Finch: Sparse and Structured Tensor Programming with Control Flow

WILLOW AHRENS, MIT CSAIL, USA
TEODORO FIELDS COLLIN, MIT CSAIL, USA
RADHA PATEL, MIT CSAIL, USA
KYLE DEEDS, University of Washington, USA
CHANGWAN HONG, MIT CSAIL, USA
SAMAN AMARASINGHE, MIT CSAIL, USA

From FORTRAN to NumPy, tensors have revolutionized how we express computation. However, tensors in these, and almost all prominent systems, can only handle dense rectilinear integer grids. Real world tensors often contain underlying structure, such as sparsity, runs of repeated values, or symmetry. Support for structured data is fragmented and incomplete. Existing frameworks limit the tensor structures and program control flow they support to better simplify the problem.

In this work, we propose a new programming language, Finch, which supports *both* flexible control flow and diverse data structures. Finch facilitates a programming model which resolves the challenges of computing over structured tensors by combining control flow and data structures into a common representation where they can be co-optimized. Finch automatically specializes control flow to data so that performance engineers can focus on experimenting with many algorithms. Finch supports a familiar programming language of loops, statements, ifs, breaks, etc., over a wide variety of tensor structures, such as sparsity, run-length-encoding, symmetry, triangles, padding, or blocks. Finch reliably utilizes the key properties of structure, such as structural zeros, repeated values, or clustered non-zeros. We show that this leads to dramatic speedups in operations such as SpMV and SpGEMM, image processing, and graph analytics.

CCS Concepts: • **Software and its engineering** → **Control structures**; **Data types and structures**; *Imperative languages*; • **Mathematics of computing** → **Mathematical software**.

Additional Key Words and Phrases: Sparse Tensor, Structured Tensor, Control Flow, Programming Language

ACM Reference Format:
Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and Structured Tensor Programming with Control Flow. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 117 (April 2025), 31 pages. https://doi.org/10.1145/3720473

1 Introduction

Arrays are the most fundamental abstraction in computer science. Arrays and lists are often the first-taught datastructure [4, Chapter 2.2], [59, Chapter 2.2]. Arrays are also universal across programming languages, from their introduction in Fortran in 1957 to present-day languages like Python [11], keeping more-or-less the same semantics. Modern array programming languages such as NumPy [48], SciPy [94], MatLab [71], TensorFlow [2], PyTorch [74], and Halide [76] have

Authors' Contact Information: Willow Ahrens, MIT CSAIL, Cambridge, Massachusetts, USA, willow@csail.mit.edu; Teodoro Fields Collin, MIT CSAIL, Cambridge, Massachusetts, USA, teoc@mit.edu; Radha Patel, MIT CSAIL, Cambridge, Massachusetts, USA, rrpatel@alum.mit.edu; Kyle Deeds, University of Washington, Seattle, Washington, USA, kdeeds@cs.washington.edu; Changwan Hong, MIT CSAIL, Cambridge, Massachusetts, USA, changwan@mit.edu; Saman Amarasinghe, MIT CSAIL, Cambridge, Massachusetts, USA, saman@csail.mit.edu.

ACM 2475-1421/2025/4-ART117
https://doi.org/10.1145/3720473

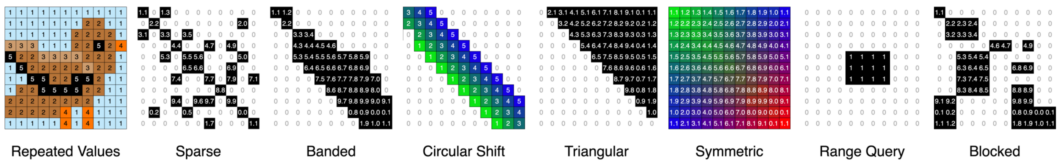


Fig. 1. A few examples of matrix structures arising in practice

pushed the limits of productive data processing with arrays, fueling breakthroughs in machine learning, scientific computing, image processing, and more.

The success and ubiquity of arrays is largely due to their simplicity. Since their introduction, multidimensional arrays have represented dense, rectilinear, integer grids of points. By **dense**, we mean that indices are mapped to value via a simple formula relating multidimensional space to linear memory. Consequently, dense arrays offer extensive compiler optimizations and many convenient interfaces. Compilers understand dense computations across many programming constructs, such as for and while loops, breaks, parallelism, caching, prefetching, multiple outputs, scatters, gathers, vectorization, loop-carry-dependencies, and more. A myriad of optimizations have been developed for dense arrays, such as loop fusion, loop tiling, loop unrolling, and loop interchange. However, while dense arrays are the easiest way to program for performance, real world applications often require more complex data structures to reach peak efficiency.

Our world is full of structured data. In this work, we make the distinction between a **tensor**, which describes any multidimensional object which relates tuples of integer coordinates to values, such as vectors or matrices, and an **array**, the previously described classical data structure. We say that a tensor is **structured** when it has patterns that allows us to optimize storage or computation of the tensor. Sparse tensors (which store only nonzero elements) describe networks, databases, and simulations [5, 13, 17, 69]. Run-length encoding describes images, masks, geometry, and databases (such as a list of transactions with the date field all the same) [43, 82]. Symmetry, bands, padding, and blocks arise due to modeling choices in scientific computing (e.g., higher order FEMs) as well as in intermediate structures in many linear solvers (e.g., GMRES) [25, 73, 78]. Combinations of sparse and blocked matrices are increasingly under consideration in machine learning [30]. Even complex operators can be expressed as structured tensors. For example, convolution can be expressed as a matrix multiplication with the Toeplitz matrix of all the circular shifts of the filter [88].

Currently, support for structured data is fragmented and incomplete. Experts must hand write variations of even the simplest kernels, like matrix multiply, for each data structure/data set and architecture to get performance. Implementations must choose a small set of features to support well, resulting in a compromise between **program flexibility** and **data structure**

Table 1. Finch supports **both** complex programs and complex data structures.

Control Flow Support	**Halide**	**Taco**	**Cora**	**Taichi**	**Stur**	**Finch**
Einsums/Contractions	✓	✓	✓	✓	✓	✓
Multiple LHS	✓		✓	✓		✓
Affine Indices	✓			✓	✓	✓
Recurrence	✓					
If-Conditions and Masks	✓	✓		✓		✓
Scatter Gather	✓			✓		✓
Early Break		✓		✓		✓
Unrestricted Read/Write	✓					

Structure Support	**Halide**	**Taco**	**Cora**	**Taichi**	**Stur**	**Finch**
Dense	✓	✓	✓	✓	✓	✓
Padded	✓					✓
One Sparse Operand		✓		✓		✓
Multiple Sparse Operands		✓				✓
Run-length						✓
Symmetric					✓	✓
Regular Sparse Blocks		✓				✓
Irregular Sparse Blocks						✓
Ragged			✓			✓

flexibility. Hand-written solutions are collected in diverse libraries like MKL, OpenCV, LAPACK or SciPy [10, 23, 75, 94]. However, libraries will only ever support a subset of programs on a subset of data structure combinations. Even the most advanced libraries, such as GraphBLAS, which support a wide variety of sparse operations over various semi-rings always lack support for other features, such as N-D tensors, fused outputs, or runs of repeated values [24, 68]. While dense tensor compilers support an enormous variety of program constructs like early break and multiple left hand sides, they only support dense tensors [45, 76]. Special-purpose compilers like TACO [57], Taichi [51], StructTensor [42], or CoRa [37] which support a select subset of structured data structures (only sparse, or only ragged tensors) must compromise by greatly constraining the classes of programs which they support, such as tensor contractions. This trade-off is visualized in Table 1.

Prior implementations are incomplete because the abstractions they use are tightly coupled with the specific data structures that they support. For example, TACO merge lattices represent Boolean logic over sets of non-zero values on an integer grid [58]. The polyhedral model allows various compilers to represent dense computations on affine regions [45]. Taichi enriches single static assignment with a specialized instruction to access only a single sparse structure, but it supports more control flow [51]. These systems restrict their scope to avoid the challenges that occur when complex control flow meets structured data. There are two challenges:

Optimizations are specific to the indirection and patterns in data structures: These structures break the simple mapping between tensor elements and where they are stored in memory. For example, sparse tensors store lists of which coordinates are nonzero, whereas run-length-encoded tensors map several pixels to the same color value. These zero regions or repeated regions are optimization opportunities, and we must adapt the program to avoid repetitive work on these regions by referencing the stored structure.

Performance on structured data is highly algorithm dependent: The landscape of implementation decisions is dramatically unpredictable. For example, the asymptotic performance of sparse matrix multiplication can be impacted by the distribution of nonzeros, the sparse format, and the loop order [8, 102]. This means that performance engineering for such kernels requires the exploration of a large design space, changing the algorithm as well as the data structures.

