Compilation of Sparse Array Programming Models

RAWN HENRY*, Massachusetts Institute of Technology, USA
OLIVIA HSU*, Stanford University, USA
ROHAN YADAV, Stanford University, USA
STEPHEN CHOU, Massachusetts Institute of Technology, USA
KUNLE OLUKOTUN, Stanford University, USA
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA
FREDRIK KJOLSTAD, Stanford University, USA

This paper shows how to compile sparse array programming languages. A sparse array programming language is an array programming language that supports element-wise application, reduction, and broadcasting of arbitrary functions over dense and sparse arrays with any fill value. Such a language has great expressive power and can express sparse and dense linear and tensor algebra, functions over images, exclusion and inclusion filters, and even graph algorithms.

Our compiler strategy generalizes prior work in the literature on sparse tensor algebra compilation to support any function applied to sparse arrays, instead of only addition and multiplication. To achieve this, we generalize the notion of sparse iteration spaces beyond intersections and unions. These iteration spaces are automatically derived by considering how algebraic properties annotated onto functions interact with the fill values of the arrays. We then show how to compile these iteration spaces to efficient code.

When compared with two widely-used Python sparse array packages, our evaluation shows that we generate built-in sparse array library features with a performance of 1.4× to 53.7× when measured against PyData/Sparse for user-defined functions and between 0.98× and 5.53× when measured against SciPy/Sparse for sparse array slicing. Our technique outperforms PyData/Sparse by 6.58× to 70.3×, and (where applicable) performs between 0.96× and 28.9× that of a dense NumPy implementation, on end-to-end sparse array applications. We also implement graph linear algebra kernels in our system with a performance of between 0.56× and 3.50× compared to that of the hand-optimized SuiteSparse:GraphBLAS library.

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

Additional Key Words and Phrases: Sparse Array Programming, Sparse Arrays, Compilation

ACM Reference Format:
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 1 (November 2021), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Both authors contributed equally to the paper

Authors' addresses: Rawn Henry, Massachusetts Institute of Technology, 32 Vassar St, Cambridge, MA, 02139, USA, rawn@mit.edu; Olivia Hsu, Stanford University, 353 Jane Stanford Way, Stanford, CA, 94305, USA, owhsu@stanford.edu; Rohan Yadav, Stanford University, 353 Jane Stanford Way, Stanford, CA, 94305, USA, rohany@cs.stanford.edu; Stephen Chou, Massachusetts Institute of Technology, 32 Vassar St, Cambridge, MA, 02139, USA, s3chou@csail.mit.edu; Kunle Olukotun, Stanford University, 353 Jane Stanford Way, Stanford, CA, 94305, USA, kunle@stanford.edu; Saman Amarasinghe, Massachusetts Institute of Technology, 32 Vassar St, Cambridge, MA, 02139, USA, saman@csail.mit.edu; Fredrik Kjolstad, Stanford University, 353 Jane Stanford Way, Stanford, CA, 94305, USA, kjolstad@stanford.edu.

2475-1421/2021/11-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Arrays are fundamental data structures that let us represent collections of numbers, tabular data, grids embedded in Euclidean space, tensors, and more. They naturally map to linear memory and it is unsurprising that they have been the central data structure in languages built for numerical computation since Fortran [Backus et al. 1957] and APL [Iverson 1962]. In fact, Python became prevalent in computational science, data analytics, and machine learning partially due to the introduction of the NumPy array programming library [Harris et al. 2020].

An array programming model is a programming model whose expressions operate on arrays as a whole through element-wise operations, broadcasts, and reductions over dimensions. From APL [Iverson 1962] introduced in 1960 to NumPy [Harris et al. 2020] today, array programming languages have played a prominent role in our programs. For example, NumPy permits element-wise operations and reductions with any user-defined function, broadcasting, and slicing.

A sparse array is an array where many components have the same value, known as a *fill value*. Sparse arrays are becoming increasingly important as the need for numerical computation across large, sparsely populated systems increases in scientific computing, data analytics, and machine learning. They can be used to model sparse matrices and tensors [Virtanen et al. 2020], sparse grids [Hu et al. 2019], and even graphs [Mattson et al. 2013]. For example, sparse arrays can represent the number of friends shared by every pair of people (the sparsity arises because most people share no friends), the set of nodes to exclude in each step of breadth-first search (Section 8.3), or black-and-white MRI images (Section 8.4.1).

Therefore, there is a need for a sparse array programming model as a counterpart to—and generalization of—dense array programming models. In fact, at the time of writing, the roadmap [SciPy 2021] of the ubiquitous SciPy library [Virtanen et al. 2020] calls directly for a sparse NumPy as one of five goals. The PyData/Sparse project has responded with an implementation [Abbasi 2018], but it relies on data transformation to implement the significant generality of sparse array programming and therefore runs significantly slower than what is possible.

Table 1. Features in our sparse array programming model compared to those in related programming models.

Paradigm	Supported Functions			Data Representation				Slicing
	(+, ×)	Any semiring	Any	Dense	Sparse		Any # of dims.	
		(∧, ∨), . . .	`foo`, . . .		Zero fill	Any fill		
Dense Array Programming (NumPy)	✔	✔	✔	✔	✘	✘	✔	✔
Dense Tensor Algebra	✔	✘	✘	✔	✘	✘	✔	✔
Sparse Tensor Algebra (TACO)	✔	✘	✘	✔	✔	✘	✔	✘
Sparse Linear Algebra	✔	✘	✘	✔	✔	✘	✘	✘
Sparse LA on Any Semiring (GraphBLAS)	✔	✔	✘	✔	✔	✘	✘	✔
Sparse Array Programming (This Work)	✔	✔	✔	✔	✔	✔	✔	✔

In this paper, we present the first sparse array programming model compiler that can fuse and compile any expression involving sparse and dense arrays with arbitrary (implicit) fill values, where the operators and reductions can be any function. The array expression $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$ is an example of a computation that cannot be expressed in sparse tensor algebra (since it uses operations that are not additions or multiplications) and that cannot be expressed in dense array programming (if the inputs B, C, and D are too large to store without compression). Table 1 and Fig. 1 show how our proposed sparse programming model is a superset of the programming models of NumPy dense array programming, TACO sparse tensor algebra, and the GraphBLAS [Mattson et al. 2013] graph algorithm library. In order to execute arbitrary functions, we generalize the compilation theory of Kjolstad et al. [2017] to support any sparse iteration space. We have also extended the sparse iteration theory to support generating code to compute on sliced windows of data, which allows for operating on subsets of sparse arrays in place. In addition, we built an API for defining these functions and for declaring their properties. Our technical contributions are:

(1) A generalization of sparse iteration space theory to include any sparse iteration space, instead of only those that can be described by intersections and unions.
(2) Code generation to support any sparse iteration space for arbitrary user-defined functions.
(3) Derivation of sparse iteration spaces from functions decorated with mathematical properties.
(4) Extension of sparse arrays to allow any fill value (not just 0) for compressed entries.
(5) Generalization of iteration spaces to allow iteration over sub-array slices of sparse arrays.

We evaluate these contributions by comparing against implementations of sparse array primitives in popular and state-of-the-art sparse array programming libraries like SciPy and PyData/Sparse, as well as in larger applications like image processing and graph processing. Our evaluation shows a normalized speedup of 0.98× to 5.63× compared to SciPy/Sparse for sub-array slicing and between 1.4× and 43.4× compared to PyData/Sparse for universal functions. Furthermore, we demonstrate our technique's ability to fuse computation with a performance improvement of 12.7× to 43.4× for fused universal functions when measured against PyData/Sparse. In the context of graph kernels, our system performs between 0.56× and 3.50× that of a hand-optimized application-specific baseline system, SuiteSparse:GraphBLAS. For practical array algorithms, we outperform PyData/Sparse by between 6.4× to 70.3×, and the relative performance of NumPy compared to our system is between 0.96× to 28.93× when a dense implementation is feasible.

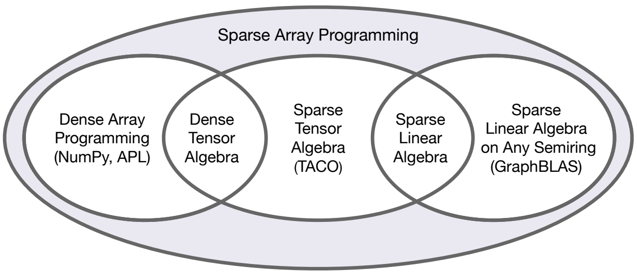
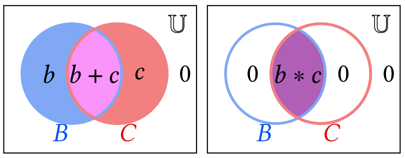

Sparse Array Programming
Dense Array Programming (NumPy, APL)
Dense Tensor Algebra
Sparse Tensor Algebra (TACO)
Sparse Linear Algebra
Sparse Linear Algebra on Any Semiring (GraphBLAS)

Fig. 1. Comparison of programming models.

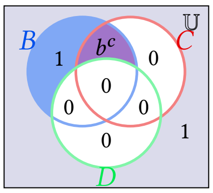
2 MOTIVATION

Array programming is a fundamental computation model that supports a wide variety of features, including array slicing and arbitrary element-wise, reduction, and broadcasting operators. However, current dense array implementations cannot store and process the increasingly large and sparse data emerging from applications like machine learning, graph analytics, and scientific computing. Sparse tensor algebra, on the other hand, is a powerful tool that allows for multilinear computation on tensors—higher-order matrices and vectors. Multi-dimensional arrays can be represented as tensors, which means that sparse tensor algebra allows for computation on sparse arrays, but there are limitations to the existing sparse tensor algebra model.

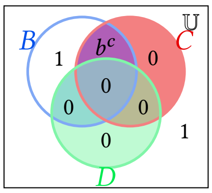
Tensor algebra computation and reductions are only defined across additions and multiplications. Element-wise addition $A = B + C$ takes the union of non-zero input values and element-wise multiplication $A = B * C$ takes the intersection, as illustrated in Fig. 2a. However, there are situations where the user would want to perform more general computation. One example is $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$, which raises B to the power of C (`power`) and filters the result by the logical inverse of D. Arbitrary functions like `power` are not expressible using sparse tensor algebra since they cannot be defined by combining the intersection (multiplication) or union (addition) of non-zero input values, as shown in Fig. 2. Sparse tensor algebra also limits the definition of sparsity to



(a) Add (union) and multiply (intersection) computation space with 0 compression



(b) Masked power with 0 compression of the result A



(c) Masked power with 1 compression of the result A

Fig. 2. Computation spaces of traditional tensor algebra operators (a) versus arbitrary function computation for the masked power example: $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$ with 0-value (b) and 1-value (c) compression of A. Color-filled regions require the computation denoted with black text, and white-filled regions are ignored.

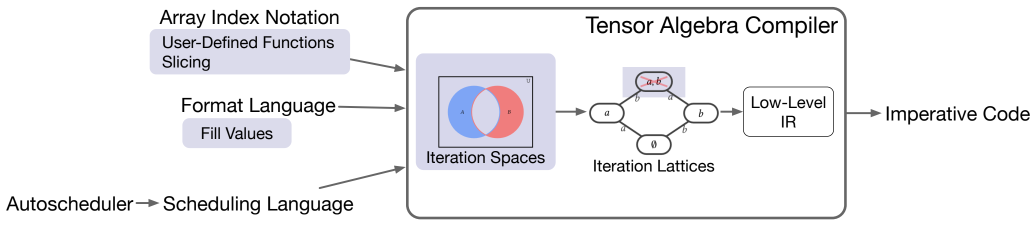


Fig. 3. Overview of the sparse array compiler system. Gray components are new contributions of this work.

having a significant number of zeros that can be compressed away (see Fig. 2b). Our `power` example motivates the need to compress out other values instead—namely 1 since $b^0 * 1 = 1$ (see Fig. 2c). Furthermore, the $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$ example is motivated by applications like medical image processing and graph algorithms, which often perform computations that apply filters and masks (like the $*\neg D_{ij}$ sub-expression). Generalizing tensor algebra to any function requires formalizing the function's properties and computational behavior. Finally, tensor algebra expressions are also restricted to computation on entire tensors, even though it can be useful to extract and compute on sub-arrays. These limitations motivate us to generalize concepts from sparse tensor algebra and dense array programming to propose a sparse array programming model and a compilation-based system that realizes it.

3 OVERVIEW

We implemented the sparse array programming model and sparse array compilation as extensions to the open-source sparse tensor algebra compiler framework TACO [Kjolstad et al. 2017], as depicted in Fig. 3. Our extension is open-source and publicly available at https://github.com/tensor-compiler/taco/tree/array_algebra. Like the TACO compiler, our sparse array compiler takes an algorithm description, a format language [Chou et al. 2018], and a scheduling language [Senanayake et al. 2020].

