run out. We provide the full list of rewrite rules in Appendix A.

Concatenation is a special case, as $a \sqcup b$ simplifies to a new (but simple) combinator, the offset $|a| + b$. This represents iteration over b, where all the coordinates are offset by the size of a's dimension. The offsets induced by concatenation are illustrated in Figures 14b and 14e.

6.2.3 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 14f, the sequence $a[s{:}e{:}r] \cup (b \times c)$ has two sub-points, labeled $a[s{:}e{:}r]$ and $(b \times c)$. 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 combinator. Consider the iteration lattice in Figure 14b. The concatenation combinator produces a state transition when a runs out, indicating a transition to start iterating over the offset elements of b[4]. However, the point $|a|+b$ is not considered a sub-point of $a \sqcup b$, because it represents a disjoint set (it requires a runs out, not just that a is missing an element). Edges to non-sub-points are denoted with a dotted edge in 14b and 14e. To compute the sub-points of a lattice point p, we gather all descendants that represent a subset of p's sequence. Note that a point is considered its own sub-point.

6.3 Lowering 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 simplified code. We provide the lowering algorithm from CIN forall loops to CFIR loops in Figure 15, discuss two of the methods used in constructing loops, and walk through two examples below.

```
func CompileForall(idx, seq, body):
  // Lattice construction in Figure 12
  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):
 build = λ sp: Compile(simplify(sp, body))
 // Case for each sub-point (Section 6.2.3)
 bodies = map(build, point.subs)
 // Find locators (Section 6.3.1)
 seq, loc = RemoveLocs(point.seq)
 if len(bodies) > 1: // Section 6.3.2
  body = Switch(idx, point.subs, bodies)
 else:
  body = bodies[0]
 return While(idx, seq, loc, body)
```

Fig. 15. Compilation of CIN to CFIR.

6.3.1 Removing Locators. As described in Section 6.1.3, when a sparse sequence is intersected with a dense sequence, the loop can be optimized by iterating over the sparse sequence and locating into the dense sequence. As in Taco, we recursively search for dense sequences and turn them into locators instead during loop construction (Figure 15). See [22, Section 5.2] for more details.

[4]The result is a *fissioned* loop, which is the most efficient way to iterate over concatenated arrays.

6.3.2 Building Conditionals. Note that some loop bodies do not require a conditional loop, such as when iterating over a single array. Conditionals are only needed when there are multiple possible states that can be transitioned to, which corresponds to the sequence operator having sub-points *other than itself*. Therefore, the construction algorithm for building a loop body only constructs a conditional `switch` statement if the sequence being iterated over has more than one sub-point (and therefore, there are multiple sub-states that needed to be considered).

6.3.3 CFIR Examples. Consider the CIN in Figure 16d, where the k loop iterates over the product of a dense iterator and a sparse iterator, unioned with another sparse iterator. This loop generates the iteration lattice in Figure 16e, which contains three non-empty states: the initial state (top), a state that iterates over only the product (middle left), and a state that iterates over only the second sparse iterator, k_b (middle right). This iteration lattice compiles to the three loops in Figure 16f,

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

(a) Array algorithm description.

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

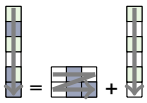
(d) Concrete index notation.

```
while k ← (iA × jA) ∪ kb
  switch k
    case (iA × jA) ∪ kb :
      c(k) = A(i, j) + b(k)
    case (iA × jA) :
      c(k) = A(i, j)
    case kb :
      c(k) = b(k)
while k ← iA × jA
  c(k) = A(i, j)
while k ← kb
  c(k) = b(k)
```

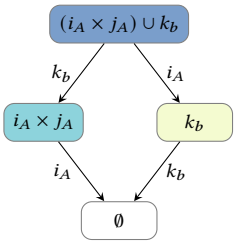
(f) Control-flow IR.

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

(b) Array format description.



(c) Logical iteration pattern.



(e) k's iteration lattice.

Fig. 16. 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 22.

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

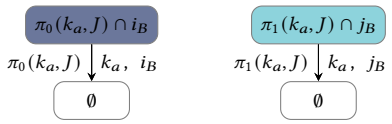
(a) Array algorithm description.