In this work, we propose a new programming language, Finch, which supports *both* flexible control flow and diverse data structures. Finch facilitates a programming model which resolves the challenges of computing over structured tensors by **combining control flow and data structures into a common representation where they can be co-optimized**. In particular, Finch automatically specializes the control flow to the data so that performance engineers can focus on experimenting with many algorithms. Finch supports a familiar programming language of loops, statements, if conditions, breaks, etc., over a wide variety of tensor structures, such as sparsity, run-length-encoding, symmetry, triangles, padding, or blocks. This support would be useless without the appropriate level of structural specialization; Finch reliably utilizes the key properties of structure, such as structural zeros, repeated values, or clustered non-zeros.

As an example, in Figure 2, a programmer might explore different ways to intersect only the even integers of two lists. The control flow here is only useful if the first example differs from the next two in that it actually selects only even indices as the two integer lists are merged and different from the last in that it does not require another tensor:

```
for i = _
    if i % 2 == 0
        c[i]=a[i]*b[i]
```

```
for i = _
    if i % 2 == 0
        ap[i] = a[i]
for i = _
    c[i] = ap[i] * b[i]
```

```
for i = _
    cp[i] = a[i] * b[i]
for i = _
    if i % 2 == 0
        c[i] = cp[i]
```

```
for i = _
   if i % 2 == 0
       f[i] = 1
for i = _
   c[i] = a[i] * b[i] * f[i]
```

Fig. 2. Four strategies to intersect even indices of two lists, represented as sparse vectors with sorted indices.

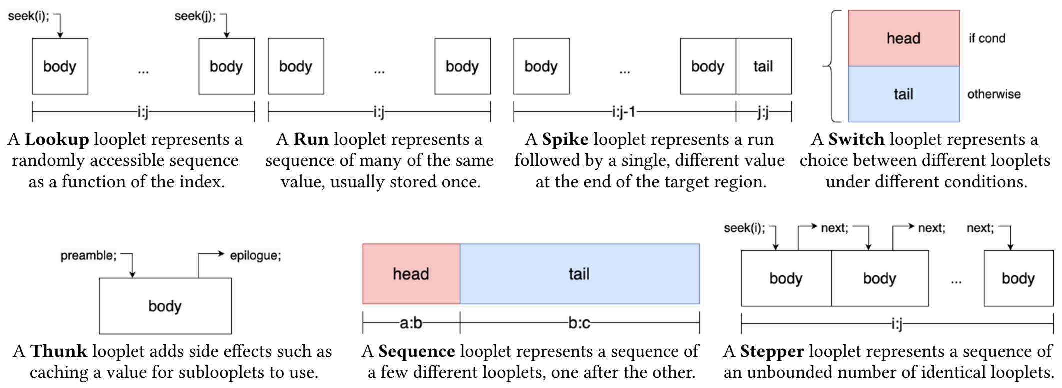
Contributions. We make the following contributions:

- More complex tensor structures than ever before. We are the first to extend level-by-level hierarchical descriptions to capture banded, triangular, run-length-encoded, or sparse datasets, and any combination thereof. We have chosen a set of level formats that completely captures all combinations of relevant structural properties (zeros, repeated values, and/or blocks). Although many systems (TACO, Taichi, SPF, Ebb) [18, 29, 51, 87] feature a flexible structure description, Finch is more capable and extensible because it uses looplets [7] to express the structure of each level.
- A rich sparse and structured tensor programming language with for-loops and complex control flow constructs at the same level of productivity of dense tensors. To our knowledge, the Finch programming language is the first to support if-conditions, early breaks, multiple left hand sides, and complex accesses (such as affine indexing or scatter/gather) over sparse and structured tensors.
- A compiler that specializes programs to data structures automatically, facilitating an expressive language for searching the space of algorithms and data structures. Finch reliably utilizes four key properties of structure: structural zeros, repeated values, clustered non-zeros, and singletons.
- Our compiler is highly extensible, evidenced by the variety of level formats and control flow constructs that we implement in this work. For example, Finch has been extended to support real-valued tensor indices with continuous tensors. Finch is also used as a compiler backend for the Python PyData/Sparse library [3].
- We evaluate the efficiency, flexibility, and expressiveness of our language in several case studies on a wide range of applications, demonstrating speedups over the state of the art in classic operations such as SpMV (geomean 1.26×, max 3.04×) and SpGEMM (geomean 1.30×, max 1.62×), to more complex applications such as graph analytics (geomean 2.47× on Bellman-Ford, reducing lines of code by 4× over GraphBLAS), and image processing (19.5× on the sketches dataset [36]).

2 Background

2.1 Looplets

Finch represents iteration patterns using looplets, a language that decomposes datastructure iterators hierarchically. Looplets represent the control-flow structures needed to iterate over any given datastructure, or multiple datastructures simultaneously. Because looplets are compiled with progressive lowering, structure-specific mathematical optimizations such as integrals, multiply by zero, etc. can be implemented using simple compiler passes like term rewriting and constant propagation during the intermediate lowering stages.



A **Lookup** looplet represents a randomly accessible sequence as a function of the index.

A **Run** looplet represents a sequence of many of the same value, usually stored once.

A **Spike** looplet represents a run followed by a single, different value at the end of the target region.

A **Switch** looplet represents a choice between different looplets under different conditions.

A **Thunk** looplet adds side effects such as caching a value for sublooplets to use.

A **Sequence** looplet represents a sequence of a few different looplets, one after the other.

A **Stepper** looplet represents a sequence of an unbounded number of identical looplets.

Fig. 3. The looplet language, as understood in a correct execution of a Finch program.

The looplets are described in Figure 3. We simplify the presentation to focus on the semantics, rather than precise implementation. For more background on looplets, we recommend the original work [7]. Several looplets introduce or modify variables in the scope of the target language. This allows looplets to lift code to the highest possible loop level. It is assumed that if a looplet introduces a variable, the child looplet will not modify that variable.

Table 2. Detailed descriptions of looplet behavior. An example compilation is given later in Figures 17 and 18

Description	**Arguments**
lookup(body): The Lookup looplet represents a randomly accessible region of an iterator, where the element at index i is given by the expression $body(i)$. While this often an array access, it could also be a computation like $f(i) = \sin(\pi i/7)$. Lookups are leaf looplets, and the body is a value, not a looplet.	• $body(i)$: A function which returns an expression for the value at index i in the current program state.
run(body): The Run looplet represents a constant region of an iterator. Runs are leaf looplets, and the body of a run is a value, not a looplet, similar to a Lookup. Run looplets do not need to store any information about their region because it is specified by the enclosing loop.	• *body*: An expression representing the value within the run in the current program state.
switch(cond, head, tail): The Switch looplet specializes the body of a looplet based on a condition, evaluated in the embedding context. If the condition is true, we use *head*, otherwise *tail*. Switch has a high lowering priority so we can see the looplets it contains and lower them appropriately. Lowering Switch first also lifts the condition as high as possible in the loop nest.	• *cond*: A Boolean-valued expression. • *head*: A looplet to execute if the condition is true. • *tail*: A looplet to execute if the condition is false.
thunk(preamble, body, epilogue): The Thunk looplet allows us to cache certain computations in program state in the *preamble*. The state can be used by the Thunk *body*, making Thunks useful for computing and caching the results of expensive computations. The *epilogue* can be used to clean up any relevant side effects.	• *preamble*: A setup program for *body*. • *body*: A looplet that can reference variables defined in *preamble*. • *epilogue*: A teardown program for *preamble* or *body*.
sequence(bodies...): The Sequence looplet represents the concatenation of two or more looplets. The arguments must be *phase* objects which regions on which each body is defined.	• *bodies...*: One or more phase objects, whose regions must be non-overlapping, covering, and ordered.
phase(ext, body): The Phase object is not a looplet, but instead helpfully couples a sublooplet with the subregion of indices it is defined on in a larger compound looplet.	• *ext*: An expression for the absolute range on which the *body* is defined. • *body*: The looplet defined within *ext*.
spike(body, tail): The Spike looplet represents a run followed by a single value. Spike can be considered a shorthand for $\textbf{sequence}(\textbf{phase}(i : j - 1, \textbf{run}(body)), \textbf{phase}(j : j, \textbf{run}(tail)))$. In the compiler, spikes are handled with special care, since they are can help align the final value to the end of the root loop extent without using any special bounds inference.	• *body*: An expression representing the value within the run. • *tail*: An expression representing the value at the end of the spike.
stepper([seek], next, stride, body): The Stepper looplet represents a variable number of looplets, concatenated. Since our looplets may be skipped over due to conditions or various rewrites, the *seek* function allows us to fast-forward the state to the start of the root loop extent when it comes time to lower the stepper. **jumper(seek, next, body):** The Jumper looplet is identical to a stepper looplet, but when two jumpers interact, the largest stride between them is taken, and the jumper with the smaller stride is demoted to a stepper within that region. Jumpers allow us to request leader-follower strategies or mutual-lookahead coiteration.	• $seek(j)$: A function that returns a program that advances state to the iteration of the stepper which processes the absolute coordinate j. • *next*: A program that advances the state to the next iteration of the stepper. • *stride*: The absolute endpoint of the current subregion of the stepper. • *body*: The looplet to execute for the current iteration of the stepper.