Unlike the TACO compiler, which compiles a tensor algebra language [Kjolstad et al. 2017], the input algorithm description for our sparse array compiler is a sparse array programming model, further described in Section 4. The programming model supports applying any functions across sparse arrays through a new language we call *array index notation* (see Section 4.2) and compressing out any value from the sparse arrays through an extended format language (see Section 4.1). Array index notation uses sparse tensors to represent sparse arrays and allows the description of any universal function along with its mathematical properties, which is detailed in Section 4.3. Additionally, computations in array index notation can be performed on sparse sub-arrays using sparse array slicing and striding, as also detailed in Section 4.2. The combination of sparse array representations and their fill values, array index notation, sparse array slicing, and user-defined functions forms the sparse array programming model. Figs. 4 and 5 show how programmers can express complex computations using this programming model[1].

Arbitrary user-defined functions are specified by a description of the function's computation and iteration pattern. The iteration pattern describes how the compiler should iterate through values of the input array space, defined directly through a set algebra composed of intersections, unions, and complements of sparse array coordinates. Instead of providing an explicit iteration pattern, users may provide mathematical properties of the function which the sparse array compiler uses, along with fill values of the input tensors, to automatically derive an iteration pattern (see Section 4.3). We describe these generalized iteration spaces and property derivations for generalized functions in Section 5.

The sparse array compiler uses the descriptions of generalized iteration spaces to create an extension of the iteration lattice intermediate representation (IR) described by Kjølstad [2020] to simplify loop and case-statement generation for an input sparse tensor computation. We describe the necessary generalizations to the iteration lattice IR in Section 6 to represent iteration over any iteration space, not just those described by intersection and union expressions. The sparse array compiler uses the generalized iteration lattice to generate low-level code that performs iteration over any iteration space. We describe how to lower an iteration lattice into low-level code as well as how to generate code that operates on slices of tensors in Section 7. Fig. 6 shows an example of optimized code that the sparse array compiler can generate using these techniques.

Finally, in Section 8 we not only evaluate our sparse array compiler against an existing sparse array programming library that provides as much generality as our system, but also against special purpose libraries that hand-code implementations of specific sparse array programs.

4 SPARSE ARRAY PROGRAMMING MODEL

In this section, we describe the features of a general sparse array programming model through a programming language we call *array index notation* that supports complex computations on sparse arrays. Array index notation generalizes the conventional tensor index notation by relaxing the definition of sparse arrays and supporting a wider range of operations on sparse arrays.

4.1 Sparse Arrays and Fill Values

Array index notation operates on multi-dimensional arrays. A multi-dimensional array can be viewed as a map from sets of (integer) coordinates to their corresponding values, which may be of any data type (e.g., floating-point values, integers, etc.).

An array is sparse if many of its components have the same value, which we refer to as the array's *fill value*. For instance, an array that encodes distances between directly-connected points in a road network (with two points having a distance of ∞ if they are not directly connected by a road) is very likely sparse since most pairs of points in the network are not directly connected, meaning most components in the array would be ∞. This distance array can be said to have a fill value of ∞, while all other (i.e., non-infinite) values in the array are its *defined values*.

Sparse arrays can be efficiently stored in memory using various data structures (formats) that omit all (or at least most) of the arrays' fill values. Fig. 7 shows two examples of sparse two-dimensional array (i.e., matrix) formats. The coordinate list (COO) format stores the row/column coordinates

[1]Example code using the PyData/Sparse API can be found in Appendix A.2 in the supplemental materials[2].

```
// Define a dense vector format
// and a sparse vector format
// with fill values of 0.
Format dv({dense}, 0);
Format sv({compressed}, 0);

// Declare inputs to be sparse
// vectors and declare output
// to be a dense vector.
Tensor<int> a(N, dv);
Tensor<int> b(N, sv);
Tensor<int> c(N, sv);

// Define computation that computes
// element-wise GCD of two vectors.
IndexVar i;
a(i) = gcd(b(i), c(i));

// Perform computation by generating
// and executing code in Fig. 6.
std::cout << a << std::endl;
```

Fig. 4. C++ code that uses our sparse array compiler to compute the element-wise greatest common divisor (GCD) of two sparse vectors.

```
def gcd(x,y):
  x,0 => { return abs(x); }
  0,y => { return abs(y); }
  x,y => {
    x = abs(x);
    y = abs(y);
    while (x != 0) {
      int t = x;
      x = y % x;
      y = t;
    }
    return y;
  }
  iteration_space:
    {x ≠ 0} ∪ {y ≠ 0}
```

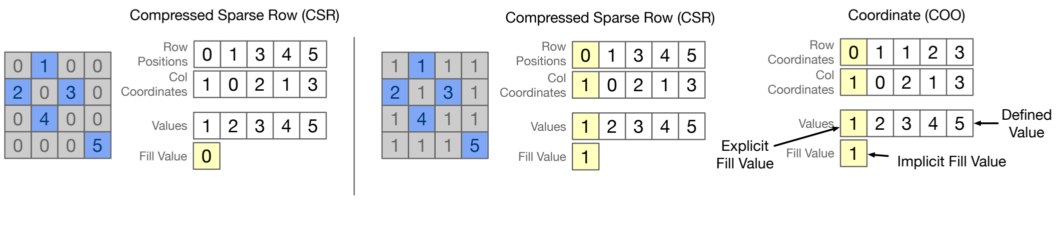
Fig. 5. A function that implements the GCD operation. It contains optimized implementations for the cases where x or y is 0, and the iteration space is explicitly defined using iteration algebra.

```
int pb = b_pos[0];
int pc = c_pos[0];
while (pb < b_pos[1] &&
       pc < c_pos[1]) {
  int ib = b_crd[pb];
  int ic = c_crd[pc];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    int x = b_vals[pb];
    int y = c_vals[pc];
    x = abs(x);
    y = abs(y);
    while (x != 0) {
      int t = x;
      x = y % x;
      y = t;
    }
    a_vals[i] = y;
  } else if (ib == i) {
    int x = b_vals[pb];
    a_vals[i] = abs(x);
  } else {
    int y = c_vals[pc];
    a_vals[i] = abs(y);
  }
  pb += (ib == i);
  pc += (ic == i);
}
while (pb < b_pos[1]) {
  int x = b_vals[pb];
  a_vals[i] = abs(x);
  pb++;
}
while (pc < c_pos[1]) {
  int y = c_vals[pc];
  a_vals[i] = abs(y);
  pc++;
}
```

Fig. 6. Code that our technique generates to compute $a_i = \text{gcd}(b_i, c_i)$, assuming b and c are sparse vectors with zeros compressed out.

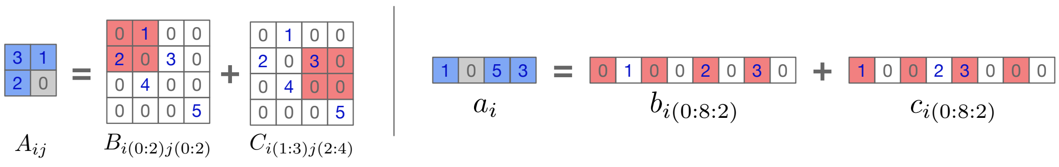
and value of every defined value in the array, while the compressed sparse row (CSR) format additionally compresses out the row coordinates by using a positions array to track which defined values belong to each row. Chou et al. [2018, 2020] showed how a format language can precisely describe a wide range of sparse array formats in a way that lets compilers generate efficient code to compute using the arrays stored in those formats. However, this language assumes that sparse arrays always have a fill value of 0, which, as the distance array example shows, is not always true.

We generalize the data representation language to support arbitrary fill values (such as ∞ and 1) by requiring that the compressed value be specified as part of the sparse array format description. Fig. 7b, for example, shows how both CSR and COO can be specified to have fill values of 1. Array components that are not explicitly stored are called *implicit fill values*, and components that are explicitly stored but also equal the fill value are called *explicit fill values*.



(a) CSR matrix with a fill value of 0 **(b)** CSR and COO matrices with a fill value of 1

Fig. 7. Examples of varying sparse array formats with different fill values.



(a) Windowing example **(b)** Striding example

Fig. 8. Array index notation supports computations on slices of sparse arrays.

4.2 Array Index Notation

As with tensor index notation, computations on multi-dimensional arrays can be expressed in array index notation by specifying how each component of the result array should be computed in terms of components of the input arrays. Element-wise addition of two three-dimensional arrays, for instance, can be expressed as $A_{ijk} = B_{ijk} + C_{ijk}$, which specifies that each component of the result array A should be computed as the sum of its corresponding components in the input arrays B and C. Array index notation can also express computations that reduce over components of operand arrays along one or more dimensions. For example, $y_i = \sum_j A_{ij}$ expresses a computation that defines each component of y to be the sum of all components in the corresponding row of A. The full syntax of array index notation can be found in Appendix A.1 in the supplemental materials[2].

Array index notation extends tensor index notation in two ways. First, array index notation allows programmers to define arbitrary functions (on top of addition and multiplication) and to use these functions in computations. So, for instance, a programmer can define a function `xor` that computes the exclusive or of three scalar inputs. The programmer may then use this function for element-wise computation with three-dimensional arrays, which can be expressed as $A_{ijk} = \text{xor}(B_{ijk}, C_{ijk}, D_{ijk})$. User-defined functions can also be used in reductions. For example, assuming `min` is a binary function that returns the smallest argument as output, the statement $y_i = \min_j A_{ij}$ expresses a computation that returns the minimum value in each row of a two-dimensional array. Section 4.3 describes how to define custom array index notation functions.

Second, array index notation allows users to slice and compute with subsets of sparse arrays. For instance, as Fig. 8a shows, the statement $A_{ij} = B_{i(0:2)j(0:2)} + C_{i(1:3)j(2:4)}$ specifies a computation that extracts 2×2 sub-arrays from B and C and element-wise adds the sub-arrays, producing a 2×2 result array A. Array index notation also supports strided accesses of sparse arrays. For instance, as Fig. 8b shows, the statement $a_i = b_{i(0:8:2)} + c_{i(0:8:2)}$ specifies computation that extracts

[2]A link to the supplemental materials can be found here.

and element-wise adds the components with even-valued coordinates from b and c. (This slicing notation corresponds to the standard Python syntax `x[lo:hi:st]`, which accesses an array `x` from coordinate `lo` to non-inclusive coordinate `hi` with stride `st`.) Slicing operations in array index notation can be viewed semantically as first extracting the sliced array into a new array where each dimension ranges from 0 to the size of the slice, and then using that new array in the rest of the computation. However, just as in dense array programming, slicing operations should be oblivious to the underlying data structures used and should not result in unnecessary data movement or reorganization. Slicing operations should instead adapt the implementation of the array index notation statement to the desired slicing operation and format of the sparse array. Section 7.3 describes our technique to emit efficient code to slice sparse arrays.

4.3 Generalized Functions

Programmers can define custom functions that can be used to express complex sparse array computations in array index notation. Programmers specify the semantics of a custom function by providing an implementation that, given any (fixed) number of scalar inputs, computes a scalar result. Function implementations are written in a C-like intermediate language that provides standard arithmetic and logical operators, mathematical functions found in the C standard library, and imperative constructs such as `if`-statements and loops. Figs. 5 and 9 illustrate how users can specify the semantics of simpler functions like bitwise-and as well as more complex functions like the greatest common divisor (GCD) function, which is implemented using the Euclidean algorithm.

```
def bitwise_and(x,y):
  x,y => {
    return x & y;
  }
  properties:
    commutative
    annihilator=0
```

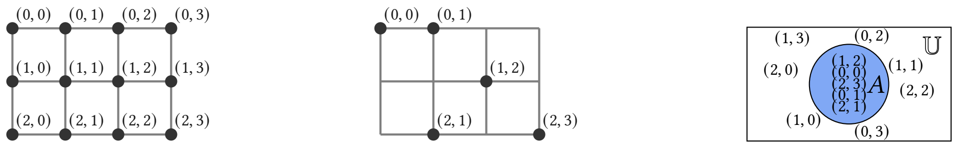
Fig. 9. A function that implements the bitwise-and operation decorated with algebraic properties. If the fill values of `x` and `y` are 0, then the iteration space for this function will be an intersection.

A user may optionally specify, for each combination of fill value and defined value inputs, how the function can be more efficiently computed for that specific combination of inputs. For example, lines 2–3 in Fig. 5 shows how a programmer can specify that, when either argument is zero, the `gcd` function simply has to return the value of the other argument. Using these additional specifications, our technique can generate code like in Fig. 6, which computes the element-wise GCD of two input vectors without having to explicitly invoke the Euclidean algorithm whenever one input is guaranteed to be zero (see lines 19–25 and 29–38).

To support efficient computing on sparse arrays with a custom function, the user must also define the subset of components in the input arrays that could return a value other than the result array's fill value. This can be done explicitly in a language we define called *iteration algebra*, which we describe in Section 5. Fig. 5 shows how a user can define the iteration algebra to specify that the `gcd` function may return a non-zero result only if at least one input is non-zero. Sections 6.2 and 7 explain how our technique can then use this iteration algebra to generate the code in Fig. 6, which computes the element-wise GCD by strictly iterating over the defined values in vectors b and c.

Instead of explicitly specifying iteration algebras for custom functions, users may also annotate functions with any subset of four predefined properties from which our technique can infer optimized iteration algebras:

- **Commutative:** A function is commutative if the order in which arguments are passed to the function does not affect the result. Arithmetic addition is an example of a commutative function, since $x + y = y + x$ for any x and y.



(a) Dense iteration space, with all points present.