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

(b) Array format description.

$$\forall\, i \in \pi_0(k_a, J) \cap i_B$$
$$\forall\, j \in \pi_1(k_a, J) \cap j_B$$
$$C(i,j) = a(k) * B(i,j)$$

(c) Concrete index notation.



(d) Iteration lattices for i and j.

```
while i ← π0(ka, J) with iB = π0(ka, J)
  while j ← π1(ka, J) ∩ jB
    C(i, j) = a(k) * B(i, j)
```

(e) Control-flow IR.

Fig. 17. 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 23b for the final generated C code from this example.

corresponding to each of the three non-empty states. When the union can be simplified (when either the product or k_b run out of elements), control flow is passed to a loop over a simpler sequence expression. Inside the loop over the union, there are cases for each possible sub-state that the iteration could be in. For example, if the product contains a coordinate that k_b does not, the second case (lines 5–6 of Figure 16f) perform a simplified version of computation, with a copy replacing the addition. The cases within a loop correspond exactly to the sub-states that the loop's sequence has.

Likewise, consider a kernel that reshapes (splits) a 1D sparse array into a 2D view, and performs element-wise multiplication with a CSR matrix, as illustrated in Figure 17. Both CIN loops iterate over the intersection of a projection and a sequence from the CSR matrix. These loops generate the almost identical iteration lattices in Figure 17d. However, the first loop intersects a sparse sequence (the projection) with a dense sequence, i_B. Therefore, the iterate-locate optimization described in Section 6.1.3 is applied to produce a loop that locates into the dense sequence. The second loop intersects two sparse sequences, so there needs to be a `switch` statement inserted to handle the case where one sparse sequence trails behind the other. This results in the loop nest shown in Figure 17e.

7 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. [23], a representation used for compiling fused sparse tensor algebra that shows how to iterate over unions and intersections efficiently. This approach decouples the mathematical intuition behind iteration, a complex abstraction when handling sparse data structures, and the actual code generation. We first describe the iterator model for formats, then for sequence combinators, and lastly provide the complete code generation algorithm from CFIR to C code.

7.1 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 20 and slicing in Figure 21. Note that Figure 21 shows a generalization of prior work [18], which supported slicing only array dimensions, while our algorithm can generate code that iterates over the slice of *any* sequence.

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

Valid is a check that the sequence has not run out of coordinates, meaning no sub-sequence 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 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 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 23a).

Next steps the sequence forward to the next non-zero value. For array levels, unions, and intersections, this performs the same conditional updates as Taco (e.g. line 25 of Figure 22). For sequence combinators, we define modular code generators for stepping forward a sequence.

Locate moves physical iterators to point to a specific logical coordinate. Section 6.1.1 described cases where random-access into a sequence is asymptotically preferable, and the locate interface is used to support this optimization. Many sequence combinators are invertible and this property can be used to locate through them, and base data structure formats often support fast random access (e.g. dense or hashed formats).

7.1.1 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 18 and 19, respectively. Extending Burrito to support an additional sparse format only requires implementing the iterator interface for that format type. Our prototype compiler supports

```
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. 18. 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. 19. Iterator model for a compressed dimension.

```
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. 20. 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. 21. Iterator interface for sequence slices.

TACO's dense, compressed, and singleton level formats [9], which compose to express many sparse data structures, such as CSR, CSC, COO, CSF, DCSR/DCSC, and other variants.

```
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. 22. Compilation of the first loop of Figure 16f, which iterates over a collapsed CSR matrix, A, and adds it to a compressed vector, b.

```
func CompileCFIR(stmt):
 match stmt with
 | While (idx, seq, locs, body) ↦
  emit Init(seq)
  emit while (Valid(seq)) {
  emit int idx = Eval(seq);
  for a, b ∈ locs
   emit Locate(Eval(a), b)
  emit CompileCFIR(body)
  emit Next(seq) }
 | Switch (idx, seqs, bodies) ↦
  emit if (Equals(idx, seqs[0])) {
  emit CompileCFIR(bodies[0]) }
  for s, b ∈ zip(seqs, bodies)[1:]
   emit else if (Equals(idx, s)) {
   emit CompileCFIR(b) }
 | Pair (cfir0, cfir1) ↦
  emit CompileCFIR(cfir0)
  emit CompileCFIR(cfir1)
 | Assign (array, idxs, expr) ↦
  emit CompileWrite(array, idxs)
  emit CompileExpr(expr)
 | Reduce (array, idxs, expr) ↦
  emit CompileReduction(array, idxs)
  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 17e.