Finch advances the state-of-the-art over the looplets work [7]. While looplets presented a way to merge iterators over single dimensional structures, Finch is the only framework to support such a broad range of multi-dimensional structured data in a programming language with fully-featured control flow. Looplets provide a powerful mechanism to simplify structured loops, but our paper shows how to make this functionality practical; Finch uses looplets as a symbolic loop simplification engine. The precise choice and implementation of tensor level structures, the lifecycle interface between levels and looplets, and the canonicalization of fancy indexing and masking all serve to utilize and recombine looplets to achieve efficient computation over structured tensors.

2.2 Fiber Trees

Fiber-tree style tensor abstractions have been the subject of extensive study [28, 29, 88]. The underlying idea is to represent a multi-dimensional tensor as a nested vector datastructure, where each level of the nesting corresponds to a dimension of the tensor. Thus, a matrix would be represented as a vector of vectors. Fiber-trees can represent sparse tensors by varying the type of vector used at each level in a tree. Thus, a sparse matrix might be represented as a dense vector of sparse vectors. The vector of subtensors in this abstraction is referred to as a **fiber**.

Instead of storing the data for each subfiber separately, most sparse tensor formats such as CSR, DCSR, and COO usually store the data for all fibers in a level contiguously. In this way, we can think of a level as a bulk allocator for fibers. Continuing the analogy, each fiber is disambiguated by a **position**, or an index into the bulk pool of subfibers. The mapping f from indices to subfibers is thus a mapping from an index and a position in a level to a subposition in a sublevel. Figure 4 shows a simple example of a level as a pool of fibers. When we need to refer to a particular fiber at position p in the level l, we may write $fiber(l, p)$. The construction of fibers from levels is lazy, and the data underlying each fiber is managed entirely by the level, so the level may choose to overlap the storage of different fibers. Thus, the only unique data associated with $fiber(l, p)$ is the position p.

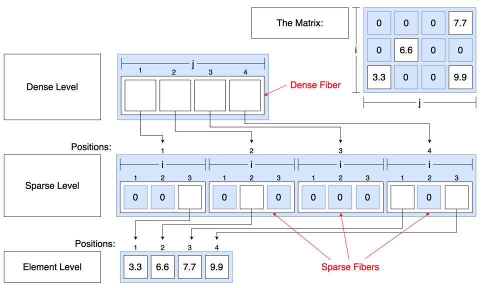


Fig. 4. A fiber tree representation of a sparse matrix in CSC format, with a dense outer level, a sparse inner level, and an element level of leaves.

3 Bridging Looplets and Finch: The Tensor Interface

Tensors use multiple dimensions to organize data with respect to orthogonal concepts. Thus, the Finch language supports multi-dimensional tensors. Unfortunately, the looplet abstraction is best suited towards iterators over a single dimension. Our level abstraction provides a bridge between the single dimensional iterators created from looplets and the multi-dimensional fiber-tree abstractions common to tensor compilers. This bridge must address three challenges. First, while looplets represent an instance of an iterator over a tensor, we may access the same tensor twice with different indices. Thus, the *unfurl* function creates separate looplet nests for each iterator. Next, since Finch programs go beyond just single Einsums, they may read and write to the same data at different times. The *declare*, *freeze*, and *thaw* functions provide machinery to manage transition between these states. Finally, we must be able to write looplet nests that modify tensors, as well as reading them. The *assemble* function manages the allocation of new data in the tensor.

Additionally, prior fiber-tree representations focus on sparsity (where only the nonzero elements are represented) and treat sparse vectors as sets of represented points. Since our fiber-tree representation must handle other kinds of structure, such as diagonal, repeated, or constant values, we must generalize our fiber abstraction to allow arbitrary mappings from indices into a space of subfibers.

In the rest of this section, we discuss how these 5 core functions (*declare*, *freeze*, *thaw*, *unfurl*, and *assemble*) function as part of a life cycle abstraction that defines a level in Finch. These interfaces add to the level abstraction, expanding the types of data that they can express via mapping to looplets and expanding the contexts in which they can be used. We then identify a taxonomy of four key structural properties exhibited in data. We implement several levels in this abstraction that capture all combinations of these structures, including specializations to zero dimensional tensors (scalars) and level structures that support different access patterns.

3.1 Tensor Lifecycle, Declare, Freeze, Thaw, Unfurl

Our simplified view of a level is enabled by our use of looplets to represent the structure within each fiber. In fact, our level interface requires only 5 highly general operations, described below.

The first three of these functions, *declare*, *freeze*, and *thaw*, have to do with managing when tensors can be assumed mutable or immutable. As we use looplets to represent iteration over a tensor, we must restrict the mutability of tensors in the region of code which iterates over them. For example, if a tensor declares it has a constant region from $i = 2 : 5$, but some other part of the computation modifies the tensor at $i = 3$, this would result in incorrect behavior. It is much easier to write correct looplet code if we can assume that the tensor is immutable while it is being read from. Thus, we introduce the notion that a tensor can be in read-only mode or update-only mode. In read-only mode, the tensor may only appear in the right-hand side of assignments. In update-only mode, the tensor may only appear in the left-hand side of an assignment, either being overwritten or incremented by some operator. We can switch between these modes using freeze and thaw functions. The *declare* function is used to allocate a tensor, initialize it to some specified size and value, and leave it in update-only mode.

Table 3. The five functions that define a level.

Description	**Arguments**
declare(*tns*, *init*, *dims*...) : Returns a program that declares a tensor of size *dims* and an initial value of *init*. This procedure thaws the tensor.	• *tns*: The tensor to declare. Must be read-only. • *init*: An expression for the initial value. • *dims*...: Expressions for the tensor dimensions.
freeze(*tns*) : Returns a program that finalizes the updates in the tensor, and readies the tensor for reading.	• *tns*: The tensor to freeze. Must be update-only.
thaw(*tns*) : Returns a program that prepares the level to accept updates, initializing internal scratchspaces, etc.	• *tns*: The tensor object to thaw. Must be read-only.
unfurl(*tns*, *ext*, *mode*) : Returns a looplet that iterates over subtensors within the tensor along the extent *ext*. When *mode* = **read**, returns a looplet nest over the values in the read-only fiber. When *mode* = **update**, returns a looplet nest over mutable subfibers in the update-only fiber. The compiler calls *unfurl* directly before iterating over the corresponding loop, so it has access to any state variables introduced by freezing or thawing the tensor.	• *tns*: The tensor or subtensor to unfurl. • *ext*: An expression representing the range to unfurl over. • *mode*: An enum representing whether to unfurl in read-only or update-only mode.
unwrap(*tns*, *mode*, [*op*], [*rhs*]) : Returns code to read or update the scalar value of a scalar or leaf node *tns* (possibly a fiber), using *op* and *rhs* in the case of update. Parent fibers may ask their children to use this function to set a dirty bit in *tns*, indicating a non-fill value has been written and that the child fiber needs to be stored.	• *tns*: The tensor object to increment. • *mode*: An enum representing whether to unwrap in read-only or update-only mode. • *op*: An expression representing the operation to apply to the scalar value. • *rhs*: An expression for the second argument to *op*.
assemble(*lvl*, pos_{start}, pos_{stop}) : Returns a program that allocates subfibers in the level from positions pos_{start} to pos_{stop}. In looplet nests which modify the output, this function is often called to construct the output tensor. For example, to handle the case where a new nonzero is discovered, the compiler might call *assemble* to obtain a location in memory to which the nonzero may be written.	• *lvl*: The level object in which subfibers are allocated. • pos_{start}: The first subfiber position to assemble. • pos_{stop}: The last subfiber position to assemble.

The *unfurl* function is used to manage iteration over a subfiber. When it comes time to iterate over a tensor, be in on the left or right hand side of an assignment, the compiler calls *unfurl* to return a looplet nest that describes the hierarchical structure of the outermost dimension of the tensor. The compiler calls *unfurl* directly before compiling the corresponding loop, so the called has access to any state variables introduced by freezing or thawing the tensor. Looplets were chosen for this purpose as a symbolic engine to ensure certain simplifications take place, but another symbolic system could have been used (e.g. polyhedral[103] or e-graph search [81]). We chose looplets because they reliably process structured iterators, predictably eliminating zero regions, using faster lookups when available, and utilizing repeated work.

Our view of a level as a fiber allocator implies an allocation function $assemble(tns, pos_{start} : pos_{stop})$, which allocates fibers at positions $pos_{start} : pos_{stop}$ in the level. We don't specify a deallocation function, instead relying on initialization to reset the fiber if it needs to be reused. While all of the previous functions are used to manage the lifecycle and iteration over a general tensor, *assemble* is quite specific to the level abstraction, and the notion of positions within sublevels.

The *assemble* function lends itself particularly to a "vector doubling" allocation approach, which we have found to be effective and flexible when managing the allocation of sparse left hand sides.