(b) Sparse iteration space, with some points missing.

(c) Set interpretation of Fig. 10b.

Fig. 10. A grid representation of iteration spaces showing a dense and sparse iteration space for 4×3 matrix.

- **Idempotent:** A function is idempotent if, for any x, the function evaluates to x whenever all arguments are x (i.e., $f(x, ..., x) = x$). The max function is an example of an idempotent function, since $\max(x, x) = x$ for any x.
- **Annihilator(x[, p]):** A function has an annihilator x if the function evaluates to x whenever any argument is x. Arithmetic multiplication, for instance, has 0 as its annihilator since multiplying 0 by any value yields 0. If p is also specified, then the function is only guaranteed to evaluate to x if the p-th argument (as opposed to any argument) is x.
- **Identity(x[, p]):** A binary function has an identity x if, for any y, the function evaluates to y whenever one argument is x and the other argument is y. Multiplication, for instance, has 1 as its identity since multiplying 1 by any y yields y. If p is also specified, then the function is only guaranteed to evaluate to y if the p-th argument (as opposed to any argument) is x.

Fig. 9 demonstrates how a programmer can specify that the `bitwise_and` function is commutative and has 0 as its annihilator. From these properties, our technique infers that the `bitwise_and` function (with inputs x and y) has iteration algebra $x \cap y$ assuming that the input arrays have 0 as fill values, as we will explain in Section 5.2.

5 GENERALIZED ITERATION SPACES

Having described the desired features of a sparse array programming model, we now explain how our sparse array compiler reasons about and implements these features. In this section, we describe how our system reasons about user-defined functions iterating over any iteration space through an IR called *iteration algebra*. Then, we describe how an iteration algebra can be derived from mathematical properties of user-defined functions.

5.1 Iteration Algebra

We can view the iteration space of loops over dense arrays as a hyper-rectangular grid of points by taking the Cartesian product of the iteration domain of each loop, as in Fig. 10a. A sparse iteration space, shown in Fig. 10b, is a grid with missing points called holes, which take on the *fill value* attached to the format of that array. Another way to view iteration spaces is as a Venn diagram of coordinates where the universe is the set of all points in a dense iteration space. Sparse arrays only define values at some of the possible coordinates in the dense space, forming subsets within the universe, as shown in Fig. 10c. This view naturally leads to a set expression language for describing array iteration spaces, which we introduce, called *iteration algebra*.

Iteration algebra is defined by introducing index variables into set expressions, where the variables in the set expressions are the coordinate sets of sparse arrays. The index variables index into the sparse arrays, controlling which coordinates are compared in the set expression. For example, the iteration algebra for $c_i = \sum_j A_{ij} b_j$ (i.e., sparse matrix-vector multiplication) is $A_{ij} \cap b_j$, where the j in A_{ij} indexes into the second dimension of A and the j in b indexes into the first dimension of b.

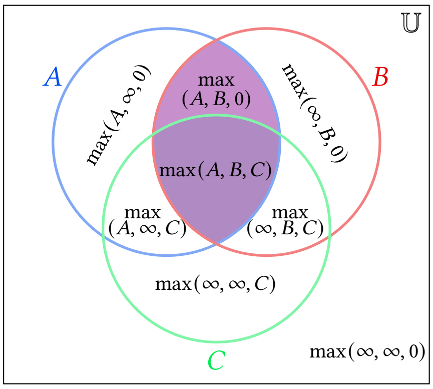


Fig. 11. Illustration of case (1), where f is the ternary max operator, A and B have fill value ∞ and C has fill value 0.

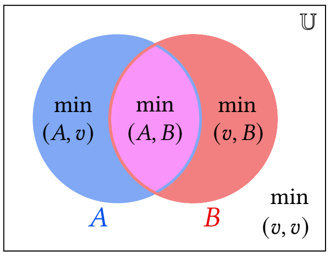


Fig. 12. Illustration of case (2), where f is the idempotent min operator and all arguments have the same fill value v.

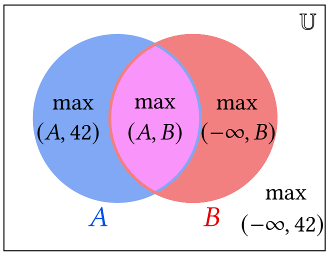


Fig. 13. Illustration of case (3), where f is the max operator with identity $-\infty$, A has fill value 42 and B has fill value $-\infty$.

Coordinate sets indexed by the same index variable are combined using the set operations. In the SpMV example, the j coordinates of A and b are combined with an intersection.

The prior work of Kjolstad et al. [2017] intertwines tensor index notation and the corresponding iteration space by interpreting additions as unions and multiplications as intersections. As such, it is limited to describing and working with spaces that are represented as compositions of those intersections and unions. Our iteration algebra addresses this by adding support for *set complements*, which makes the language complete: any iteration space can be described as compositions of intersections, unions, and complements. For example, set complements can be used to express the iteration space $A_{ij} \cap \overline{B_{ij}}$, which contains only coordinates in A that are also not present in B.

Promoting iteration algebra to an explicit compiler IR has two benefits. First, it lets users directly express the iteration space of a complicated function whose space can not be derived from simple mathematical properties. Second, it detaches the compiler machinery that generates low-level loops to iterate over data structures from the unbounded number of functions that a user may define.

5.2 Deriving Iteration Algebras

To derive the iteration algebra for an array index notation expression, our technique recurses on the expression and derives the algebra for each subexpression by combining the iteration algebras of its arguments. As an example, to derive the iteration algebra for the expression $\text{bitwise\_and}(\text{gcd}(b_i, c_i), d_i)$, our technique first derives the iteration algebra for $\text{gcd}(b_i, c_i)$ and then combines it with d_i (the iteration algebra for the second argument of `bitwise_and`).

If a function f is explicitly defined with an iteration algebra *alg*, then our technique derives the iteration algebra for an invocation of f by replacing the terms of *alg* with the iteration algebras of the function arguments. In Fig. 5, for instance, `gcd(x,y)` is defined with iteration algebra

$\{x \neq 0\} \cup \{y \neq 0\}$. So to derive the iteration algebra for $\gcd(b_i, c_i)$, our technique checks that b and c have 0 as fill values and, if so, substitutes b_i for $\{x \neq 0\}$ and c_i for $\{y \neq 0\}$, yielding $b_i \cup c_i$ as the function call's iteration algebra. (If either b or c has a fill value other than 0 though, our technique instead conservatively returns the universe $\mathbb{U}$ as the function call's iteration algebra.)

If a function is instead annotated with properties, our technique attempts to construct an iteration algebra that minimizes the amount of data to iterate over. This is done by pattern matching on the cases below, in the order they are presented. In particular, assuming a function f is invoked with arguments *args* in the target expression, we apply the cases below. For each case, we include an example of resulting iteration space on sample inputs, and visual examples for the first three cases in Figures 11, 12 and 13.

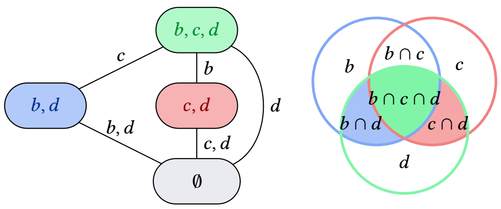
(1) **f has an annihilator α.** When f is commutative, our technique returns the algebra $\mathbb{U}$ intersected with the algebras of all arguments in *args* with fill value of α. Any coordinate c where tensor arguments with fill value α are undefined will cause f to equal α at c because α annihilates f. Therefore, we can iterate only over positions where arguments with fill value α are defined.
Example. Consider the ternary max operator $\max(A, B, C)$, where A and B have fill value ∞ (the annihilator for max, so $\alpha = \infty$) and C has fill value 0. In this case, we emit an algebra to iterate over $A \cap B$. Consider a coordinate c in C. If $c \in A \cap B$, then the max operator will return the maximum of A, B, and C. If $c \in A \cap B$, then no matter what C's value at c is, it will be annihilated by A or B having the value of ∞ (see Fig. 11).

(2) **f is idempotent and all arguments have the same fill value v.** Our technique returns the union of the algebras of all arguments. Since all arguments have fill value v and f is idempotent, f applied at all points outside the union of all arguments evaluates to v.
Example. Consider the min operator $\min(A, B)$, where A and B have some arbitrary fill value v. Because min is idempotent, tt is correct to iterate over the union of A and B —at all coordinates $c \notin A \cup B$, the result of min is $\min(v, v) = v$ (see Fig. 12).

(3) **f has an identity i.** If all arguments have fill value i, then our technique returns the union of the algebras of all arguments, because computation only must occur where the arguments are defined. If all but one argument have fill value i, then our technique can also return the same algebra, but marks that the resulting expression has the fill value v of the remaining argument, since f applied to i and v returns v.
Example. Consider the max operator $\max(A, B)$ where A has fill value $-\infty$ and B has fill value 42. Here, we can infer the result tensor should have fill value 42 since the computation at any coordinate outside of $A \cup B$ is $\max(-\infty, 42) = 42$ (see Fig. 13).

(4) **f is not commutative.** When f is not commutative, cases (1) and (3) can be applied, but only to the position p where the property holds.
Example. Let $f(a, b) = a/b$ has an annihilator 0 at position 0, so case (1) could be applied to iterate only over the defined values of the input array a if it had fill value 0.

If none of these cases match but the result array's fill value is left unspecified by the user, our technique can still return the union of the algebras of all arguments (and constant propagate through f to determine an appropriate fill value for the result). Otherwise, our technique falls back to returning $\mathbb{U}$ as the function call's iteration algebra. In the case of a function call `bitwise_and(x,y)` though, our technique can simply apply the first rule (since Fig. 9 specifies the function is commutative and has 0 as its annihilator) to derive the iteration algebra $x \cap y$ for the function call. Thus, our technique can infer that the expression bitwise_and$(\gcd(b_i, c_i), d_i)$ has $(b_i \cup c_i) \cap d_i$ as its iteration space.



```
while $b$, $c$, and $d$ have coordinates left do
    if in region $[b, c, d]$ then ...
    else if in region $[b, d]$ then ...
    else if in region $[c, d]$ then ...
while $b, d$ has coordinates left do
    if in region $[b, d]$ then ...
while $c, d$ has coordinates left do
    if in region $[c, d]$ then ...
```

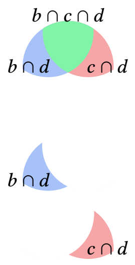


Fig. 14. An iteration lattice for the tensor algebra $(b_i + c_i) * d_i$ and sparse array bitwise_and(gcd$(b_i, c_i), d_i)$ expressions, which both have the iteration space $(b \cup c) \cap d$ for index variable i, along with the sequential pseudocode that gets emitted. The lattice points are colored to match the corresponding Venn diagram. The subsections of the Venn diagram on the right-hand side of the figure correspond to the `while`-loop conditions and `if`-conditions in the code.

6 GENERALIZED ITERATION LATTICES

After constructing an iteration algebra from an array index notation expression as described in Section 5, our compiler translates the algebra into an IR to represent how tensors must be iterated over to realize an iteration space corresponding to the iteration algebra. In particular, we generalize iteration lattices and their construction method described by Kjølstad [2020] to support iteration algebras containing set complements. We first present an overview of iteration lattices, and then detail how they must be extended in order to describe any arbitrary iteration space.

6.1 Background

An iteration lattice divides an iteration space into regions, which are described by the tensors that intersect for each region. These regions are the powerset of the tensors that form the iteration space. Thus, an iteration space with k tensors divides into 2^k *iteration regions* (the last region is the empty set $\emptyset$ where no sets intersect). An iteration lattice is a partial ordering of the the powerset of a set of tensors by size of each subset. Each subset in the powerset is referred to as a *lattice point*. Ordering the tensors in this way forms a lattice with increasingly fewer tensors to consider for iteration, as shown in Fig. 14 for the tensor algebra expression $(b_i + c_i) * d_i$ and for the sparse array expression bitwise_and(gcd$(b_i, c_i), d_i)$. An iteration lattice can also be visualized as a Venn diagram, where points in the lattice correspond to subspace regions, also shown in Fig. 14. We say a lattice point p_1 dominates another point p_2 (i.e., $p_1 > p_2$) if p_1 contains all tensors of p_2 as a subset.

An iteration lattice can be used to generate code that coiterates over any iteration space made up of unions and intersections. The lattice coiterates over several regions until a segment (i.e., tensor) runs out of values. It then proceeds to coiterate over the subset of regions that do not have the exhausted segment. The lattice points enumerate the regions that must be considered at a particular point in coiteration, and enumerate the regions that must be successively excluded until all segments have run out of values. In order to iterate over an iteration lattice, we proceed in the following manner beginning at the top point of the lattice, also referred to as the lattice root point:

(1) Coiterate over the current lattice point's tensors until any of them runs out of values.
(2) Compute the *candidate coordinate*, which at each step is the smallest of the current coordinates of the tensors (assuming coordinates are stored in sorted order within each tensor).

(3) Check what tensors are currently at that coordinate to determine which region the candidate coordinate is in. The only regions we need to consider are those one level below the current lattice point since these points exclude the tensor segments that have run out of values.
(4) Follow the lattice edge for the tensors that have run out of values to a new lattice point, and repeat the process until reaching the bottom.