Fig. 23. (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.

7.1.2 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 20 and 21, respectively. These implementations show the composability of this interface. We also illustrate a labeled example of the full C code generated from Figure 16 in Figure 22, with line labels corresponding to the interface that has generated each line.

7.2 Compiling CFIR

We give the algorithm for compiling CFIR based on the iterator model in Figure 23a. 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 `switch` statements are used to evaluate which sub-state the iterator may currently be in, and generate C++ 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 both compute and shape operators. We provide an example of such code in Figure 23b, which is annotated with the line numbers from Figure 23a that have generated each line of the C++ code, in addition to the line-by-line annotations of Figure 22 that illustrates calls to iterator model components for the first CFIR loop in Figure 16f.

8 Evaluation

Our evaluation provides evidence that demonstrates the following claims:

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

We first describe the existing state-of-the-art libraries that we compare to in Section 8.1, provide our benchmarking methodology in Section 8.2, and then provide evidence for the above claims in the following sections. We implemented Burrito in Racket [12], a language for designing DSLs. Burrito generates C++ code that we call from our Python benchmarking infrastructure via nanobind [20].

8.1 Baselines

We compare Burrito-generated code to `scipy.sparse` [37] v1.11.2 and `pydata/sparse` [2] v0.15.1, the only libraries that we are aware of with support for both shape and compute operators. Both libraries follow a simple reduction approach for shape operators. 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 the CSR[5] matrix to COO[6] and then calls `COO.reshape`. Likewise, when a user requests a non-standard output format (e.g. *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. Such data reorganization approaches to generality naturally come at a performance cost.

For sparse tensor algebra alone, Burrito's compilation technique produces equivalent code as Taco. We therefore do not compare directly to Taco.

Note that both libraries we compare to call `numpy` [16] operators (handwritten C++ kernels) on arrays that represent the sparse arrays whenever possible, so our evaluation is largely comparing C++ loops to C++ loops, though there is a larger amount of Python overhead for small arrays in the

[5]The Compressed-Sparse-Row format.
[6]The COOrdinate list format.

libraries versus Burrito-generated code. Our evaluations are performed on CSR/CSC and COO matrices because `scipy` does not support other sparse types or higher-dimensional arrays. To the best of our knowledge, there are no other systems that support multiple sparse shape operators for comparisons, and Burrito generates the same loops as Taco for direct array slicing.

8.2 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 (both libraries and the Burrito-generated code are single-threaded programs). Our generated kernels are compiled with clang++ 14.0.3. Python 3.11.4 is used to run all benchmarks.

We evaluate on all real-valued matrices in the SuiteSparse [10] matrix data set, except the largest matrix (MOLIERE_2016) because our machine did not have sufficient memory. These matrices span several domains, including computer vision, structural engineering, economics, graph analytics, and computational fluid dynamics. The x-axis of all graphs is the number of non-zero coordinates in the SuiteSparse matrix being operated on.

In each benchmark, we report the minimum time out of 10 iterations, with a 5 second timeout per iteration. For benchmarks that require multiple matrices (e.g. 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 with elements shifted by a small number (10), due to the need for shape compatibility, as in prior evaluations of sparse tensor algebra [22]. Like Taco, 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), and `hstack` means *horizontal* stacking (concatenation along the second axis). Benchmarks label operands with their array types (e.g. "CSR" or "COO"), and single letter labels ("C" and "D") correspond to compressed and dense vectores, respectively.