3.2 The 4 Key Structures

In the Finch programming model, the programmer relies on the Finch compiler to specialize to the sequential properties of the data. In our experience, the main benefits of specializing to structure come from the following properties of the data:

- **Sparsity** Sparse data is data that is mostly zero, or some other fill value. When we specialize on this data, we can use annihilation ($x * 0 = 0$), identity ($x * 1 = 1$), or other constant propagation properties ($ifelse(false, x, y) = y$) to simplify the computation and avoid redundant work.
- **Blocks** Blocked data is a subset of sparse data where the nonzeros are clustered and occur adjacent to one another. This presents two opportunities: We can avoid storing the locations of the nonzeros individually, and we can use more efficient randomly accessible iterators within the block. [7, 54, 95].
- **Runs** Runs of repeated values may occur in dense or sparse code, cutting down on storage and allowing us to use integration rules such as `for i = 1:n; s += x end` ⟶ `s += n * x` or code motion to lift operations out of loops [7, 34].
- **Singular** When we have only one non-fill region in sparse data, we can avoid a loop entirely and reduce the complexity of iteration [7, 42].

Sparse	**Blocked**	**Runs**	**Singular**	**Format**
				Dense
			✓	n/a
		✓		RunList
		✓	✓	n/a
	✓			n/a
	✓		✓	n/a
	✓	✓		n/a
	✓	✓	✓	n/a
✓				SparseList
✓			✓	SparsePinpoint
✓		✓		SparseRunList
✓		✓	✓	SparseInterval
✓	✓			SparseBlockList
✓	✓		✓	SparseBand
✓	✓	✓		n/a
✓	✓	✓	✓	n/a

Fig. 5. All combinations of our 4 structural properties and the corresponding formats we have chosen to represent them. Not all combinations are relevant. There is no benefit to a block of runs if the run lengths are stored individually. Blocks and singletons only make sense in the context of sparsity.

In the following section, we consider a set of concrete implementations of levels that expose all combinations of these structures, paying some attention to a few important special cases: random access, scalars, and leaf levels. We summarize the structures in Table 4 and Table 5.

3.3 Implementations of Structures

3.3.1 Sequentially Constructed Levels. We consider all combinations of the four structural properties in Table 5, resulting in 8 key level formats which proved to be useful. While it is impossible to write code which precisely addresses every possible structure, our level formats can be combined in a tree to express a wide variety of hierarchical structures, as shown in Figure 6.

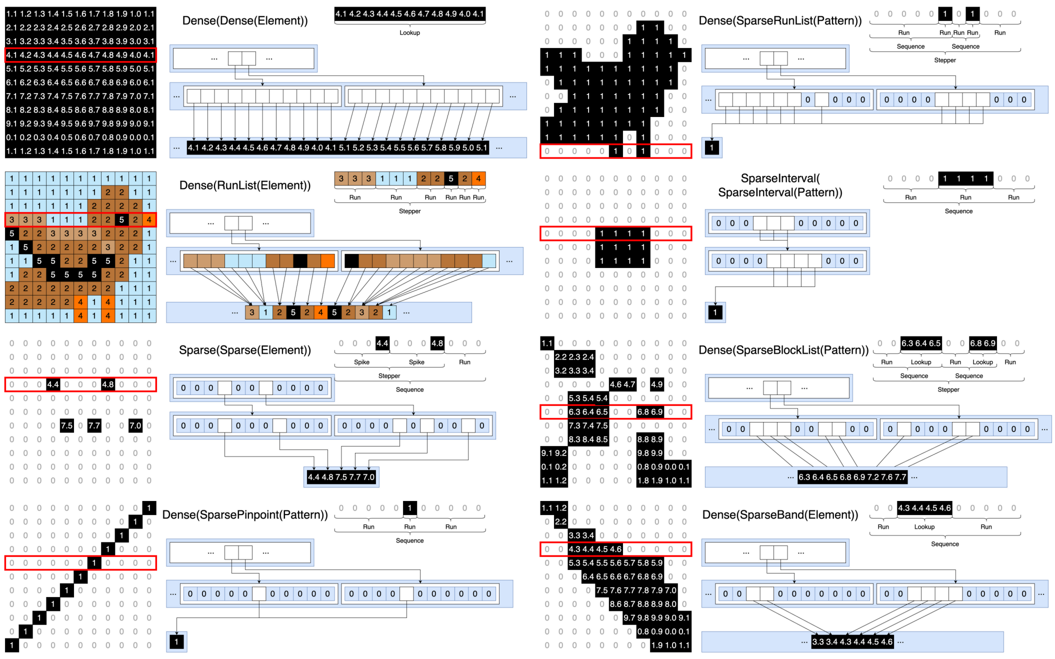


Fig. 6. Several examples of matrix structures represented using the level structures identified in Table 5. Comparing to [7, Figure 3], we have added a level-by-level structural decomposition to the looplets.

3.3.2 *Non-sequentially Constructed Levels.* To reduce the implementation burden and improve efficiency in the common case, our structured levels only support bulk, sequential construction. However, some problems or loop orders require writing out of order. For example, out-of-order access occurs in our SpGEMM and histogram algorithms in Sections 7.2 and 7.4. This requires more complicated datastructures with higher performance overhead, such as hash tables or trees. We therefore support several randomly accessible sparse datastructures. These general sparse datastructures may be used as intermediates to convert to our more specialized structures later.

3.3.3 *Scalars.* Because leaf levels are geared towards representing multiple leaves, we also introduce a much simpler Scalar format to represent 0-dimensional tensors (a single leaf). Scalars don't have as much structure because they only concern one value. However, we allow the programmer to declare that a scalar might be sparse, or that it might be used in a reduction which can be exited early. Scalar structures can interact with other tensor structures in crucial ways.

We introduce the SparseScalar to indicate that it might be equal to the fill value and request for the compiler to simplify subsequent computations accordingly. Constant propagation through tensors is known to be a complex compiler pass [70]. SparseScalars present an alternative by specializing reads for the possible fill value when a runtime check is passed.

```
p = ShortCircuitScalar{0}()
@finch begin
    p .= 0
    for j=_
        p[] *= A[j]
```

Fig. 7. Using a ShortCircuitScalar to find the product of values in A.

We also introduce ShortCircuitScalars, which signal that the compiler should early break out of a reduction loop when the loop hits an annihilator value. For example, Figure 7 computes the product of vector elements, exiting the loop when one of them is zero. Ordinary **break** statements may affect the value of all other statements in the loop, violating our lifecycle constraints. We instead support early break as a structural property as it allows us to elegantly compose with other structures, including other

ShortCircuitScalars which may trigger at different times. SparseScalars and ShortCircuitScalars are novel contributions of this work; other systems don't include them, limiting the impact of sparsity.

3.3.4 Leaf Levels. The leaf level stores the actual entries of the tensor. In most cases, it is sufficient to store each entry at a separate position in a vector. This is accomplished by the **ElementLevel**. However, when all of the values are the same, an additional optimization can be made by storing the identical value only once. In this work, we introduce the concept of a **PatternLevel** to handle this binary case. The PatternLevel has a fill value of *false*, and returning *true* for all "stored" values. The PatternLevel allows us to easily represent unweighted graphs or other Boolean matrices.

4 The Finch Language

```
EXPR := LITERAL|VALUE|INDEX|VARIABLE|EXTENT|CALL|ACCESS
STMT := ASSIGN|LOOP|DEFINE|SIEVE|BLOCK|DECLARE|FREEZE|THAW

 DECLARE := TENSOR .= EXPR(EXPR...)   #V is the set of all values
  FREEZE := @freeze(TENSOR)           #S is the set of all Symbols
    THAW := @thaw(TENSOR)             #T is the set of all types
  TENSOR := TENSORNAME :: WRAPPER(TENSOR, EXPR...)
  ASSIGN := ACCESS <<EXPR>>= EXPR        TENSORNAME := S
    LOOP := for INDEX = EXPR                LITERAL := V
              STMT                            VALUE := S::T
            end                                 WRAPPER := S
  DEFINE := let VARIABLE = EXPR                 INDEX := S
              STMT                           VARIABLE := S
            end                                EXTENT := EXPR : EXPR
   SIEVE := if EXPR                              CALL := EXPR(EXPR...)
             STMT                            ACCESS := TENSOR[EXPR...]
            end                                   MODE := @mode(TENSOR)
   BLOCK := begin
              STMT...
            end
```

(a) The syntax of the Finch language. Compare this grammar to the Concrete Index Notation of TACO [57, Figure 3], noting the addition of multiple left-hand sides (via blocks), access with arbitrary expressions, and explicit declaration, as well as freeze and thaw.