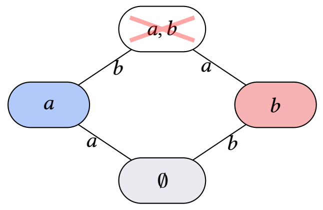
This strategy leads to successively fewer segments to coiterate and regions to consider, which generates code consisting of a sequence of coiterating `while`-loops that become simpler as it moves down the lattice.

6.2 Representing Set Complements

To support iterating over any iteration space—composed from intersections, unions, *and complements*—we introduce the concept of an *omitter point* to iteration lattices. An omitter point is a lattice point where computation must not occur, in contrast to the original lattice points where computation must be performed.

To distinguish omitter points from the original lattice points, we rename the original points to *producer points* since they produce a computation. Omitter points with no producers as children are equivalent to points missing from the lattice, since no loops and conditions need to be emitted for both cases. By contrast, omitter points that dominate producer points must be kept, since these omitter points lead to code that explicitly skips computation in a region.

Fig. 15 illustrates the iteration space (upper right) for a function like a logical `xor` with a symmetric difference iteration algebra, along with the corresponding iteration lattice (left) which contains an omitter point (marked with a red $\times$) at a, b. The pseudocode and the partial iteration spaces show how the coiteration algorithm successively eliminates regions from the iteration space, as the vectors a and b runs out of values. This iteration space is not supported by prior work and illustrates the expressive power of omitter points. An omitter point is needed so the sparse array compiler knows to generate code that coiterates over the vectors a and b while explicitly avoiding computing and storing values when both vectors are defined.



```
while $a$ and $b$ have coordinates left do
    if in region $[a, b]$ then do nothing
    else if in region $[a]$ then . . .
    else if in region $[b]$ then . . .
while $a$ has coordinates left do
    if in region $[a]$ then . . .
while $b$ has coordinates left do
    if in region $[b]$ then . . .
```

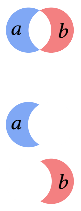


Fig. 15. Iteration lattice and corresponding coiteration pseudocode for `xor` that has the iteration algebra $(a \cup b) \cap \neg(a \cap b)$. The treatment of the omitter point is the same when emitting `while`-loops. When emitting inner-loop `if`-statements, we do nothing at $a \cap b$. Without the explicit skip, we may accidentally end up performing computations inside the $a \cap b$ region. The check for inclusion in $a \cap b$ includes checks that a and b are not explicit fill values at the current point.

6.3 Construction

We generate lattices from an iteration algebra using a recursive traversal of the iteration algebra shown in Algorithm 1. Our algorithm first performs two preprocessing passes over the input iteration algebra A. The first pass uses De Morgan's Laws to push complement operations down the input algebra until complements are applied only to individual tensors (i.e. $\overline{B \cap C} \Rightarrow \overline{B} \cup \overline{C}$).

Algorithm 1 Iteration Lattice Construction Algorithm

```
// Let $\mathcal{L}$ represent an iteration lattice and $p$ represent an iteration lattice point.
procedure BuildLattice (Algebra A)
    if A is Tensor(t) then                                              ▷ Segment Rule
        return $\mathcal{L}(p([t], \text{producer=true}))$
    else if A is $\overline{\text{Tensor(t)}}$ then                     ▷ Complement Rule
        $p_o = p([t, \mathbb{U}], \text{producer=false})$
        $p_p = p([\mathbb{U}], \text{producer=true})$
        return $\mathcal{L}([p_o, p_p])$
    else if A is (left ∩ right) then                                    ▷ Intersection Rule
        $\mathcal{L}_l, \mathcal{L}_r$ = BuildLattice(left), BuildLattice(right)
        cp = $\mathcal{L}_l$.points() × $\mathcal{L}_r$.points()
        mergedPoints = $[p(p_l + p_r, \text{producer=}p_l\text{.producer} \wedge p_l\text{.producer}) : \forall (p_l, p_r) \in \text{cp}]$
        mergedPoints = RemoveDuplicates(mergedPoints, ommitterPrecedence)
        return $\mathcal{L}$(mergedPoints)
    else if A is (left ∪ right) then                                    ▷ Union Rule
        $\mathcal{L}_l, \mathcal{L}_r$ = BuildLattice(left), BuildLattice(right)
        cp = $\mathcal{L}_l$.points() × $\mathcal{L}_r$.points()
        mergedPoints = $[p(p_l + p_r, \text{producer=}p_l\text{.producer} \vee p_l\text{.producer}) : \forall (p_l, p_r) \in \text{cp}]$
        mergedPoints = mergedPoints + $\mathcal{L}_l$.points() + $\mathcal{L}_r$.points()
        mergedPoints = RemoveDuplicates(mergedPoints, producerPrecedence)
        return $\mathcal{L}$(mergedPoints)
end procedure
```

The second pass (called *augmentation*) reintroduces tensors present in function arguments but not present in the input iteration algebra, without changing its meaning. For example, consider the function $f(a, b) = a/b$ which has an annihilator of 0 at a. The algebra derivation procedure in Section 5.2 tells us that the iteration algebra for f is a (assuming a has fill value 0)—note that b is not included in the algebra even though it is an argument to f. The augmentation pass uses the set identity $A \cup (B \cap \overline{B})$ to reintroduce any tensor B into the algebra. All tensors present in function arguments but not present in the iteration algebra are brought back into the algebra in this step.

After preprocessing, our algorithm performs a recursive tree traversal matching on each set operator (complement, union, intersection) in the iteration algebra. Unlike lattice construction in Kjolstad et al. [2017], we introduce the Complement Rule and the handling of omitter points in the Intersection and Union Rules. At a high level, our algorithm performs the following operations at each set operator in the algebra:

- **Segment Rule.** Return a lattice with a producer point containing the input tensor.
- **Complement Rule.** Return a lattice that omits computation at the input tensor and performs computation everywhere else.
- **Intersection Rule.** Return a lattice representing the intersection of the two input lattices.
- **Union Rule.** Return a lattice representing the union of the two input lattices.

In the Intersection and Union Rules, taking the cross product of points in the left and right lattices may create duplicate points with different types. These duplicates are resolved with producer precedence in the Union Rule, and omitter precedence in the Intersection Rule. Finally, we prune any omitter points that dominate no producer points since they are equivalent to points missing from the lattice. Fig. 16 visualizes our algorithm applied to the iteration algebra $a \cap \overline{b}$. We first apply the Segment Rule to a and the Complement Rule to $\overline{b}$, and then apply the Intersection Rule on the resulting lattices. A similar example of the Union Rule can be found in Fig. 17.

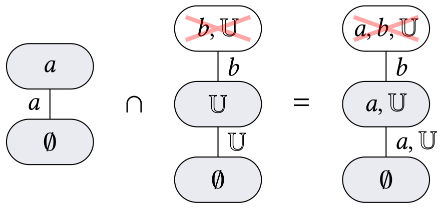


Fig. 16. Intersection Rule for $a \cap \overline{b}$. The a and $\overline{b}$ lattices were generated using the Segment and Complement Rules respectively.

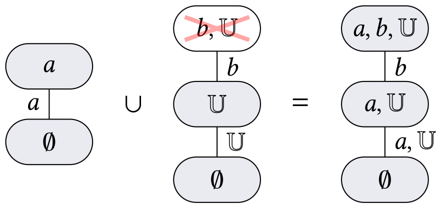


Fig. 17. Union Rule for $a \cup \overline{b}$. The a and $\overline{b}$ lattices were generated using the Segment and Complement Rules respectively.

The presentation of Algorithm 1 is limited to the case when tensors are not repeated in the iteration algebra expression. This is because the lattice point pairs, when being merged, are unaware of whether or not the duplicated tensor fell out from the iteration space in the other lattice point. If the tensor did fall out from the lattice for one point and is in the lattice point for the other, then we end up getting that the two points represent non-overlapping iteration spaces and should not be merged, as illustrated by Fig. 18 between the left a tensor and right b tensor for point pair (p_l, p_r). We solve this by modifying the Cartesian product of points in the algorithm to a filtered Cartesian product, which is fully described in Appendix Algorithm 2 in the supplemental materials[2]. Briefly, the filtered Cartesian product ignores any point pairs from the Cartesian product between $p_l \in \mathcal{L}_l$ and $p_r \in \mathcal{L}_r$ that do not overlap. It determines this by checking for every tensor t in point p_l, whether t exists in p_r's root point but does not exist in point p_r itself (and vice versa).

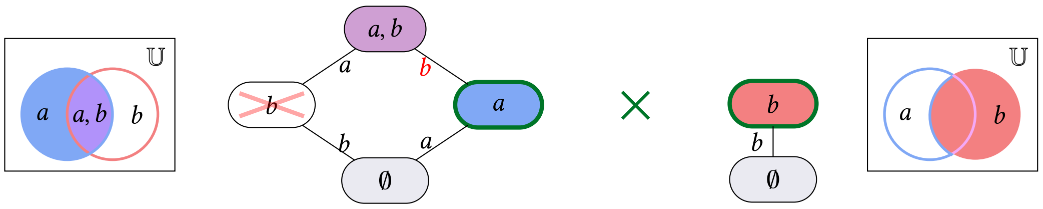


Fig. 18. Iteration lattice and space of two lattices with a repeat tensor b. The $(\mathcal{L}_l \times \mathcal{L}_r)$ produces a point pair: (left point a, right point b) shown in green. When merging that pair, the two Venn diagrams show that the left a-only region (blue) and right whole-b region (red) do not overlap.

7 GENERALIZED CODE GENERATION

In this section, we describe how the generalized iteration lattices described in Section 6 can be used to generate code that iterates over generalized iteration spaces. We also describe optimizations that can be performed during code generation using different properties of user-defined functions, and we describe how code generation is performed for expressions that slice sparse tensors.

7.1 Lowering Generalized Iteration Lattices

Our technique for code generation draws on the code generation technique that TACO uses, as described in [Kjølstad 2020]. The main difference is how iteration lattices are lowered into code, since there are now types associated with each lattice point.

Like TACO, our technique first lowers an array index notation expression into *concrete index notation*, which explicitly denotes the forall loops over index variables. For example, the array index notation expression $\text{xor}(A_{ij}, B_{ij})$ corresponds to the concrete index notation expression

$\forall_i \forall_j\ \mathrm{xor}(A_{ij}, B_{ij})$. The code generation algorithm walks through the forall statements in the concrete index notation expression. At each $\forall_i\ S$ statement, our technique constructs an iteration lattice $\mathcal{L}$ from S and i. Then, at each lattice point $p \in \mathcal{L}$, our technique generates a loop that coiterates over all tensors in p. Next, for each point p' such that $p' \leq p$, our technique emits an `if`-statement whether to enter that case and recursively invokes the code generation procedure on a statement S' formed by removing tensors from S not in p'. In our technique, when p' is a producer point, a compute statement is emitted since producer points correspond to regular points in standard iteration lattices. This entails inlining user-defined function implementations within the case; if the function has multiple implementations (like in Fig. 5), our technique chooses an optimized implementation based on what tensors are present in p'.

When p' is an omitter point, it must be handled differently, as p' represents computation that must not occur. When considering the statement S' constructed from p', our technique emits nothing in order to skip computation at S', as long as S' has no foralls (i.e., it can access tensor values directly). To see why computation cannot always be skipped at omitter points, again consider the expression $\forall_i \forall_j\ \mathrm{xor}(A_{ij}, B_{ij})$. As discussed previously, the iteration lattice for `xor` has an omitter point at A, B. When lowering the loop over i, omitting computation at the point where A and B have equal i coordinates would be incorrect, since computation must be omitted at coordinates that have equal values for both i and j. Finally, when omitting computation, our technique also emits code to check that the tensors do not have explicit fill values at the considered coordinates.

7.2 Reduction Optimizations

When generating code for reductions, our technique can take advantage of properties of the reduction function to emit code that avoids iterating over entire dimensions or that breaks out of reduction loops early. In particular, our technique can perform the following optimizations based on the identity and annihilator properties of the reduction function f:

- **Identity i.** If the (inferred) fill value of the tensor expression being reduced over is equal to i, then we can iterate over only the defined values of the tensor mode, rather than the entire dimension. This optimization corresponds to the standard optimization used by TACO when reducing over addition in the addition-multiplication $(+, \times)$ semiring.
- **Annihilator α.** If target reduction is being performed into a scalar value, then we can insert code to break out of the reduction if the reduction value ever equals α. The loop ordering is important to apply this optimization. Consider the array index notation expression $B_{ij} = \mathrm{reduction}_k(A_{ijk})$. If the loops are ordered as $i \rightarrow j \rightarrow k$ then this optimization could be applied, because for each i and j, k is reduced into B_{ij}. If the loops were instead ordered $i \rightarrow k \rightarrow j$ then this optimization could not be performed, since attempting to break out of the reduction could skip unrelated iterations of the j loop.

7.3 Slicing

This section describes how slicing operations like windowing and striding can be compiled into an expression from array index notation. The intuition for our approach comes from examining slicing operations in dense array programming libraries like NumPy. In NumPy, taking a slice of a dense array is a constant time operation, where an alias to the underlying dense array is recorded, along with a new start and end location. Operations on the sliced array use those recorded bounds when iterating over the array, and offset the coordinates by the bounds of the slice. Rather than viewing a slice as an extraction of a component, we can view it as an iteration space transformation that restricts the iteration space to within that slice, then projects the iteration space down to a *canonical iteration space*, where each dimension ranges from zero to the size of the slice.