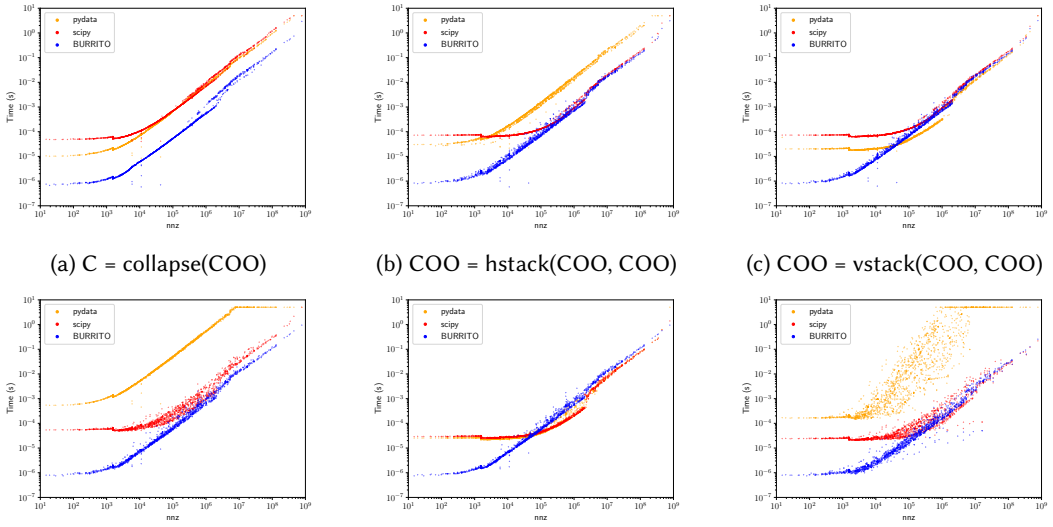
8.3 Comparison to Hand-written Kernels

We compare Burrito's generated shape operators to the shape operator implementations in `scipy.sparse` and `pydata/sparse` that *do not* perform data structure conversions, shown in Figure 24. This comparison shows that a compilation-based approach can match the performance of hand-written code.

Reshape. Figure 24a shows the performance of collapsing a 2D matrix into a 1D sparse vector[7]. Burrito outperforms `scipy.sparse` by geometric mean 15.3× and `pydata/sparse` by 10.5×. Both libraries perform this reshaping operation by a series of three calls to numpy operations over the same length of arrays, while Burrito generates a single loop that fuses the three operations.

Concatenation. Figures 24b, 24c, 24d, and 24e evaluates the performance of various forms of matrix concatenation. They have geometric mean speed-ups of 3.52× over `scipy.sparse` and 16.7× over `pydata/sparse` (Figure 24b), 4.48× and 1.69× (Figure 24c), 6.29× and 651× (Figure 24d), and 1.66× and 1.67× (Figure 24e) respectively. For vertically stacking matrices, the libraries sometimes outperform Burrito-generated code on larger matrices by up to 11.4× and 11.7× for `scipy.sparse` and `pydata/sparse` respectively. Upon investigation, we believe it is because the libraries rely on hand-vectorized C++ kernels for these operations (that do little more than `memcpy`s), while the C++ compiler used to compile Burrito-generated code did not produce the same vectorized loops. The general loop structure for these kernels are the same between the libraries and Burrito-generated code. Nonetheless, these benchmarks demonstrate that generated code can match or exceed the

[7] `scipy.sparse` does not directly support sparse vectors, but can represent them using a CSR matrix with a single row.



(a) C = collapse(COO) (b) COO = hstack(COO, COO) (c) COO = vstack(COO, COO)

(d) CSR = hstack(CSR, CSR) (e) CSR = vstack(CSR, CSR) (f) CSR = slice(CSR)

Fig. 24. Runtimes of shape operators for kernels that `scipy.sparse` `pydata/sparse` have hand-implemented kernels without performing data structure conversions. The performance shift around 10^6 non-zero values correspond to each system shifting to u64 for matrix indices instead of u32.

performance of hand-written reshape kernels, and show that future work on optimizations such as vectorization could be useful for Burrito.