$$[\![\mathbf{loop}(i,\mathbf{extent}(a,b),\mathbf{block})]\!]^F = \cup^F_{iv \in \mathbb{Z} \cap [[\![a]\!]^F, [\![b]\!]^F]} [\![\mathbf{block}]\!]^{F, i \mapsto iv}$$

$$[\![\mathbf{access}(\mathit{tensor}, \mathit{exprs}...)]\!]^F = [\![\mathit{tensor}]\!]^F([\![\mathit{exprs}]\!]^F ...)$$

$$[\![\mathit{tensorname}]\!]^F = F(\mathit{tensorname})$$

$$[\![\mathit{wrapper}(\mathit{tensor}, \mathit{exprs})]\!]^F = ([\![\mathit{wrapper}]\!]^W(\mathit{exprs}))([\![\mathit{tensor}]\!]^F)$$

$$[\![\mathbf{block}(\mathit{stmt}_1, \mathit{stmts}...)]\!]^F = [\![\mathit{stmt}_1]\!]^F \cup^F [\![\mathbf{block}(\mathit{stmts}...)]\!]^F$$

$$[\![\mathbf{block}()]\!]^F = \{\}$$

$$[\![\mathbf{sieve}(\mathit{expr}, \mathit{stmt})]\!]^F = \begin{cases} [\![\mathit{stmt}]\!]^F & [\![\mathit{expr}]\!]^F \\ \{\} & \end{cases}$$

$$[\![\mathbf{declare}(\mathit{var}, \mathit{expr}, \mathit{stmt})]\!]^F = [\![\mathit{stmt}]\!]^{F, \mathit{var} \mapsto [\![\mathit{expr}]\!]^F}$$

$$[\![\mathbf{assign}(\mathbf{access}(\mathit{tensor}, \mathit{idxExpr}), \mathit{op}, \mathit{expr})]\!]^F = F \cup^F \{\mathit{tensor} \mapsto$$
$$[\![\mathit{tensor}]\!]^F \cup \{[\![\mathit{idxExprs}]\!]^F ... \mapsto [\![\mathit{op}]\!]^F([\![\mathit{tensor}]\!]^F([\![\mathit{idxExprs}]\!]^F ...), [\![\mathit{expr}]\!]^F)\}\}$$

(b) Semantics of Finch. The domain F assigns tensor names to functions ($\mathbb{Z}^N \to V$) and W assigns wrappers to functions ($V^M \mapsto ((\mathbb{Z}^N \mapsto V) \mapsto \mathbb{Z}^{N'} \mapsto V)$), representing transformations of tensors. Dimensions are computed via the rules laid out in Section 4.

Fig. 8. Syntax and Semantics for Finch

The syntax of Finch is displayed in Figure 8a, and a denotational semantics is displayed in Figure 8b. The Finch language mirrors most imperative languages such as C with for-loops and control flow. Notable statements that have been added to the language include **`for`**, **`let`**, blocks of code with **`if`**, wrappers of tensors, and the lifecycle functions that let us declare, freeze, and thaw tensors. As discussed in Section 3.3, our language handles **`break`** as a structural property of scalars.

The denotational semantics of our language concern large dense iteration spaces, but the implementation eliminates many of these unnecessary iterations through aggressive optimizations, carefully using life cycles, dimensions, sparsity via looplets, and control flow as a form of sparsity. Section 5 details the specifics of how we compile our syntax to efficient code over structured data.

As detailed in the previous section, tensors are defined externally via an interface that supports the *declare*, *freeze*, *thaw*, and *unfurl* functions. The first three are supported directly in the syntax whereas the fourth will be introduced through evaluation of loops and accesses, in the next section. We do not intend the user to insert freeze or thaw manually, but we include them in the language since they are added by a compiler pass (described in Section 5). Tensors can only change between read and update mode in the scope in which they were defined, so we can insert freeze/thaw automatically by checking whether the tensor is being read or written to in each child scope. We error if a tensor appears on both the left hand and right hand sides within the same child scope.

Table 4. The main level formats supported by Finch. Note that all non-leaf levels store a the dimension of the subfibers and a child level. Since we must be able to handle the case where a sublevel is not stored because a parent level is sparse, all of Finch's sparse formats use a dirty bit during writing to determine whether the sublevel has been modified from it's default fill value and thus, whether it needs to be stored.

Sequentially Constructed Levels
Dense: The dense format is the simplest format, mapping $\mathit{fiber}(l, p)[i] \rightarrow \mathit{fiber}(l.\mathit{lvl}, p \times l.\mathit{shape} + i)$. This format is used to store dense data and is often a convenient format for the root level of a tensor. Due to its simplicity, freezing and thawing the level are no-ops.
RunList: Used to represent runs of repeated values, storing two vectors, *right* and *ptr*, with q^{th} run in the p^{th} subfiber starting and ending at $\mathit{right}[\mathit{ptr}[p] + q]$ and $\mathit{right}[\mathit{ptr}[p] + q + 1] - 1$, respectively. A challenge arises for this level: it is difficult to merge duplicate runs. An example would be merging runs of subfibers of length 3, representing colors in an image. Ideally, we would be able to detect duplicate subfibers and merge them on the fly, but we cannot determine which subfibers are equal because the sublevel cannot be read in update-only mode. Instead, the duplicates are merged during the freeze phase. The compiler *freeze*s the sublevel, *declare*s a separate sublevel *buf* as a buffer to store the deduplicated subfibers, and compares neighboring subfibers in the main level, copying deduplicated subfibers to the buffer.
SparseList: The simplest sparse format, used to construct popular formats like CSR, CSC, DCSR, DCSC, and CSF. It stores two vectors, *idx* and *ptr*, such that $\mathit{idx}[\mathit{ptr}[p] + q]$ is the index of the q^{th} nonzero in the subfiber at position p.
SparsePinpoint: Similar to SparseList, but only one nonzero in each subfiber, eliminating the need for the *ptr* field. It stores a vector *idx*, such that $\mathit{idx}[p]$ is the nonzero index in the subfiber at position p.
SparseRunList: Similar to RunList level, but because runs are sparse, we must also store the start of each run. It stores three vectors *left*, *right*, and *ptr*, such that the q^{th} run in the p^{th} subfiber begins and ends at $\mathit{left}[\mathit{ptr}[p]+q]$ and $\mathit{right}[\mathit{ptr}[p]+q]$, respectively. Like RunList, it also stores a duplicate sublevel, *buf*, for deduplication.
SparseInterval: Similar to SparseRunList, but only stores one run per subfiber, eliminating the need for the *ptr* field. This level does not deduplicate as it cannot store intermediate results with more than one run. It stores two vectors, such that the run in subfiber p begins and ends at $\mathit{left}[p]$ and $\mathit{right}[p]$ respectively.
SparseBlockList: Used to represent blocked data. It stores three vectors, *idx*, *ptr*, and *ofs*, such that $\mathit{ofs}[\mathit{ptr}[p]+q] : \mathit{ofs}[\mathit{ptr}[p]+q+1] - 1$ are the subpositions of block q ending at index $\mathit{idx}[\mathit{ptr}[p] + q]$ in the subfiber at position p.
SparseBand: Similar to SparseBlockList, but stores only one block per subfiber, eliminating the need for the *ptr* field. It stores two vectors *idx* and *ofs*, such that $\mathit{ofs}[p] : \mathit{ofs}[p + 1] - 1$ are the subpositions of the block ending at $\mathit{idx}[p]$ in subfiber p. Banded tensors are a superset of ragged tensors, where every band starts in the first column. In practice, the overhead of storing a 1 for the start of each band is minimal.
Nonsequentially Constructed Levels
SparseHash: The sparse hash format uses a hash table to store the locations of nonzeros, and sorts the unique indices for iteration during the freeze phase. This allows for efficient random access, but not incremental construction, as the freeze phase runs in time proportional to the number of nonzeros in the entire level. It stores two vectors, *idx* and *ptr*, such that $\mathit{idx}[\mathit{ptr}[p] + q]$ is the index of the q^{th} nonzero in the subfiber at position p. Also stores a hash table *tbl* for construction and random access in the level.
SparseBytemap The SparseBytemap format uses a bytemap to store which locations have been written to. Unlike the SparseHash format, the bytemap assembles the entire space of possible subfibers. This accelerates random access in the format, but requires a high memory overhead. Because we don't want to reallocate all of the memory in each iteration, the declaration of this format instead re-assembles only the dirty locations in the tensor. This format is analogous to the default workspace format used by TACO. It stores two vectors, *idx* and *ptr*, such that $\mathit{idx}[\mathit{ptr}[p] + q]$ is the index of the q^{th} nonzero in the subfiber at position p. These vectors are used to collect dirty locations. It also stores *tbl*, a dense array of Booleans such that $\mathit{tbl}[\mathit{shape} * p + i]$ is true when there is a nonzero at index i in the subfiber at position p.
Leaf Levels
Element: The element level uses an array *val* to store a value for each position p. The zero (fill) value is configurable.
Pattern: The pattern level statically represents a leaf level with a fill value of *false* and whose stored values are all *true*.
Scalars
Scalar: A dense scalar that, unlike a variable, supports reduction.
SparseScalar: A scalar with a dirty bit which specializes on the fill value when it occurs.
ShortCircuitScalar: A scalar which triggers early breaks in reductions whenever an annihilator is encountered. ShortCircuitScalars trigger stepper and lookup looplets to re-specialize the loop whenever a reduction into the scalar hits an annihilator, removing the reduction since the value can no longer change. Short-circuiting conditions are lowered by inserting a branch into the loop body which checks for the short circuit condition. The branch contains the (hopefully simplified) remainder of the loop, followed by a `break`. Re-specialization of other looplets is not required because only steppers and lookups have more than a constant number of iterations.