Using this intuition, we can view operating on a slice of a tensor dimension, or tensor mode, as restricting the iteration space over that mode to some set S which contains all coordinates in the desired slice. This corresponds to intersecting the iteration lattice for the sliced modes with S. However, when S is a set that has a restricted shape (like for a rectangular slice), the intersection with S can be compiled directly into the tensor expression. This specialization is directed by capabilities of the data structured storing the sliced tensor mode, which provide information about what operations the mode supports [Chou et al. 2018]. We describe how to specialize slicing operations for dense tensor modes that support the capability to efficiently locate (i.e., random access) into arbitrary positions, and for compressed modes that support the ability to iterate over defined elements of the tensor. We include generated code for the array index notation expression $B_{ij} = A_{i(1:5:2)j(2:6:2)}$ in Figure 19 to visualize the effect of slicing on the kernel. In the example, A is a CSR format two-dimensional array and B is a dense two-dimensional array.

```
// Limit outer loop to the slice 1:5:2.
for (int i = 0; i < 2; i++) {
  // Project access to A into the slice.
  int iA = (i * 2) + 1;
  int jA_s = A2_pos[iA];
  int jA_e = A2_pos[(iA + 1)];
  // Seek the start of the slice 2:6:2.
  jA_s = bSearch(A2_crd, jA_s, jA_e, 2);
  // Iterate from the start of the slice.
  for (int jA = jA_s; jA < jA_e; jA++) {
    // Check that coordinate is aligned
    // to the stride 2:6:2.
    if ((A2_crd[jA] - 2) % 2 != 0)
      continue;
    // Project coordinate into canonical
    // iteration space of 2:6:2.
    int j = (A2_crd[jA] - 2) / 2;
    // Break if coord is outside slice.
    if (j >= 2)
      break;
    int jB = i * 2 + j;
    B_vals[jB] = A_vals[jA]; }}
```

Fig. 19. Generated kernel for $B_{ij} = A_{i(1:5:2)j(2:6:2)}$ demonstrating array slicing. Red indicates slicing-related code.

For modes that support efficient locate, our technique supports slicing in a way that is similar to how dense array programming libraries slice arrays. In particular, our technique emits code that operates entirely on the canonical iteration space, and projects accesses to the tensor into the slice's iteration space. For a slice `lo:hi:st`, dense `for`-loops over the sliced mode range from 0 to `(hi - lo) / st` instead of 0 to `dim`. Then, whenever a value `i` is used to access the sliced tensor mode, it is projected from the canonical iteration space into the slice by replacing `i` with `(i * st) + lo`.

Slicing modes that only support efficient iteration is the inverse of how slicing is performed for modes with efficient locate. Since it is not possible to efficiently access only the positions within the slice, our technique generates code that instead iterates over the coordinates in the mode and project these coordinates into the canonical iteration space. When iterating over a tensor mode with a slice `lo:hi:st`, the generated code must restrict the iteration to coordinates between `lo` and `hi`. It does this by seeking and skipping to the first coordinate greater than or equal to `lo`, and then breaking out of iteration at the first coordinate greater than or or equal to `hi`. To restrict the iteration space along with the desired stride `st`, our technique must also emit code that ensures any coordinate `c` read from the tensor aligns with the `st` by skipping coordinates where `(c - lo) % st != 0`. Finally, our technique emits code that projects a coordinate `c` read from iteration into the canonical iteration space by setting `c` equal to `(c - lo) / st`. At this point, the remaining steps for code generation can proceed as before, as the resulting coordinates are all within the slice and mapped to the canonical iteration space of the slice.

8 EVALUATION

We evaluate our sparse array programming compiler by comparing to the PyData/Sparse library, which is the only general sparse array language implementation known to us. We also compare to the less general SciPy/Sparse and GraphBLAS libraries, which consist of hand-implemented functions, to demonstrate our performance against hand-optimized code. Finally, we implement a medical imaging edge detection algorithm and the Minimax algorithm from game theory to demonstrate the applicability of our system. We restrict our evaluation to multi-core CPUs, as our implementation does not yet support GPUs.

8.1 Methodology

All experiments are run on a dual-socket, 12-core Intel Xeon E5-2680 v3 machine @ 2.5 GHz with 30 MB of L3 cache and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS. Our system and generated kernels are compiled with GCC 7.5.0. Python 3.6.9 is used to run all Python code. In our evaluation, we compare against PyData/Sparse [Abbasi 2018] version 0.11.2, SciPy [Virtanen et al. 2020] version 1.5.4, NumPy [Harris et al. 2020] version 1.19.5, and SuiteSparse:GraphBLAS [Davis 2019] version 4.0.3. We disable hyperthreading and use `numactl` to restrict execution to a single socket. All execution times, except in Section 8.3, are compared over an average of 10 executions.

8.2 Comparison to Sparse Array Programming Libraries

In the Python ecosystem, programmers have two main options for operating on sparse matrices or arrays: SciPy/Sparse and PyData/Sparse. SciPy/Sparse is a SciPy package for working with sparse matrices. It contains some common sparse matrix formats along with hand-written C implementations for many operations, but is limited in the scope of array programming features supported. For additions and multiplications, our system generates the exact same code as the TACO system [Kjolstad et al. 2017], which performs competitively with the hand-optimized implementations like those in SciPy [Chou et al. 2018].

PyData/Sparse is a recent project that supports tensors of arbitrary dimensions in the COO format. Like the NumPy library for dense array processing, it also supports general user-defined functions. The PyData/Sparse implementation utilizes existing NumPy and SciPy/Sparse dense kernels by first transforming and transposing the data into shapes that NumPy and SciPy/Sparse can operate on. Then, the PyData/Sparse algorithm will transform the results back into COO format. While the kernels used by PyData/Sparse are heavily optimized, its data transformation-based approach adds additional data movement overhead. By contrast, our techniques for sparse array programming can generate optimized kernels that operate on tensors of any dimension and data format, without performing unnecessary data movement.

8.2.1 Binary Operations. We demonstrate the flexibility and performance of our techniques by implementing a subset of the NumPy element-wise universal functions (ufuncs) that have iteration spaces different from intersection and union. We evaluate the `logical_xor`, `ldexp`, `right_shift` and `power` ufuncs, which have the iteration spaces shown in Fig. 20. SciPy/Sparse does not support most ufuncs outside of addition and multiplication and NumPy implementations cannot materialize the tensors into a dense format, so we restrict our comparison to PyData/Sparse.

Table 2. FROSTT tensors used in our evaluation

Tensor name	Non-zeros	Order	Shape
nips	3,101,609	4	2,482 x 2,862 x 14,036 x 17
uber-pickups	3,309,490	4	183 x 24 x 1,140 x 1,717
chicago-crime	5,330,673	4	6,186 x 24 x 77 x 32
vast	26,021,945	5	165,427 x 11,374 x 2 x 100 x 89
enron	54,202,099	4	6,066 x 5,699 x 244,268 x 1,176
nell-2	76,879,419	3	12,092 x 9,184 x 28,818

We evaluate the above ufuncs on the subset of real-valued tensors from the FROSTT tensor repository [Smith et al. 2017] and SuiteSparse sparse matrix repository [Davis and Hu 2011] that PyData/Sparse could successfully load without memory issues. Characteristics about the FROSTT

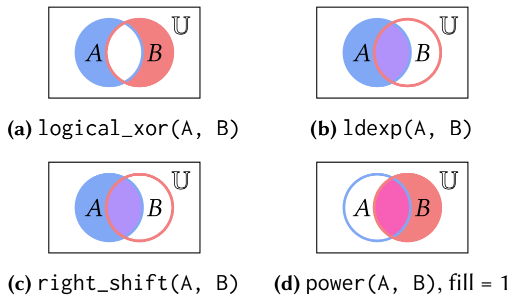


Fig. 20. Iteration spaces of the benchmarked ufuncs.

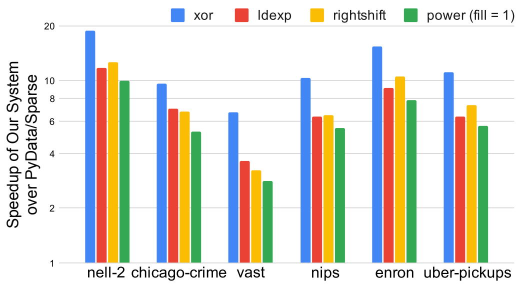


Fig. 21. Speedup of our system over PyData/Sparse on a log scale for ufunc operations on FROSTT tensors.

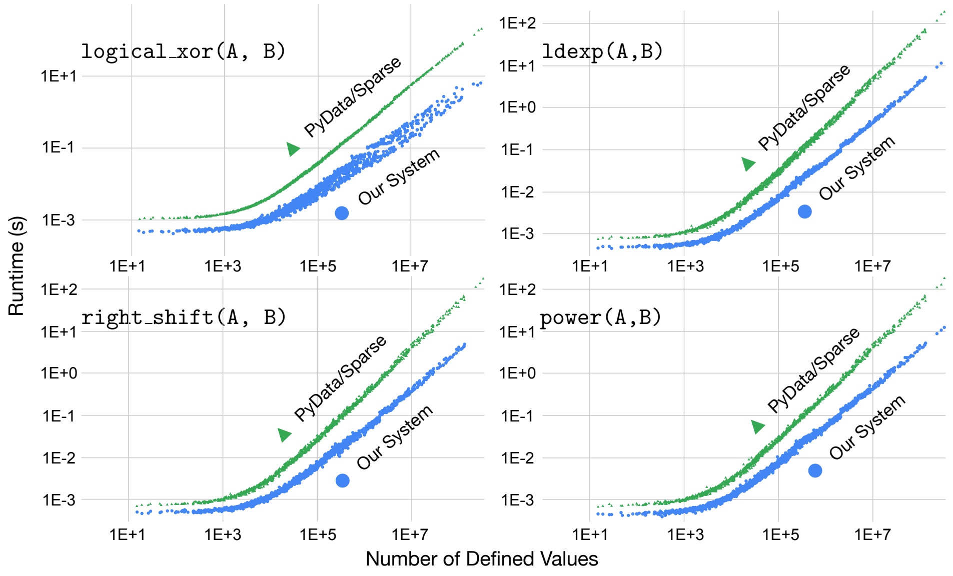
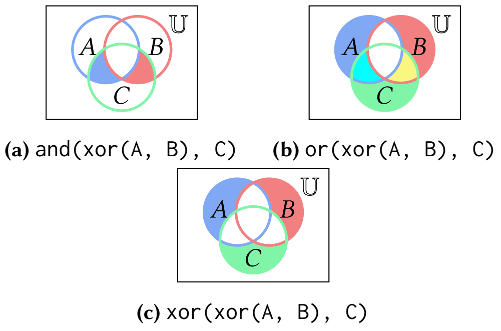


Fig. 22. Ufunc operations on SuiteSparse tensors using a log-log scale.

tensors satisfying these constraints can be found in Table 2. The tensors in these datasets were used as the first argument to the ufunc. We constructed synthetic inputs for the second argument by shifting the coordinates in the last tensor mode from the first argument by one position and setting the data to a constant value. Finally, since some ufuncs are sensitive to the particular value in the operand tensor (such as `ldexp`), we filled the shifted tensor with a small constant value of 2. Both PyData/Sparse and our system use a single thread for these kernels.

We show the normalized execution times of PyData/Sparse's ufunc operations on the FROSTT tensors in Fig. 21 and the execution times of our system and PyData/Sparse on the SuiteSparse matrices in Fig. 22. The geometric mean speedup of our system is 7.55× on the FROSTT tensors and 4.24× on the SuiteSparse matrices. The PyData/Sparse's data-movement heavy approach to sparse array programming is much slower than our approach on all inputs, which iterates directly through the sparse data structures for the exact iteration pattern of the target ufunc.



(a) `and(xor(A, B), C)` **(b)** `or(xor(A, B), C)` **(c)** `xor(xor(A, B), C)`

Fig. 24. Iteration spaces of fused ufunc benchmarks.

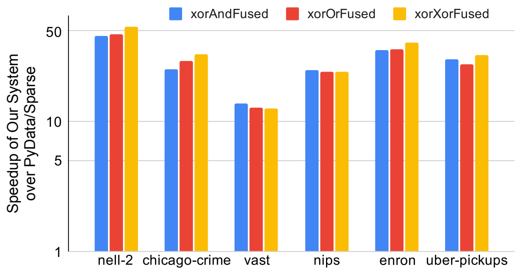


Fig. 25. Log-scale speedup of our system over PyData/Sparse for fused ufunc operations on FROSTT tensors.