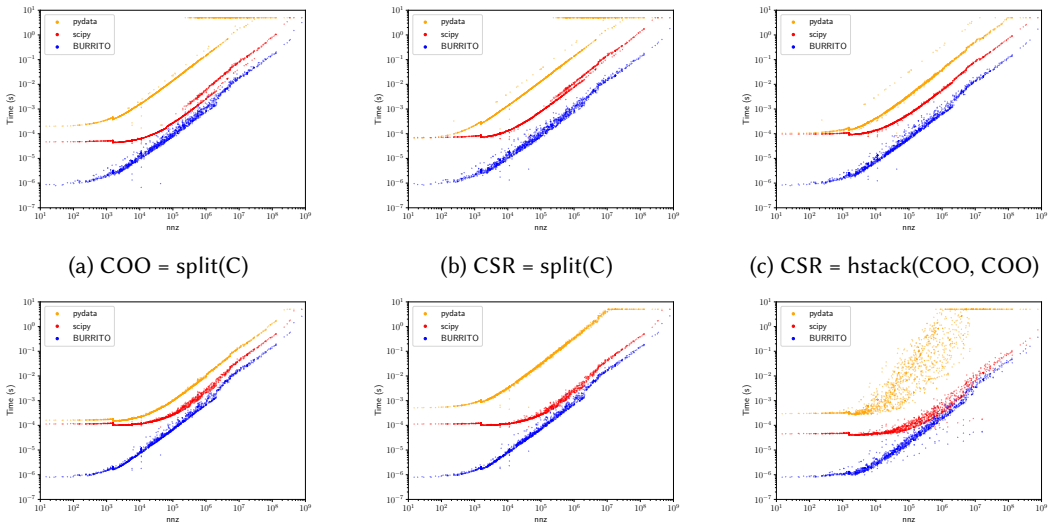
Slicing. Figure 24f shows the performance of slicing the first dimension of a CSR matrix[8]. The geometric mean speed-ups are 4.64× over `scipy.sparse` and 478× over `pydata/sparse`. The `scipy.sparse` library and Burrito generated code perform similarly on large matrices, but `pydata/sparse`'s implementation varies largely and performs significantly worse due to the code being written to handle slicing any dimension of an array (the code is not specialized for a 2D array like `scipy.sparse` and Burrito-generated code). This highlights a trade-off between generality and performance in this system. Note that `pydata/sparse` also timed out on many test matrices, and crashed on many as well for this operation.

8.4 Comparison to Reduced Kernels

To demonstrate the importance of allowing code specialization for particular input and output data structures, we evaluate shape operators with data structures that SotA libraries perform data structure conversions to compute. We compare Burrito-generated code against implementations that perform a data structure conversion on the input or output array in addition to applying the hand-written shape operator, and show that code that does not need to perform these conversions can offer significant performance improvements, and use less memory than the library approaches, as Burrito-generated code does not need to allocate the intermediate tensors that the SotA libraries allocate. This encourages our compiler-based approach.

Reshape. Figures 25a and 25b show the performance of splitting a 1D sparse vector into a 2D sparse matrix of different types. The geometric mean speed-ups across matrices are 7.29× over

[8]The slice takes alternating elements from the first half of the array. We refer the reader to Henry et al. [18, Section 8.2.4] for an in-depth evaluation of non-fused slicing operations on sparse arrays.



(a) COO = split(C) (b) CSR = split(C) (c) CSR = hstack(COO, COO)

(d) COO = vstack(CSR, CSR) (e) COO = hstack(CSR, CSR) (f) COO = slice(CSR)

Fig. 25. Runtimes of shape operators for kernels that the SotA libraries implement via data-structure conversions and reductions to hand-written kernels. Performance shifts around 10^6 non-zero values correspond to each system shifting to using u64 for indices instead of u32.

`scipy.sparse` and 246× over `pydata/sparse`, and 13.0× and 176×, respectively. `scipy.sparse` performs a conversion on the compressed vector into a different compression scheme before reshaping into a 2D COO matrix, while `pydata/sparse` performs the conversion after the reshape. Note that this means the second benchmark, 25b, `scipy.sparse` performs *two* conversions, both before and after the shape operator is applied.