Our expressions support a wide variety of scalar operations on literals, indices, extents, wrappers, and calls to externally defined functions. Wrapper tensors are static higher order functions on tensors that serve to optimize indexing logic such as `i + j` or `i <= j`; an initial pass converts indexing to wrappers when possible. We implement wrappers as transformations on looplets or other properties of the tensor interface. This supports a more efficient, lazy implementation of complex indexing as opposed to naive random access. For examples of wrappers, see Table 5.

Our syntax is highly permissive: by allowing blocks of code with multiple statements, we implicitly support many features gained through complicated scheduling commands in other frameworks, such as multiple outputs, masking to avoid work, temporary tensors, and arbitrary loop fusion and nesting. These features are seen in our implementation of Gustavson's sparse-sparse matrix multiply, which writes to a temporary tensor in an inner loop and then reuses it; or in our breadth-first search, which uses an `if` statement to avoid operating on vertices outside the frontier.

Dimensionalization Rules. Looplets typically require the dimension of the loop extent to match the dimensions of the tensor. However, it is cumbersome to write the dimensions in loop programs, and most tensor compilers have a means of specifying the dimensions automatically. In many pure Einsum languages like TACO, determining dimensions is not needed because any tensor dimensions that share an index are assumed to be the same [58]. Other languages, such as Halide, perform bounds inference where known bounds are symbolically propagated to fill in unknown bounds, often from output/input sizes to intermediates via some approximation such as interval analysis or polyhedral methods [45, 76]. We refer to the process of discovering suitable dimensions as **dimensionalization**. Loop bounds in Finch are computed automatically via a few simple rules. There are currently two kinds of dimensions in Finch: _ represents a dimensionless quantity, and `a:b` represents an integer dimension. Dimensions can be joined with the `meet` operation, which returns the dimension that is not _ or else asserts that the two extents match.

- The dimension of an index is defined as the `meet` of the loop bound and the tensor dimension corresponding to any right-hand-side accesses with that index.
- The n^{th} dimension of a tensor declaration is defined as the `meet` of all index dimensions in the n^{th} mode of left-hand-side accesses to that tensor, from its declaration to its first read.
- The dimension of `i + c`, where `c` is a constant, is the dimension of `i` shifted by `c`.
- The dimension of `~(x)`, or any other unrecognized function, is _.
- More rules may be added as Finch is extended to recognize more indexing syntax.

5 The Finch Compiler

The Finch compiler takes a Finch program together with a program state defining the formats of tensors, and produces efficient structure aware code. The compiler operates in several stages. The first stages normalize the program to make it easier to process. The final stages lower a normalized program recursively, one loop at a time. For each loop, all tensors that are indexed by the loop index are transformed into looplets based on their structure, and these looplets are lowered to executable code. The overall flow is summarized in Figure 9.

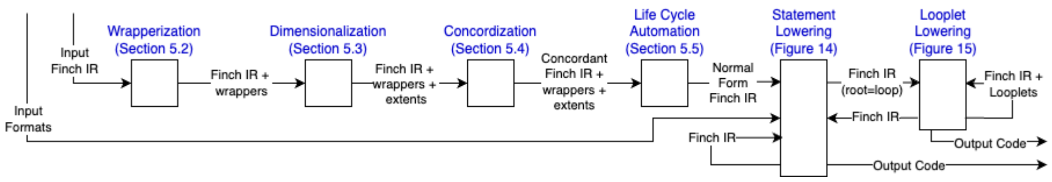

Input Finch IR
Input Formats
Wrapperization (Section 5.2)
Finch IR + wrappers
Dimensionalization (Section 5.3)
Finch IR + wrappers + extents
Concordization (Section 5.4)
Concordant Finch IR + wrappers + extents
Life Cycle Automation (Section 5.5)
Normal Form Finch IR
Statement Lowering (Figure 14)
Finch IR (root=loop)
Finch IR
Finch IR
Looplet Lowering (Figure 15)
Finch IR + Looplets
Output Code
Output Code

Fig. 9. Stages of the Finch Compiler.

5.1 Finch Normal Form

Our core recursive lowering compiler described in Figure 14 and Figure 15 is designed to handle a particular class of programs we refer to as **Finch Normal Form**. This section defines the properties of Finch Normal Form. Later sections will describe how to normalize all Finch programs which are well-defined under the semantics in Figure 8b. The properties of our normal form are as follows:

- **Access with Indices:** Though Finch allows general expressions (including affine expressions and general function calls) in an access (i.e. `A[i + j]` or `A[I[i]]`), the normal form restricts to allow only indices in accesses (i.e. `A[i]`), rather than more general expressions.
- **Evaluable Dimensions:** Loop dimensions and declaration dimensions must be evaluable at the time we compile them, so we restrict the dimensions in normal form to dimensions that are extents with limits defined in the scope of the corresponding loop or declaration statement.
- **Concordant:** Finch is column-major by default to match Fortran[11] and Julia[19]. A Finch program is **concordant** when the order of indices in each access match the order in which loops are nested around it. For example, `for j = _; for i = _; s[] += A[i, j] end end` is concordant but `for i = _; for j = _; s[] += A[i, j] end end` is not.
- **Lifecycle Constraints:** Tensors in read mode may appear on the right hand side only. Tensors in update mode may appear on the left hand side only. Tensors may only change modes in the same scopes in which they were defined, imposing a simplifying dataflow constraint.

The following compiler passes explain how programs that violate each of these constraints can be rewritten to programs that satisfy them and thus how we can support such a wide variety of programs. For example, we can write nonconcordant programs like `for i = _; for j = _; s[] += A[i, j] end end` by inserting a loop to randomly access `A`.

Table 5. Wrapper tensors

OffsetTensor shifts tensors such that `offset(tns, delta...)[i...] == tns[i + delta...]`. The shifting is achieved by modifying the ranges returned by the looplets in the wrapped tensor. `A[i..., j + c, k...] -> OffsetTensor(A, (0..., c, 0...))[i..., j, k...]`	
ToeplitzTensor adds a dimension that shifts another dimension of the original tensor. The added dimensions are produced during a call to `Unfurl`, when a lookup looplet is emitted for the first dimension. `A[i_1, ..., i_n, j + k, l...] -> ToeplitzTensor(A, n)[i_1, ..., i_n, j, k, l...]`	
PermissiveTensor allows for out-of-bounds access or padding. Permissive indices have dimension `_`. The out-of-bounds value is `missing`, and the `coalesce` function can be used to return the first nonmissing value. `A[i..., ~j, k...] -> PermissiveTensor(A, (false..., true, false...))[i..., j, k...]`	
ProtocolizedTensor allows for advanced iteration protocols. The ProtocolizedTensor selects between several different implementations of `unfurl` that a level may support. Finch recognizes several protocols: • The `follow` protocol indicates the structure should be ignored and random access used for each element. • The `walk` protocol declares that the structure of the iterator should be used in the computation. • The `gallop` protocol declares that the structure of a tensor should lead an iteration and the compiler should specialize to that structure with a higher priority than others. A galloping protocol over two SparseList levels produces a mutual-binary-search merge algorithm popularized in the case of worst-case-optimal join queries [14, 72, 93]. `A[i..., p(j), k...] -> ProtocolizedTensor(A, (nothing..., p, nothing...))[i..., j, k...]`	
SwizzleTensor is a lazily transposed tensor that changes the interpretation of the order of modes in the tensor. Unlike other wrappers, a SwizzleTensor is compiled during the wrapperization pass rather than introduced by it. `swizzle(A, perm)[idx...] -> A[idx[perm]...]`	

UpTriMask is a mask tensor that represents Boolean upper triangular matrices. We introduce the mask via rewrite rules, taking care to emit an expression in column-major order. For example,
`i <= j ->  UpTriMask()[i, j]`
`i > j ->  !UpTriMask()[i, j]`
`i <= j ->  !UpTriMask()[j, i-1]`
	`unfurl(UpTriMask(), ext, reader) =`
	`  Lookup(body(j) = UpTriMaskCol(j))`
	`unfurl(UpTriMaskCol(j), ext, reader) =`
	`  Sequence(`
	`    Phase(stop = j, body = Run(true)),`
	`    Phase(body = Run(false)))`

DiagMask is a mask tensor that represents Boolean diagonal matrices. It is introduced via rewrite rules such as:
`i == j -> DiagMask()[i, j]`
`i != j -> !DiagMask()[i, j]`
When `i` would be bound at a higher loop depth than `j`, care is taken to reverse the loop order and emit the mask in column-major order.
	`unfurl(DiagMask(), ext, reader) =`
	`  Lookup(body(j) = DiagMaskCol(j))`
	`unfurl(DiagMaskCol(j), ext, reader) =`
	`  Sequence(`
	`    Phase(stop = j-1, body = Run(false)),`
	`    Phase(stop = j, body = Run(true)),`
	`    Phase(body = Run(false)))`

5.2 Wrapperization

Many fancy operations on indices can be resolved by introducing equivalent **wrapper tensors** which modify the behavior of the tensors they wrap, or by introducing **mask tensors** which replace index expressions like `i <= j` with their equivalent masks (in this case, a triangular mask tensor). Wrappers and masks are summarized in Table 5. More formally, a **wrapper tensor** is any tensor that wraps a tensor variable in an access, and can overload the behavior of *unfurl*, *unwrap*, and *size*, as well as modify the ranges declared by any looplets the wrapper contains. For example, the `offset` wrapper tensor shifts the declared ranges of looplets to shift the tensor with respect to the loop index. Wrappers may also trigger a rewrite (such as a transpose) during wrapperization.

```
for i=_, j=_
  if i <= j
    s[] += A[i - 1, j]
↓
for i=_, j=_
  if UpTriMask()[i, j]
    s[] += OffsetTensor(A, (-1, 0))[i, j]
↓
for i = 1:n
  for j = 1:i
    s[] += A.val[(i - 1) + j * n]
```

Fig. 10. Wrapperization. While `i <= j` is only an expression, `UpTriMask()[i, j]` uses looplets to restrict iteration to `1:i`.