8.2.2 Fusing Operations. Our approach for sparse array programming can also fuse different functions together, which avoids doing work that is then discarded. Our technique only iterates over the spaces where defined values are produced, which avoids materializing temporaries. We evaluate the fused kernels described in Fig. 24, which are constructed with the `logical_xor`, `logical_and`, and `logical_or` ufuncs. We use the FROSTT tensors described in Section 8.2.1 as inputs to the fused functions, with the third tensor argument created by the same shifting operation described previously. We report normalized execution times of fused operations in PyData/Sparse against our system in Fig. 25. By fusing operations, our system iterates over less data and avoids allocation of intermediate results. By contrast, PyData/Sparse's allocation of an intermediate and extra pass over the data cause it to have an even larger slowdown than for a single ufunc application.

Fusing functions can decrease the amount of work realized in the final output tensor. For example, consider the fused `and` and `xor` kernel described in Fig. 24. Without fusing, first `xor(A, B)` must be computed, and then the result must be `and`-ed with `C`. Since the annihilator of `and` is `false`, all coordinates in the iteration space of `xor` that are not present in `C` will be `false` in the final result. If the `and` and `xor` functions are fused, then the generated code avoids computing any values for `false` coordinates in `C` in the first place. To demonstrate this effect, we compare the fused `and−xor` kernel of PyData/Sparse to our system on a set of square matrices of increasing size where each matrix has a uniformly random sparsity of 1%. We plot the execution time of both systems against the number of defined values in the matrix in Fig. 23. Our system is already generally faster then PyData/Sparse, but our system's execution time still grows at a slower rate because it can avoid doing operations that are later ignored through fusing.

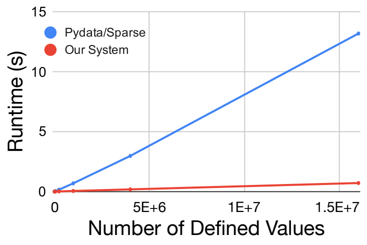


Fig. 23. Scaling of fused and/xor operation on random tensors.

8.2.3 Memory Usage. We present memory usage results between our system and PyData/Sparse on individual and fused ufunc operations on each of the considered FROSTT tensors. To measure the memory used for our system, we count the size of allocations performed to hold input and output data structures. Since our approach does not allocate any temporary data structures, this is all of the memory used by our system. To measure the memory used by PyData/Sparse, we use the Python package `memory_profiler`, which records a line-by-line profile of the total memory allocated/released by each line of a Python program.

We present the memory usage results in Figure 26, which groups the memory used data by each considered tensor in the FROSTT dataset. On average, we find that PyData/Sparse uses an average

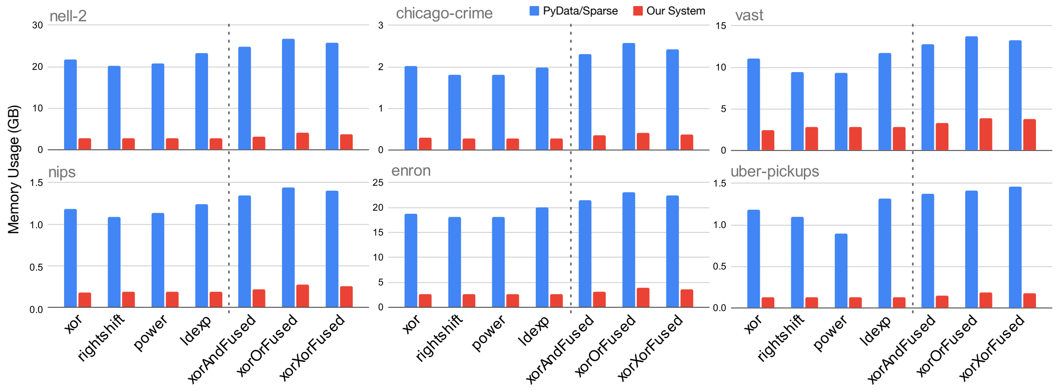


Fig. 26. Memory usage (in GB) of our system versus PyData/Sparse on FROSTT tensors. The experiments are run for both the ufunc and fused operations, which is differentiated by a vertical dashed line.

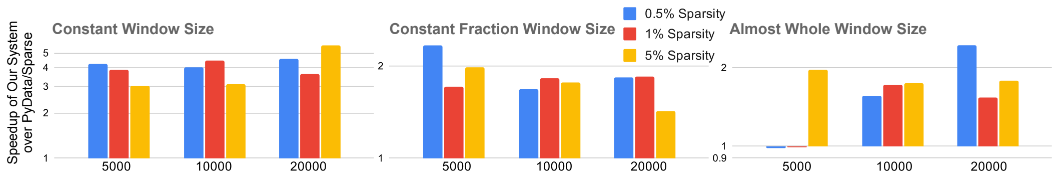


Fig. 27. Various window size slicing operations across different square matrix sizes, parameterized by sparsity and plotted on a log-scale.

of 6.55x more memory than our system to perform these ufunc operations. PyData/Sparse uses a similar amount of memory as our system to store the input and output tensors, but is unable to compress as much of these tensors since they are stored in a coordinate list format. Much of the measured space overhead of PyData/Sparse comes from large amount of intermediate storage allocated during execution of the ufunc operation, which is often more than twice the space used to store the input tensors themselves.

8.2.4 *Slicing.*

To evaluate the performance of code that our technique generates to perform slicing and striding, we perform an element-wise addition between uniformly random square sparse matrices with varying sparsities that have been sliced in different ways. We choose a simple kernel and input tensors to highlight the costs of data movement caused by slicing operations. We consider square slices of a constant 500×500 size, a constant fraction (1/4) of the matrices' rows, and slices that contain the entire matrix except for the first and last rows. We separately consider slices of the whole matrix with strides of widths 2, 4, and 8. Execution times normalized against our system for slicing and striding can be found in Figs. 27 and 28. We exclude results from PyData/Sparse in these figures as it was consistently an order of magnitude slower than both our system and SciPy/Sparse.

The geometric mean speedup of our system over SciPy/Sparse is 2.25× on the slicing benchmarks and 1.47× on the striding benchmarks. These speedups come from the implementation strategy of SciPy/Sparse and PyData/Sparse. In particular, SciPy/Sparse and PyData/Sparse implement slicing by first deep copying and repacking elements into a new sparse array, and then performing the desired computation. This deep copying and repacking step incurs extra cost compared to our approach, which operates directly on the slice.

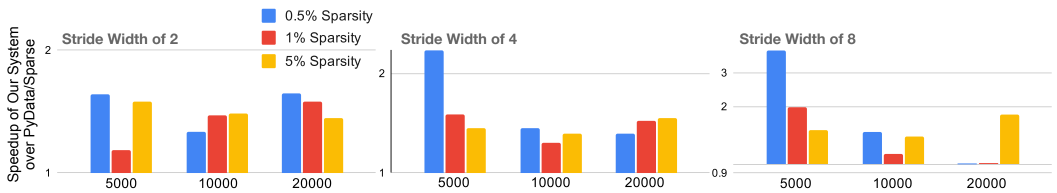


Fig. 28. Various stride length slicing operations across different square matrix sizes, parameterized by sparsity and plotted on a log-scale.

8.3 GraphBLAS Kernels

Many graph algorithms, such as those for performing breadth-first search and for solving the all-pairs shortest paths problem, can be expressed using linear algebra (with semirings beyond the standard $(+, \times)$ semiring) [Kepner and Gilbert 2011]. For instance, each iteration of breadth-first search on a graph can be expressed as the multiplication of the graph's adjacency matrix by a vector that represents the current frontier, which returns a new vector that represents the next set of vertices to be visited. Kepner et al. [2016] show how GraphBLAS, an API that exposes a fixed set of common linear algebra primitives like matrix-vector multiplication (mxv) and matrix-matrix multiplication (mxm), can be used to implement efficient graph applications.

Our technique can generate efficient code for many of the core primitives in GraphBLAS. To demonstrate this, we use our technique to generate code that implement mxv

$$y_i = \texttt{mask}(\neg m_i, \bigoplus_j (A_{ij} \otimes x_j))$$

and mxm ($A_{ij} = \bigoplus_k (B_{ik} \otimes C_{kj})$) for both Boolean and tropical semirings. (The tropical semiring replaces $\oplus$ with min and $\otimes$ with $+$, while the Boolean semiring assumes that all values are Boolean and replaces $\oplus$ with $\vee$ and $\otimes$ with $\wedge$. mask returns the value of the second argument only if the first argument evaluates to true.) We then measure the performance of the generated code (running with 12 threads) and compare it against that of SuiteSparse:GraphBLAS [Davis 2019], a highly-optimized implementation of GraphBLAS. We report average execution times over 1000 iterations for mxv and over 100 iterations for mxm.

Tables 3 and 4 show the results of our experiment. For mxv, our system is on average 1.26× faster than SuiteSparse:GraphBLAS (i.e., SuiteSparse) when computing with the Boolean semiring and 1.13× faster when computing with the tropical semiring. This is because our technique generates code that uses the same algorithm as SuiteSparse but is fully specialized to the input's types and formats. By contrast, SuiteSparse, though partially specialized using C macros, still has to perform some dynamic dispatching to handle inputs of arbitrary types and formats, which adds run-time overhead. For mxm, our technique is on average 1.02× faster than SuiteSparse when computing with the Boolean semiring and has performance 0.836× that of SuiteSparse when computing with the tropical semiring. Our generated code and SuiteSparse both use a linear combination of rows algorithm to compute the kernel. However, when the output matrix has relatively few defined values per row, SuiteSparse is able to use hash tables to store partial results. By contrast, our technique currently can only generate code that stores partial results using dense arrays, thus reducing cache efficiency. Table 4 also shows, though, that code our technique emits has similar or better performance on average than SuiteSparse when the latter also uses dense arrays to store partial results (which is preferable when the output is relatively dense).

Table 3. Performance of (complement-masked) matrix-vector multiplication (`mxv`) kernels, generated by our sparse array compiler and implemented in SuiteSparse:GraphBLAS, on varying SuiteSparse matrices. Relative and absolute execution times (in parentheses) are shown, with the faster implementation highlighted in gray. We use input vectors that are 25% dense and mask vectors with 25% of elements being false. All matrices are stored in CSR, while mask vectors are stored using dense arrays and other vectors are stored using byte maps. Our technique generates code that is competitive with SuiteSparse:GraphBLAS in terms of performance, with the generated code being 1.13–1.26× as fast as hand-optimized code on average.

Matrix	Boolean Semiring		Tropical Semiring	
	SuiteSparse	Our System	SuiteSparse	Our System
belgium_osm	1× (1.62 ms)	1.38× (1.17 ms)	1× (2.33 ms)	1.05× (2.22 ms)
cit-Patents	1× (8.08 ms)	1.33× (6.06 ms)	1× (17.3 ms)	1.19× (14.6 ms)
coAuthorsCiteseer	1× (0.336 ms)	1.80× (0.186 ms)	1× (0.561 ms)	1.35× (0.417 ms)
coPapersDBLP	1× (1.76 ms)	1.77× (0.993 ms)	1× (9.19 ms)	1.18× (7.79 ms)
delaunay_n24	1× (27.2 ms)	0.768× (35.5 ms)	1× (68.4 ms)	0.98× (69.7 ms)
indochina-2004	1× (17.8 ms)	1.22× (14.7 ms)	1× (46.3 ms)	0.967× (47.9 ms)
rgg_n_2_24_s0	1× (47.6 ms)	0.941× (50.6 ms)	1× (138 ms)	1.18× (116 ms)
road_central	1× (25.5 ms)	1.04× (24.4 ms)	1× (52.8 ms)	1.01× (52.1 ms)
road_usa	1× (31.6 ms)	0.889× (35.6 ms)	1× (73 ms)	0.957× (76.2 ms)
roadNet-CA	1× (2.17 ms)	1.35× (1.61 ms)	1× (5.5 ms)	1.54× (3.57 ms)
ship_003	1× (0.214 ms)	2.00× (0.107 ms)	1× (1.04 ms)	1.11× (0.93 ms)
soc-LiveJournal1	1× (15.2 ms)	1.25× (12.2 ms)	1× (46.3 ms)	1.12× (41.3 ms)
Geometric mean	1×	**1.26×**	1×	**1.13×**

Table 4. Performance of matrix-matrix multiplication (`mxm`) kernels, generated by our sparse array compiler and implemented in SuiteSparse:GraphBLAS, on varying SuiteSparse matrices. Relative and absolute execution times (in parentheses) are shown, with the fastest implementation highlighted in gray. All matrices are stored in CSR, and we use each matrix as both inputs to the kernel. For SuiteSparse:GraphBLAS, we report results for whichever algorithm the library chooses (Any) as well as for their implementation of Gustavson's algorithm, which uses dense arrays to store partial results. Again, on the whole, our technique generates code that is competitive with SuiteSparse:GraphBLAS in terms of performance, with the generated code being 0.836–1.02× as fast as hand-optimized code on average.