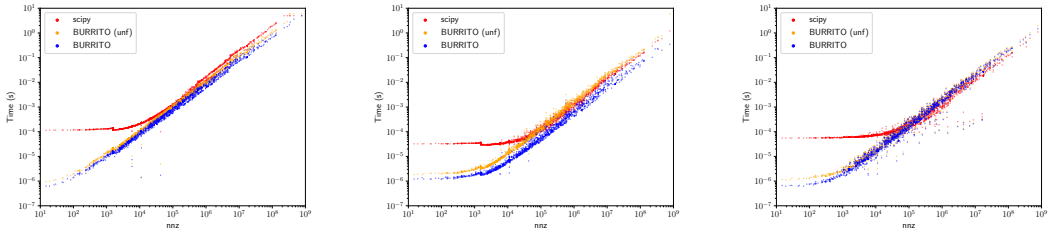
Concatenation. Figures 25c, 25d, and 25e show the performance evaluation of various forms of matrix concatenation where the the libraries perform conversions after the shape operator produces a temporary matrix. They have geometric mean speed-ups of 12.7× over `scipy.sparse` and 52.6× over `pydata/sparse` (Figure 25c), 8.13× and 21.3× (Figure 25d), and 8.33× and 384× (Figure 25e), respectively. These performance gains are largely a result of reduced memory allocations (Burrito-generated code does not need to allocate expensive temporaries like the libraries do).

Slicing. Figure 25f shows the performance of slicing the first dimension of a CSR matrix and inserting into a COO matrix. The geometric mean speed-ups are 8.17× over `scipy.sparse` and 511× over `pydata/sparse`. Note that `pydata/sparse` also timed out on many test matrices, and crashed on many as well for this operation.

8.5 Shape and Compute Operator Fusion

To evaluate the performance benefits of fusing shape and compute operators, we perform a series of comparisons of kernels with and without fusion. For relevance to the state-of-the-art, we also compare to `scipy.sparse`, the faster of the two SotA libraries. We provide a performance evaluation of the speed-up gained by fusing shape and compute kernels on the following benchmarks:

(1) Element-wise multiplication of a flattened CSR matrix with a compressed vector in Figure 26a. The geometric mean (geomean) speed-up across matrices is 1.42× over unfused Burrito, and 3.08× over `scipy.sparse`.



(a) Multiplication of a collapsed CSR matrix with a sparse vector.

(b) SpMV with concatenated CSR matrices on a dense vector.

(c) Multiplication of a sliced CSR matrix with another CSR matrix.

Fig. 26. Runtimes of fused Burrito code versus `scipy` and unfused Burrito code.

(2) Matrix-vector multiplication (SpMV) of two vertically stacked CSR matrices with a dense vector in Figure 26b. The geomean speed-up is 2.23× over unfused Burrito, and 3.86× over `scipy.sparse`.
(3) Element-wise multiplication of a slice of the first dimension of a CSR matrix with another CSR matrix in Figure 26c. The geomean speed-up is 1.22× over unfused Burrito, and 3.24× over `scipy.sparse`.

The benefit of fusing such operators is largely a result of increased temporal locality and removing the allocation of intermediate tensors. This is especially important when the temporaries are large enough to fall out of cache. There are some cases where `scipy.sparse` outperforms the *unfused* Burrito-generated code (e.g. with the fused SpMV, where `scipy.sparse`'s fast `vstack`, discussed in Section 8.3, outperforms Burrito's `vstack`), but *fused* Burrito-generated code still performs best across these benchmarks.

9 Related Works

Taco. Our work builds on the Taco line of work [9, 18, 21, 22, 29]. While our implementation of Burrito subsumes the Taco work on tensor algebra compilation [22] and formats [9], it does not yet directly support a scheduling language to control, e.g., loop tiling or parallelization [21, 29][9] or user-defined functions (UDFs) [18]. Although we believe that our ideas on reshape operations are orthogonal to – but compatible with – scheduling languages, future work is needed to work out how to schedule the highly irregular loops generated from fused shape operators. We additionally believe that the Taco UDFs work [18] fits well into our programming model, as implementing set complements fits cleanly into the iterator model. We leave implementing a feature-complete array programming compiler as future work.