A **mask tensor** is a Boolean tensor with implicit structure that uses a predefined looplet nest, rather than the level abstraction. For example, the `UpTriMask` tensor uses looplets to represent the structure of a Boolean upper triangular matrix. Mask tensors are implemented using static looplets that are constructed during the unfurl step. Mask tensors allow us to lift computations with masks to the level of the loop, without modifying the loop directly.

5.3 Dimensionalization

In Section 4, we described a simple set of rules to calculate dimensions. We implement these rules in a straightforward algorithm to assign dimensions to loops and declaration statements (output tensors). Finch determines the dimension of a loop index i from all of the tensors using i in an access, as well as the bounds in the loop itself, and operates similarly for declarations. Finch can compute these dimensions in a single pass over the program. When the compiler reaches a **read** access, the dimensions of the tensor must be constant and are used to compute the loop index dimension. When the compiler reaches an **update** access, we make record of the indices used for later. Because **freeze** and must occur outside of loops which access a tensor, when we reach a **freeze** we can use those recorded indices to compute the dimensions of the corresponding **declare**.

```
#A is 3 x 4, B is 4 x 5
C .= 0
for i = 1:3
  for j = _
    for k = _
      C[i, j] += A[i, k] * B[k, j]
↓
C .= 0
for i = 1:3
  for j = 1:5
    for k = 1:4
      C[i, j] += A[i, k] * B[k, j]
```

Fig. 11. Dimensionalization

For example, in Figure 11, the second dimension of `A` must match the first dimension of `B`. The first dimension of A must match the `i` loop dimension, `1:3`. Finch will resize declared tensors to match indices used in writes, so `C` is resized to `1:3 x 1:5`. If no dimensions are specified elsewhere, Finch will use the dimension of the declared tensor. Dimensionalization occurs after wrappers are de-sugared, so wrappers can be used to modify dimensions with indexing expressions. Users can exempt an index from dimensionalization by wrapping it in ~ to produce a "PermissiveTensor"

5.4 Concordization

After dimensionalization, Finch runs a pass over the code to make the program concordant by inserting single-iteration loops. Examples are given in Figure 12. The algorithm targets each indexing expression `x` which is not an index or is not bound before subsequent indices in the access. The algorithm replaces `x` with a new index `j`, and inserts a loop `for j = x:x` at the appropriate level in the loop nest to make the expression column-

```
                            for i = _
for i = _                     for j = _
  for j = _          →          for k = i:i
    s[] += A[i, j]                s[] += A[k, j]

                            for i = _
  for i = _          →        for j = I[i]:I[i]
    A[I[i]] += 1                A[j] += 1
```

Fig. 12. Concordization to col-major

major. An index expression is considered bound when all of its constituent expressions are defined, either by a for-loop or an earlier definition, or it may be a constant.

5.5 Life Cycle Automation

The last normalization pass inserts the `@freeze` or `@thaw` statements automatically. Tensors are only allowed to change mode within the scope in which they were declared. If they have not been inserted already, this pass automatically inserts these statements in the program, easing the programmer's burden and bridging between structured and dense languages. The pass walks the program and tracks the current mode of each tensor, depending on whether the tensor is read or updated in each statement within the tensor's declared scope block.

```
y .= 0
for i = _
    y[i] = x[i] + 1
for i = _
    x[i] += 1
    y[i] += 1
for i = _
    x[i] += y[i]
```

→

```
y .= 0
for i = _
    y[i] = x[i] + 1
@thaw(x)
for i = _
    x[i] += 1
    y[i] += 1
@freeze(y)
for i = _
    x[i] += y[i]
@freeze(x)
```

Fig. 13. Life cycle automation.

5.6 Recursive Lowering

Finally, after normalization, the program is lowered recursively, node by node. This phase is presented as a staged execution of a small step operational semantics (SOS) for Finch Normal Norm programs. Figure 14 evolves Finch control flow towards loops. Figure 15 lowers loops with looplets.

Though are semantics are phrased as an interpreter, we stress that what goes into the compiler is a program and some formats, and what comes out is code. In Figure 8b, we offer a denotational

$$\frac{\langle val, (e,t,d)\rangle \longrightarrow val' \qquad var \notin d}{\langle \mathbf{define}(var, val, body), (e,t,d)\rangle \longrightarrow \langle body, (e[var \mapsto val'], t, \{\})\rangle}\,\textit{Define}$$

$$\frac{}{\langle \mathbf{literal}(val), (e,t,d)\rangle \longrightarrow val}\,\textit{Literal}$$

$$\frac{}{\langle \mathbf{variable}(name), (e,t,d)\rangle \longrightarrow e(\mathbf{variable}(name))}\,\textit{Variable}$$

$$\frac{\langle args_i, (e,t)\rangle \Rightarrow vals_i \qquad \langle f, (e,t)\rangle \Rightarrow g}{\langle \mathbf{call}(f, args...), (e,t)\rangle \longrightarrow \langle\!\langle g(vals...), t\rangle\!\rangle}\,\textit{Call}$$

$$\frac{}{\langle \mathbf{index}(name), (e,t,d)\rangle \longrightarrow e(\mathbf{index}(name))}\,\textit{Index}$$

$$\frac{\langle node, algebra\rangle \longrightarrow node'}{\langle E[node], s\rangle \longrightarrow \langle E[node'], s\rangle}\,\textit{Simplify}$$

$$\frac{\langle body, s\rangle \longrightarrow s'}{\langle \mathbf{block}(body, tail...), s\rangle \longrightarrow \langle \mathbf{block}(tail...), s'\rangle}\,\textit{Block}$$

$$\frac{\langle cond, (e,t,d)\rangle \Rightarrow true}{\langle \mathbf{sieve}(cond, body), (e,t,d)\rangle \longrightarrow \langle body, (e,t,\{\})\rangle}\,\textit{SieveTrue}$$

$$\frac{e(tns) \mapsto tns' \qquad e(\mathbf{mode}(tns)) \mapsto \mathbf{read} \qquad \langle\!\langle unwrap(tns', \mathbf{read}), t\rangle\!\rangle \longrightarrow tns''}{\langle E[\mathbf{access}(tns)], s\rangle \longrightarrow \langle E[tns''], s\rangle}\,\textit{Access}$$

$$\frac{\langle cond, s\rangle \Rightarrow false}{\langle \mathbf{sieve}(cond, body), s\rangle \longrightarrow s}\,\textit{SieveFalse}$$

$$\frac{\begin{array}{c} e(tns) = tns' \qquad \langle op, (e,t,d)\rangle \longrightarrow op' \qquad \langle rhs, (e,t,d)\rangle \longrightarrow rhs' \\ e(\mathbf{mode}(tns)) = \mathbf{update} \qquad \langle\!\langle unwrap(tns', \mathbf{update}, op', rhs'), t\rangle\!\rangle \longrightarrow t' \end{array}}{\langle E[\mathbf{assign}(\mathbf{access}(tns), op, rhs)], (e,t,d)\rangle \longrightarrow (e, t', d)}\,\textit{Assign}$$

$$\frac{}{\langle \mathbf{value}(ex, type), (e,t,d)\rangle \Rightarrow \langle\!\langle ex, t\rangle\!\rangle}\,\textit{Value}$$

$$\frac{\begin{array}{c} s = (e,t,d) \qquad tns \notin d \qquad e(tns) = tns' \\ \langle init, s\rangle \Rightarrow init' \qquad \forall i \langle init, dims_i\rangle \Rightarrow dims'_i \qquad \langle\!\langle declare(tns', init', dims'...), t\rangle\!\rangle \longrightarrow t' \end{array}}{\langle \mathbf{declare}(tns, init, dims), s\rangle \longrightarrow (e[\mathbf{mode}(tns) \mapsto \mathbf{update}], t', d \cup \{tns\})}\,\textit{Declare}$$

$$\frac{s = (e,t,d) \qquad e(\mathbf{mode}(tns)) = \mathbf{update} \qquad tns \in d \qquad e(tns) = tns'}{\langle \mathbf{freeze}(tns), s\rangle \longrightarrow (e[\mathbf{mode}(tns) \mapsto \mathbf{read}], \langle\!\langle freeze(tns'), t\rangle\!\rangle, d)}\,\textit{Freeze}$$

$$\frac{s = (e,t,d) \qquad e(\mathbf{mode}(tns)) = \mathbf{read} \qquad tns \in d \qquad e(tns) = tns'}{\langle \mathbf{thaw}(tns), s\rangle \longrightarrow (e[\mathbf{mode}(tns) \mapsto \mathbf{update}], \langle\!\langle thaw(tns'), t\rangle\!\rangle, d)}\,\textit{Thaw}$$

Fig. 14. Basic evaluation semantics, roughly defining most of these language constructs to function similarly to their classical definitions. The state, s, of the program is a tuple (e, t, d) of a variable value environment, another state t corresponding to the state in the host language, and finally the set of tensors defined within the current scope, d. We evolve tensor state with $\langle\rangle$ and host state with $\langle\!\langle\rangle\!\rangle$. Several looplets introduce variables into the host state, which may be read when evaluating the **value** node. Lifecycle functions are designed to be implemented and executed in the host language, but these semantics enforce that each function may update state in the host language and flip the mode of the tensor between **read** and **update**.

semantics which described the format-agnostic mathematical behavior of Finch programs as if tensors were functions. In Figures 14 and 15, we offer a structural operational semantics which succinctly describes the format-specific behavior of a hypothetical Finch interpreter. Our semantics can formally answer questions such as "which expressions will be annihilated by zero?", or "how many steps would be required to traverse a certain combination of formats?".