Matrix	Boolean Semiring			Tropical Semiring		
	SuiteSparse (Any)	Our System	SuiteSparse (Gustavson's)	SuiteSparse (Any)	Our System	SuiteSparse (Gustavson's)
belgium_osm	1× (0.032 s)	0.896× (0.036 s)	0.256× (0.125 s)	1× (0.040 s)	0.599× (0.067 s)	0.243× (0.166 s)
cit-Patents	1× (0.655 s)	0.804× (0.815 s)	0.506× (1.30 s)	1× (0.816 s)	0.577× (1.41 s)	0.451× (1.81 s)
coAuthorsCiteseer	1× (0.077 s)	1.40× (0.055 s)	1.02× (0.076 s)	1× (0.118 s)	1.14× (0.104 s)	0.921× (0.128 s)
coPapersDBLP	1× (2.48 s)	1.12× (2.2 s)	1.00× (2.48 s)	1× (4.27 s)	0.946× (4.51 s)	1.00× (4.27 s)
delaunay_n24	1× (1.99 s)	1.05× (1.90 s)	0.966× (2.06 s)	1× (2.49 s)	0.955× (2.61 s)	0.938× (2.65 s)
indochina-2004	1× (120 s)	0.997× (121 s)	1.00× (120 s)	1× (207 s)	0.748× (276 s)	0.999× (207 s)
rgg_n_2_24_s0	1× (7.77 s)	1.23× (6.34 s)	1.21× (6.44 s)	1× (10.8 s)	1.13× (9.59 s)	1.12× (9.69 s)
road_central	1× (0.941 s)	0.852× (1.1 s)	0.66× (1.43 s)	1× (1.11 s)	0.661× (1.68 s)	0.627× (1.77 s)
road_usa	1× (0.777 s)	0.689× (1.13 s)	0.698× (1.11 s)	1× (0.966 s)	0.558× (1.73 s)	0.681× (1.42 s)
roadNet-CA	1× (0.064 s)	0.836× (0.077 s)	0.837× (0.076 s)	1× (0.083 s)	0.687× (0.121 s)	0.832× (0.1 s)
ship_003	1× (0.14 s)	1.28× (0.109 s)	1.22× (0.116 s)	1× (0.236 s)	1.31× (0.18 s)	1.19× (0.198 s)
soc-LiveJournal1	1× (22.6 s)	1.32× (17.2 s)	1.18× (19.1 s)	1× (42.4 s)	1.17× (36.1 s)	1.10× (38.7 s)
Geometric mean	1×	**1.02×**	**0.815×**	1×	**0.836×**	**0.778×**

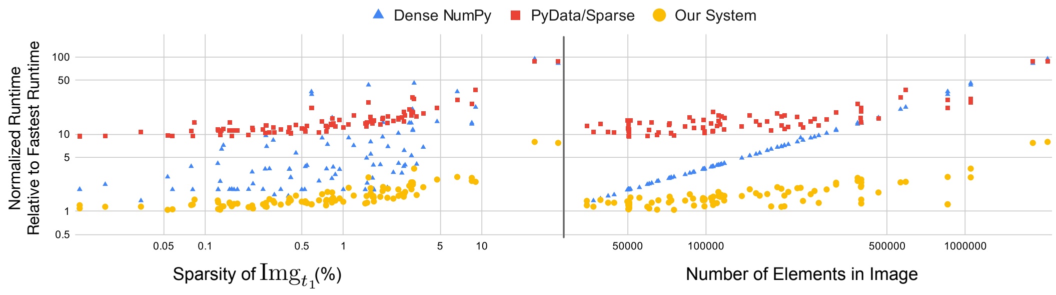


Fig. 29. Medical imaging edge detection performance normalized by the fastest runtime point on a log-log scale plotted with respect to Img_{t_1} sparsity (left) and number of image pixels (right). The normalized runtime value of 1 corresponds to an absolute runtime of 244.7 μs

8.4 Applications

To demonstrate the usefulness of our system, we used it to implement two algorithms: an edge detection algorithm from medical imaging and the MinMax algorithm for game decision making. We compare our system to implementations using NumPy and PyData/Sparse.

8.4.1 Medical Imaging Edge Detection. Image processing and computer vision approaches often use array programming. More specifically, the medical imaging field applies these processing techniques to patient images from various imaging modalities. Oftentimes, after initial measurements are taken, the measurements are post-processed for digital enhancement, diagnostic purposes, and even to create domain specific machine learning models. Many systems and libraries exist for (grayscale) medical image analysis that include functions like logical `and`, `xor`, `or`, and `not` [Huang et al. 2006; Kim et al. 2000; Wollny et al. 2013]. We implement boundary edge detection on magnetic resonance imaging (MRI) images [Somkantha et al. 2011] to demonstrate the practical application of our sparse array programming system. We implement a computer vision thresholding technique to determine the edges of an MRI image, which are then filtered using a region-of-interest (ROI) mask (see Appendix Fig. 33 in the supplemental materials[2]). The masked edge detection is represented by the equation $\mathrm{Img}_{post} = (\mathrm{Img}_{t_2} \wedge \mathrm{ROI}) \oplus (\mathrm{Img}_{t_1} \wedge \mathrm{ROI})$, where Img is the original two-dimensional single-channel MRI image and Img_{t_1} and Img_{t_2} are thresholded versions of Img using $t_1 = 75\%$ and $t_2 = 80\%$ respectively.

We compare the average execution time of the masked edge detection on MRI brain images from a dataset on Kaggle [Chakrabarty 2019]; an example image can be found in Appendix Fig. 33 in the supplemental materials[2]. Our sparse array compiler has a geometric mean speedup of 2.69× (0.96× to 28.9×) faster than the dense NumPy implementation and 9.41× (6.58× to 17.9×) faster than the PyData/Sparse implementation, as shown in Fig. 29. We also demonstrate the benefits of sparsity since the dense implementation scales linearly with the number of image pixels.

8.4.2 Game Playing Minimax Algorithm. In game theory, game choices are often represented as a decision tree where each node represents a potential state in the game and each edge represents a move decision, with leaf-node values representing a heuristic of that game state (see Fig. 31). In addition to interpreting sparse arrays as images (Section 8.4.1) or graphs (Section 8.3), we can use sparse arrays to represent tree-like structures. Artificial intelligence algorithms, like the Minimax algorithm, are often used on these game-state decision trees to calculate the optimal move given a starting game position and assuming that the opponent will also choose their moves optimally. Our

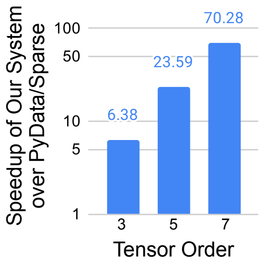
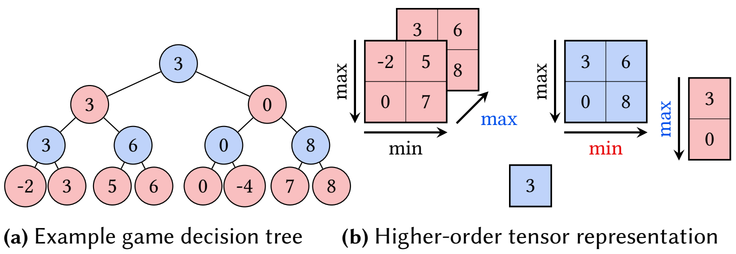


Fig. 30. Minimax speedup plotted on a log-scale



(a) Example game decision tree
(b) Higher-order tensor representation
max
min
max
max
min
max

Fig. 31. Simple Minimax Algorithm Example

system can represent the Minimax algorithm, which alternates taking the minimum and maximum value at each level of the game tree, as: $\text{opt} = \max_i \min_j \max_k \ldots (A_{ijk\ldots})$.

We implement this algorithm on our system and compare against PyData/Sparse for higher-order tensors. We cannot compare against NumPy since the tensors are too large for a dense representation, and we cannot compare against SciPy/Sparse since these tensors have greater than two dimensions. We test this for tensor orders $o = 3, 5, 7$, where the dimensions are $20 \times 20 \times 43^{(o-2)}$ to represent the first o moves of a chess game; this was chosen since chess has 20 opening moves and then 43 moves on average for a board state at any given time. The sparsity of the tensor comes from sparse sampling and pruning of the decision tree, where the fill values represent pruned nodes. Fig. 30 illustrates that we outperform PyData/Sparse by 6.38× to 70.3× depending on the tensor order and that our performance improves significantly with increasing tensor order.

9 RELATED WORKS

We will explore four areas of related work, ordered from most to least relevant. We begin with prior work on execution of array programs on sparse arrays, which is divided into two strategies: generating bespoke code for each operation or emulating sparse array programs by reorganizing data and then calling hand-written functions. We then survey the large body of work on array programming models (for dense arrays). And finally, we discuss prior work on generalizing sparse linear and tensor algebra for machine learning and graph algorithms.

Sparse Array Language Compilation

This paper is the first to describe how to generate bespoke code for general sparse array programs—any function applied across sparse and dense arrays with any fill value, including element-wise application, broadcasts, and reduction. But there exists prior work on generating code for sparse linear and tensor algebra, which are subsets of sparse array languages where the functions must be additions and multiplications applied in linear expressions.

Most directly related to our work is the body of work on the Sparse Tensor Algebra Compiler (TACO) [Chou et al. 2018, 2020; Kjolstad et al. 2019; Kjolstad et al. 2017; Senanayake et al. 2020]. Our work shows how to generalize the compilation theory behind TACO [Kjølstad 2020] to the much broader class of array programs, by allowing any function to be applied to sparse arrays with any fill value. We achieve this generalization by introducing functions with annotated properties, an iteration algebra containing complements, and omitter points to iteration lattices.

Other prior work on generating bespoke code for subsets of sparse linear algebra include the MT1 [Bik and Wijshoff 1993], Bernoulli [Kotlyar et al. 1997], and CHiLL-I/E [Venkat et al. 2015] compilers, which analyze and transform imperative code that implements dense linear algebra kernels to sparse implementations. CHiLL-I/E can transform dense and sparse multiplication operations on matrices and vectors to implementations where one operand is sparse. MT1 supports those operations as well as several other built-in operators and intrinsic functions, such as +, ==, and `sqrt`. It can also generate code for operations with multiple sparse data structures by introducing dense temporaries, thus turning them into sparse-dense iteration. Bernoulli maps dense linear algebra implementations to relational algebra and then further maps the relational algebra to templated sparse implementations.

Sparse Array Language Emulation

The alternative explored in prior work to generating bespoke code for sparse array computations is to emulate sparse array programs using a finite set of hand-written implementations. That approach requires data movement to reshape the data to the available functions. An early system of this sort is the MATLAB Tensor Toolbox [Bader and Kolda 2007], which executes high-order tensor algebra by re-organizing the tensors to look like matrices. Since this approach requires pre/post-processing to re-organize data, it is slower than bespoke generated implementations.

We know of only one prior system for the general category of sparse array programming languages, namely the PyData/Sparse library [Abbasi 2018]. This system executes one two-operand operation at a time in the following steps: First, a hand-written function iterates through the defined elements of the two array operands and divides them into three sets: those that both have defined values, those that only the first operand has, and those that only the second operand has. Next, it invokes NumPy's dense implementations to compute the function at hand on each of those subspaces. And finally, it re-integrates the three sets of resulting values into a result array. By contrast, our work generates bespoke implementations that do not require data pre/post-processing and therefore performs significantly better, as shown in our evaluation.

Array Languages

There is a large body of prior work on dense array programming models, as defined in this paper, going back to APL [Iverson 1962]. Modern variants include ZPL [Lin and Snyder 1993] and NumPy [Harris et al. 2020], but the core operations—element-wise operations, reductions, and broadcasts—remain the same. Furthermore, many compiler techniques have been developed to compile dense array programs, including the polyhedral model [Lamport 1974]. Our novel contribution is compiling the array programming model to sparse arrays. Another key insight, which differs from dense array programming models, is that functions applied across sparse arrays must be decorated with algebraic properties for the system to be able to generate efficient code.

Generalizations of Sparse Linear and Tensor Algebra

Two additional bodies of work have made steps towards the full generalization of sparse array programming, by generalizing sparse linear and tensor algebra to compute sparse neural networks and graph algorithms. Systems with support for sparse neural networks must support non-linear functions in addition to sparse linear algebra. For example, PyTorch [Paszke et al. 2019] supports hand-implemented `softmax` and `log_softmax` on sparse tensors, while TensorFlow [Abadi et al. 2016] supports max element-wise and reduction operations. We expect new non-standard operations to keep arising in the future, which motivates a comprehensive sparse array programming model.

In addition, several researchers [Davis 2019; Kepner et al. 2016; Mattson et al. 2013] have defined and implemented APIs, namely GraphBLAS, for linear algebra computations where the operations are different semirings than $(+, \times)$, such as $(\wedge, \vee)$ or $(\min, +)$. Computations in different types of semirings provide surprising features, such as the ability to compute several graph algorithms through matrix multiplications. And since all semirings behave linearly, the same implementation can be reused by just replacing the meaning addition and multiplication. Researchers have also proposed sparse tensor algebra libraries with support for multiple semirings [Solomonik and Hoefler 2015]. Sparse array programming supports operations in different semirings, but generalizes the programming model to computations with any function, whether it partakes in a semiring or not. Furthermore, our work generalizes the arrays to support any fill value.