Sparse Shape Operator Compilation. As discussed in the introduction, most prior work in sparse array compilation focuses on compiling compute operators [5, 22, 39], with the exceptions of two compilers: Henry et al. [18] 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 [3] 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 [14] to implement many relational operators, and more recently, Kovach et al. [23] introduced the stream model for iterating over sparse arrays,

[9]We are not aware of any work that combines sparse shape operations with such scheduling operations.

which supports a similar iterator interface based on a model equivalent to *init-valid-next*. Our iterator model builds on Kovach et al. [23], and we introduce additional primitives to support shape operators. Chou et al. [9], Looplets [3], and SparseTIR [38] 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 [6, 7, 11, 13, 15, 26, 34, 40], and a compiler for efficient sparse shape operators must do the same. Kjolstad et al. [21] introduced a simple loop IR and scheduling rewrites that allow for avoiding discordant traversals, and Burrito uses that IR (CIN) and thus can be extended to support the same transformations.

Sparse Array Libraries. There are a number of sparse array libraries [1, 25, 37] that implement some number of shape operators. We compare to the most complete of these, `scipy.sparse` and `pydata/sparse`, in Section 8. 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 [4, 16, 17, 19, 24, 27, 31], 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 [32, 33] 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.

10 Conclusion

We extend sparse iteration theory to handle shape operators in addition to compute operators, and describe the first compiler for a sparse array programming language with multiple shape operators in addition to 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.

Data-Availability Statement

Performance results were generated with a publicly available artifact [28] containing all benchmarking code and scripts, as well as instructions for reproducibility. The Burrito compiler is also available here. Benchmarking results may very based on the hardware used.

Acknowledgments

We thank our reviewers for their valuable feedback. We also thank Rohan Yadav for helpful discussions and repeated feedback throughout the writing process. We would also like to thank Alex Ozdemir, Amanda Liu, Andrew Adams, Devanshu Ladsaria, Evan Laufer, Gina Sohn, James Dong, Katherine Mohr, Maaz Bin Safeer Ahmad, Manya Bansal, Matthew Sotoudeh, Olivia Hsu, Rubens Lacouture, Scott Kovach, Shiv Sundram, and Shoaib Kamil, for their helpful feedback on drafts of this paper. This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was in part supported by the National Science Foundation under Grant CCF-2143061. Alexander J Root was supported by the NSF Graduate Research Fellowship.

A Sequence Simplification

Figure 27 provides the rewrite rules used to simplify sequences during iteration lattice construction in Section 6.2.2 and Figure 12. These rewrites are iteratively applied to simplify a sequence expression bottom-up after a particular index sequence (e.g. from a tensor) has been replaced with an empty sequence, $\varnothing$, for sparse dimensions, or a full sequence, $\mathcal{F}$, for dense dimensions.

As an example, consider the sequence expression $(a \times b) \cup c$. The edge sequences (Section 6.2.1) are a and c. Let us first consider removing c: if c is sparse, then we apply the union rewrite rule to simplify the expression:

$$(a \times b) \cup c \to (a \times b) \cup \varnothing \to (a \times b)$$

However, if c is dense, then we apply a different rule: the union of a *full* sequence (dense-and-empty) with any other sequence is *full*. This is because a dense sequence can only become empty if the entire dimension has been iterated over, so the union itself must have been fully iterated. We denote a *full* sequence as $\mathcal{F}$. The series of rewrites for this case are then:

$$(a \times b) \cup c \to (a \times b) \cup \mathcal{F} \to \mathcal{F}$$

Now, consider removing a. If a is sparse, we perform the following rewrites:

$$(a \times b) \cup c \to (\varnothing \times b) \cup c \to \varnothing \cup c \to c$$

This is a consequence of a sparse a: the iteration space $a \times b$ is not fully iterated, so c may not be empty. The same is true if a is dense but b is sparse. However, if a is dense *and* b is dense, then removing a means fully iterating over the space $a \times b$ (the same space iterated over by c). Thus, the following rewrites are performed because that space is fully iterated over:

$$(a \times b) \cup c \to (\mathcal{F} \times b) \cup c \to \mathcal{F} \cup c \to \mathcal{F}$$

Our treatment of full (dense-and-empty) and empty (sparse-and-empty) allows our iteration lattice algorithm to avoid generating lattice points for states that are impossible to reach, just as Taco [22]'s iteration lattice construction algorithm does via its filter step.