10 FUTURE WORK

We see many avenues of future work, both in transferring our technology into industry and building on our theory and implementation. We are investigating how to integrate our technology into third-party ecosystems such as PyData/Sparse to serve as a code generation back-end. This effort would allow the larger Python community to take advantage of the general sparse array programming techniques that we describe. We also aim to explore new research areas stemming from this work, such as how to provide compiler feedback to the user in catching both performance and correctness bugs around user supplied function properties and iteration spaces. We also foresee interesting combinations of sparse iteration algebra with the polyhedral model's powerful dependency analysis and transformation primitives. Finally, we aim to extend our implementation to target GPUs so that programmers can take advantage of GPU acceleration in their sparse array programs. And, beyond GPUs, we are exploring compilation of sparse array programs to both a new class of sparse domain-specific architectures and to distributed cloud and supercomputers.

11 CONCLUSION

This paper shows how to build a general compiler for array programs on sparse arrays, by generalizing prior work on tensor algebra compilation. The resulting compiler can generate efficient code for programs that apply any function, annotated with algebraic properties, across any number of sparse arrays. It supports reductions, broadcasts, slicing, and the data structures and fill values can be selected independently for each array. Moreover, the compiler can fuse together operations using different functions by generating a joint sparse iteration space. The expressive power of the sparse array programming language supported by the system is sufficient to encompass dense and sparse tensor algebra, array programming languages like NumPy, and GraphBLAS systems for graph algorithms in linear algebra with semirings.

12 ACKNOWLEDGEMENTS

We thank Alex Aiken, Axel Feldmann, Charles Yuan, Aviral Pandey, and Peter Ahrens for their helpful feedback. Olivia Hsu and Rohan Yadav were supported by an NSF GRFP Fellowship. This work was supported in part by the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008923 and DE-SC0018121; DARPA under Awards HR0011-18-3-0007 and HR0011-20-9-0017; and NSF under Grant Numbers 1937301, 2028602, CCF-1563078, 1563113. This research was also supported in part by a Google Research Scholar and the Stanford Data Analytics for What's Next (DAWN) Affiliate Program—with affiliate members Ant Financial, Facebook, Google, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

Hameer Abbasi. 2018. Sparse: a more modern sparse array library. In *Proceedings of the 17th Python in Science Conference.* 27–30.

John W. Backus, R. J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, Harold Stern, Irving Ziller, Robert A. Hughes, and Roy Nutt. 1957. The FORTRAN automatic coding system. In *Western Joint Computer Conference.* Los Angeles, California, 188–198. https://doi.org/10.1145/1455567.1455599

Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB Computations with Sparse and Factored Tensors. *Journal on Scientific Computing* 30, 1 (2007), 205–231. https://doi.org/10.1137/060676489

Aart J. C. Bik and Harry A. G. Wijshoff. 1993. Compilation Techniques for Sparse Matrix Computations. In *International Conference on Supercomputing.* ACM, 416–424. https://doi.org/10.1145/165939.166023

Navoneel Chakrabarty. 2019. *Brain MRI Images for Brain Tumor Detection.* https://www.kaggle.com/navoneel/brain-mri-images-for-brain-tumor-detection

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

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

Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

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

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, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.

Su Huang, Rafail Baimouratov, Pengdong Xiao, Anand Ananthasubramaniam, and Wieslaw L Nowinski. 2006. A Medical Imaging and Visualization Toolkit in Java. *Journal of Digital Imaging* 19, 1 (2006), 17–29. https://doi.org/10.1007/s10278-005-9247-6

Kenneth E Iverson. 1962. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference.* 345–351.

J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC).* 1–9. https://doi.org/10.1109/HPEC.2016.7761646

Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, USA.

Jinman Kim, David D. Feng, and Tom W. Cai. 2000. A Web Based Medical Image Data Processing and Management System. In *Selected Papers from the Pan-Sydney Workshop on Visualisation - Volume 2* (Sydney, Australia) *(VIP '00).* Australian Computer Society, Inc., AUS, 89–91.

F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 180–192. https://doi.org/10.1109/CGO.2019.8661185

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

Fredrik Berg Kjølstad. 2020. *Sparse tensor algebra compilation.* Ph.D. Dissertation. Massachusetts Institute of Technology.

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par Parallel Processing.* Springer, Passau, Germany, 318–327. https://doi.org/10.1007/BFb0002751

Leslie Lamport. 1974. The Parallel Execution of DO Loops. *Commun. ACM* 17, 2 (1974), 83–93. http://research.microsoft.com/en-us/um/people/lamport/pubs/do-loops.pdf

Calvin Lin and Lawrence Snyder. 1993. ZPL: An array sublanguage. In *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 96–114.

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

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary 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. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

SciPy. 2021. SciPy Roadmap V1.6.2. https://docs.scipy.org/doc/scipy-1.6.2/reference/roadmap.html [Online; accessed 04/12/2021].

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. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. http://frostt.io/

Edgar Solomonik and Torsten Hoefler. 2015. Sparse tensor algebra as a parallel programming model. *arXiv preprint arXiv:1512.00066* (2015).

K. Somkantha, N. Theera-Umpon, and S. Auephanwiriyakul. 2011. Boundary Detection in Medical Images Using Edge Following Algorithm Based on Intensity Gradient and Texture Gradient Features. *IEEE Transactions on Biomedical Engineering* 58, 3 (2011), 567–573. https://doi.org/10.1109/TBME.2010.2091129

Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. 521–532. https://doi.org/10.1145/2737924.2738003

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

Gert Wollny, Peter Kellman, María J. Ledesma-Carbayo, Matthew M. Skinner, Jean-Jaques Hublin, and Thomas Hierl. 2013. MIA - A free and open source software for gray scale medical image analysis. *Source Code Biol Med*, Article 20 (2013). https://doi-org.stanford.idm.oclc.org/10.1186/1751-0473-8-20

A APPENDIX

A.1 Array Index Notation Grammar

The full syntax of array index notation can be found in Figure 32.

$$
\begin{array}{lcl}
\langle \mathit{array\_stmt} \rangle & ::= & \langle \mathit{access} \rangle \text{ `='} \langle \mathit{expr} \rangle \\
\langle \mathit{access} \rangle & ::= & \langle \mathit{tensor} \rangle_{\{\langle \mathit{index} \rangle\}} \\
\langle \mathit{index} \rangle & ::= & \langle \mathit{index\_var} \rangle \; [\; \langle \mathit{index\_slice} \rangle \;] \\
\langle \mathit{index\_slice} \rangle & ::= & \text{`('} \; \langle \mathit{lo} \rangle \; \text{`:'} \; \langle \mathit{hi} \rangle \; [\; \text{`:'} \; \langle \mathit{st} \rangle \;] \; \text{`)'} \\
\langle \mathit{expr} \rangle & ::= & \langle \mathit{literal} \rangle \mid \langle \mathit{access} \rangle \mid \langle \mathit{call\_expr} \rangle \mid \langle \mathit{reduce\_expr} \rangle \\
& \mid & \langle \mathit{binary\_expr} \rangle \mid \langle \mathit{unary\_expr} \rangle \mid \text{`('} \; \langle \mathit{expr} \rangle \; \text{`)'} \\
\langle \mathit{call\_expr} \rangle & ::= & \langle \mathit{func} \rangle \; \text{`('} \; \langle \mathit{expr} \rangle \; \{ \text{`,'} \; \langle \mathit{expr} \rangle \} \; \text{`)'} \\
\langle \mathit{reduce\_expr} \rangle & ::= & \underset{\langle \mathit{index\_var} \rangle}{\langle \mathit{func} \rangle} \; \langle \mathit{expr} \rangle \\
\langle \mathit{binary\_expr} \rangle & ::= & \langle \mathit{expr} \rangle \; \langle \mathit{op} \rangle \; \langle \mathit{expr} \rangle
\end{array}
$$

Fig. 32. The syntax of array index notation. Expressions within braces may be repeated any number of times. $\langle func \rangle$ and $\langle op \rangle$ both represent arbitrary (user-defined or predefined) functions and are implemented in the same way; they differ only in how they are invoked.

A.2 PyData/Sparse API

An example of performing the xor operation on two sparse tensors using PyData/Sparse is found below.

```python
import numpy
import sparse

# Create some tensors.
dim = 1000
A = sparse.random((dim, dim, dim))
B = sparse.random((dim, dim, dim))
# Perform the XOR computation.
C = numpy.logical_xor(A, B)
```

An example performing the GCD operation can be found below:

```python
import numpy
import

def gcd(x, y):
  return ... # Compute the GCD between x and y.
# Register the gcd function as a ufunc.
gcd = np.frompyfunc(gcd, 2, 1)

# Create some tensors.
dim = 1000
A = sparse.random((dim, dim, dim))
B = sparse.random((dim, dim, dim))
# Perform the XOR computation.
C = gcd(A, B)
```

While this code is simpler than the code to use our sparse array compiler, users do not have control over many factors, such as the formats of the tensors, and are restricted to the predefined set of NumPy functions.

A.3 Iteration Lattice Construction Algorithm

As described in Section 6, the presented iteration lattice construction algorithm (Algorithm 1) supports only array index notation expressions that do not contain repeat tensors. Fig. 18 illustrates an example of when iteration sub-spaces do not overlap when the index notation contains a repeated tensor. This example motivates our implementation of a filtered Cartesian Product.

We include the full algorithm that does support repeated tensors in Algorithm 2.

Algorithm 2 Full iteration lattice construction algorithm

```
procedure ConstructLattice (FunctionAlgebra A, FunctionArguments args)
    // Preprocessing steps
    Algebra A = DeMorgan(A)                                  ▷ Apply De Morgan's Law
    Algebra A = Augment(A, args)                             ▷ Augmentation pass
    return BuildLattice(A)
end procedure

// let $\mathcal{L}$ represent an iteration lattice and $p$ represent an iteration lattice point
procedure BuildLattice (Algebra A)
    if A is Tensor(t) then                                   ▷ Segment Rule
        return $\mathcal{L}(p(\{ t \}, \text{producer=true}))$
    else if A is ~Tensor(t) then                             ▷ Complement Rule
        $p_o = p(\{ t, \mathbb{U} \}, \text{producer=false})$
        $p_p = p(\{ \mathbb{U} \}, \text{producer=true})$
        return $\mathcal{L}(\{ p_o, p_p \})$
    else if A is (left ∩ right) then                         ▷ Intersection Rule
        $\mathcal{L}_l, \mathcal{L}_r$ = BuildLattice(left), BuildLattice(right)
        cp = FilteredCartesianProduct($\mathcal{L}_l$.points(), $\mathcal{L}_r$.points())
        mergedPoints = $\{ p(\{ p_l + p_r \}, \text{producer=}p_l\text{.producer} \wedge p_l\text{.producer}) : \forall (p_l, p_r) \in \text{cp} \}$
        mergedPoints = RemoveDuplicates(mergedPoints, ommitterPrecedence)
        return $\mathcal{L}$(mergedPoints)
    else if A is (left ∪ right) then                         ▷ Union Rule
        $\mathcal{L}_l, \mathcal{L}_r$ = BuildLattice(left), BuildLattice(right)
        cp = FilteredCartesianProduct($\mathcal{L}_l$.points(), $\mathcal{L}_r$.points())
        mergedPoints = $\{ p(\{ p_l + p_r \}, \text{producer=}p_l\text{.producer} \vee p_l\text{.producer}) : \forall (p_l, p_r) \in \text{cp} \}$
        mergedPoints = mergedPoints + $\mathcal{L}_l$.points() + $\mathcal{L}_r$.points()
        mergedPoints = RemoveDuplicates(mergedPoints, producerPrecedence)
        return $\mathcal{L}$(mergedPoints)
end procedure

procedure FilteredCartesianProduct (LatticePoints left, LatticePoints right)
    $p_{l,\text{root}}, p_{r,\text{root}}$ = left.root, right.root
    for ($p_l$ in left) × ($p_r$ in right) do overlap = true
        for tensor in $p_l$ do
            if (tensor in $p_{r,\text{root}}$) ∧ (tensor not in $p_l$) then overlap = false
        end for
        for tensor in $p_r$ do
            if (tensor in $p_{l,\text{root}}$) ∧ (tensor not in $p_l$) then overlap = false
        end for
        if overlap then cp += {($p_l$, $p_r$)}
    end for
    return cp
end procedure
```

A.4 Medical Imaging Edge Detection

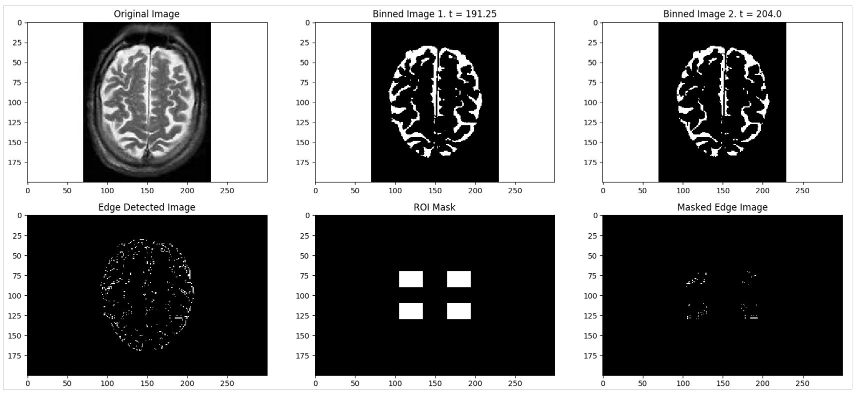


Fig. 33. Example MRI image, thresholding, ROI mask, and output