Union

$$\frac{a : \mathcal{F} \lor b : \mathcal{F}}{a \cup b : \mathcal{F}} \qquad \frac{a : \varnothing \land b : \mathcal{X}}{a \cup b : \mathcal{X}} \qquad \frac{a : \mathcal{X} \land b : \varnothing}{a \cup b : \mathcal{X}} \qquad \frac{a : \mathcal{X} \land b : \mathcal{Y}}{a \cup b : \mathcal{X} \cup \mathcal{Y}}$$

Intersection

$$\frac{a : \mathcal{F} \lor b : \mathcal{F}}{a \cap b : \mathcal{F}} \qquad \frac{a : \varnothing \lor b : \varnothing}{a \cap b : \varnothing} \qquad \frac{a : \mathcal{X} \land b : \mathcal{Y}}{a \cap b : \mathcal{X} \cap \mathcal{Y}}$$

Product

$$\frac{a : \mathcal{F} \land b \text{ is dense}}{a \times b : \mathcal{F}} \qquad \frac{a : \mathcal{F} \lor a : \varnothing}{a \times b : \varnothing} \qquad \frac{a : \mathcal{X} \land b : \mathcal{Y}}{a \times b : \mathcal{X} \times \mathcal{Y}}$$

Concatenation

$$\frac{a : \mathcal{F} \lor b : \mathcal{F}}{a \sqcup b : \mathcal{F}} \qquad \frac{(a : \mathcal{F} \lor a : \varnothing) \land b : \mathcal{X}}{a \sqcup b : |a| + \mathcal{X}} \qquad \frac{(b : \mathcal{F} \lor b : \varnothing) \land a : \mathcal{X}}{a \sqcup b : \mathcal{X} + |b|} \qquad \frac{a : \mathcal{X} \land b : \mathcal{Y}}{a \sqcup b : \mathcal{X} \sqcup \mathcal{Y}}$$

Projection

$$\frac{a : \mathcal{F}}{\pi_k(a, J) : \mathcal{F}} \qquad \frac{a : \varnothing}{\pi_k(a, J) : \varnothing} \qquad \frac{a : \mathcal{X}}{\pi_k(a, J) : \pi_k(\mathcal{X}, J)}$$

Slicing

$$\frac{a : \mathcal{F}}{a[s{:}e{:}r] : \mathcal{F}} \qquad \frac{a : \varnothing}{a[s{:}e{:}r] : \varnothing} \qquad \frac{a : \mathcal{X}}{a[s{:}e{:}r] : \mathcal{X}[s{:}e{:}r]}$$

Fig. 27. Sequence simplification rules used for iteration lattice construction in Figure 12. $\varnothing$ represents a sparse-and-empty set, and $\mathcal{F}$ represents a dense-and-empty set. $\mathcal{X}$ and $\mathcal{Y}$ are used to denote non-empty sets. As in Section 4.2, the notation $|a|$ represents the size of the dimension a iterates over, *not* the size of the set a. The notation $x + |a|$ and $|a| + x$ represent right padding and left padding the set x by $|a|$, respectively.

References

[1] 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.

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

[3] 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

[4] 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.

[5] 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

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

[7] 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.

[8] 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.

[9] 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

[10] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

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

[12] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

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

[14] 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

[15] 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.

[16] 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

[17] 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

[18] 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

[19] 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

[20] Wenzel Jakob. 2022. nanobind: tiny and efficient C++/Python bindings. https://github.com/wjakob/nanobind.

[21] 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.

[22] 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

[23] 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

[24] 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

[25] 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.

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

[27] 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.* Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[28] Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart Bik, and Fredrik Kjolstad. 2024. *Artifact for OOPSLA 2024 Paper: Compilation of Shape Operators on Sparse Arrays.* https://doi.org/10.5281/zenodo.13381305

[29] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proceedings of the ACM on Programming Languages* 4 (November 2020). Issue OOPSLA.

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

[31] 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

[32] 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

[33] 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

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

[35] W.F. Tinney and J.W. Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809. https://doi.org/10.1109/PROC.1967.6011

[36] 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.

[37] 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

[38] 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

[39] 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

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

Received 2024-04-06; accepted 2024-08-18