Our evaluation rules in SOS are closely related to the lowering rules used to define a compiler. Lowering rules would be similar to Figures 13 and 14, with a few key differences. First, any changes

$$T := \mathsf{EXPR} \mid \mathsf{STMT}$$

$$\begin{aligned} E :=& [\cdot] \mid \mathbf{loop}(T, T, E) \mid \mathbf{block}(E, T\ldots) \mid \mathbf{block}(T, E, T\ldots) \mid \mathbf{sieve}(E, T) \mid \mathbf{assign}(E, T, T) \mid \mathbf{assign}(T, T, E) \mid \\ & \mathbf{declare}(T, E, T) \mid \mathbf{declare}(T, T, E) \mid \mathbf{call}(E, T\ldots) \mid \mathbf{call}(T, E, T\ldots) \mid \mathbf{access}(T, E, T\ldots) \mid \mathbf{access}(T, T, E\ldots) \end{aligned}$$

$$\frac{e(tns) \mapsto tns' \qquad e(\mathbf{mode}(tns)) \mapsto m \qquad \langle\!\langle unfurl(tns', ext, m), t \rangle\!\rangle \Rightarrow tns''}{\langle \mathbf{loop}(i, ext, E[\mathbf{access}(tns, j\ldots, i)]), s\rangle \to \langle \mathbf{loop}(i, ext, E[\mathbf{access}(tns'', j\ldots, i)]), s\rangle}\,\textit{Unfurl}$$

$$\frac{}{\begin{aligned}&\langle \mathbf{loop}(i, ext, E[\mathbf{access}(\mathbf{run}(body), j\ldots, i)]), s\rangle \\ &\quad \to \langle \mathbf{loop}(i, ext, E[\mathbf{access}(body, j\ldots)]), s\rangle\end{aligned}}\,\textit{Run}$$

$$\frac{e(i) = i' \qquad \langle\!\langle seek(i'), t \rangle\!\rangle \to t'}{\begin{aligned}&\langle E[\mathbf{access}(\mathbf{lookup}(seek, body), j\ldots, i)], (e, t, d)\rangle \\ &\quad \to \langle E[\mathbf{access}(body, j\ldots)], (e, t', d)\rangle\end{aligned}}\,\textit{Lookup}$$

$$\frac{}{\begin{aligned}&\langle \mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{assign}(\mathbf{access}(\mathbf{run}(body), j\ldots, i), op, rhs)]), s\rangle \\ &\quad \to \langle \mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{sieve}(i == a, \mathbf{assign}(\mathbf{access}(body, j\ldots), op, rhs))]), s\rangle\end{aligned}}\,\textit{AcceptRun}$$

$$\frac{\langle\!\langle cond, t \rangle\!\rangle \Rightarrow true}{\begin{aligned}&\langle E[\mathbf{access}(switch(cond, head, tail), i\ldots)], s\rangle \\ &\quad \to \langle E[\mathbf{access}(head, i\ldots)], s\rangle\end{aligned}}\,\textit{SwitchTrue}$$

$$\frac{\langle\!\langle cond, t \rangle\!\rangle \Rightarrow false}{\begin{aligned}&\langle E[\mathbf{access}(switch(cond, head, tail), i\ldots)], s\rangle \\ &\quad \to \langle E[\mathbf{access}(tail, i\ldots)], s\rangle\end{aligned}}\,\textit{SwitchFalse}$$

$$\frac{}{\begin{aligned}&\langle \mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{access}(\mathbf{phase}(\mathbf{extent}(c, d), body), j\ldots, i)]), s\rangle \\ &\quad \to \langle \mathbf{loop}(i, \mathbf{extent}(max(a, c), min(b, d)), E[\mathbf{access}(body, j\ldots, i)]), s\rangle\end{aligned}}\,\textit{Phase}$$

$$\frac{\langle\!\langle preamble, t \rangle\!\rangle \to t' \qquad \langle E[body], (e, t', d)\rangle \to (e', t'', d) \qquad \langle\!\langle epilogue, t'' \rangle\!\rangle \to t'''}{\langle E[thunk(preamble, body, epilogue)], (e, t, d)\rangle \to (e', t''', d)}\,\textit{Thunk}$$

$$\frac{\langle \mathbf{loop}(i, ext, E[\mathbf{access}(head, j\ldots, i)]), s\rangle \to s'}{\begin{aligned}&\langle \mathbf{loop}(i, ext, E[\mathbf{access}(sequence(head, tail), j\ldots, i)]), s\rangle \\ &\quad \to \langle \mathbf{loop}(i, ext, E[\mathbf{access}(tail, j\ldots, i)]), s'\rangle\end{aligned}}\,\textit{Sequence}$$

$$\frac{\langle node, algebra\rangle \to node'}{\langle E[node], s\rangle \to \langle E[node'], s\rangle}\,\textit{Simplify}$$

$$\frac{\langle\!\langle seek(a), t \rangle\!\rangle \to t'}{\begin{aligned}&\langle \mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{access}(stepper(seek, body, next), j\ldots, i)]), (e, t, d)\rangle \\ &\quad \to \langle \mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{access}(stepper(body, next), j\ldots, i)]), (e, t', d)\rangle\end{aligned}}\,\textit{StepperSeek}$$

$$\frac{\langle \mathbf{loop}(i, ext, E[\mathbf{access}(body, j\ldots, i)]), (e, t, d)\rangle \to (e', t', d) \qquad \langle\!\langle next, t' \rangle\!\rangle \to t''}{\begin{aligned}&\langle \mathbf{loop}(i, ext, E[\mathbf{access}(stepper(body, next), j\ldots, i)]), s\rangle \\ &\quad \to \langle \mathbf{loop}(i, ext, E[\mathbf{access}(stepper(body, next), j\ldots, i)]), (e', t'', d)\rangle\end{aligned}}\,\textit{StepperNext}$$

$$\frac{}{\begin{aligned}&\langle \mathbf{loop}(i, \mathbf{extent}(a, b), body), s\rangle \\ &\quad \to \langle \mathbf{block}(\mathbf{define}(i, a, body), \mathbf{sieve}(a < b, \mathbf{loop}(i, \mathbf{extent}(a + 1, b), body))), s\rangle\end{aligned}}\,\textit{Loop}$$

Fig. 15. Looplet evaluation semantics. The state s of the program is a tuple (e, t, d) of a variable value environment, host language state t, and the current tensor scope, d. Note that E is an evaluation context that applies anywhere in the syntax tree. The nonlocal evaluations of looplets are what allow looplets to hoist conditions and subranges out of loops. However, this also means we must specify the priority in which we apply looplet rules, which is as follows: *Thunk* > *Phase* > *Switch* > *Simplify* > *Run* > *Spike* > *Sequence* > *StepperSeek* > *StepperNext* > *Lookup* > *AcceptRun* > *Unfurl* > *Loop* > *Access*. Many looplets, most notably the thunk looplet, introduce variables into the host language environment. While looplets may modify variables they introduce themselves (steppers often increment some state variables), we forbid child looplets from modifying state variables that they didn't introduce. This allows us to treat the **value** node as a constant. The *Simplify* rule references *algebra*, which is our variable defining a set of straightforward simplification rules. These rules include simple properties like $x * 0 \to 0$ to more complicated ones such as constant propagation. We omit the full set of rules for brevity and refer to [7, Figure 5] for examples.

to variable values in the "target environment" would simply be lowered to variable assignments. Second, instead of evaluating expressions when we apply a rule, we lower the expressions to code, using variables to reference the results. Finally, when runtime information is used to determine which rule to use, we instead lower both rules and emit a runtime branch to decide between them.

For example, though there are two rules to lower **sieve** depending on whether the condition is true (*SieveTrue* and *SieveFalse*) both branches are lowered with an if to decide between them.

This stage of the compiler carefully intermixes our control flow and tensors into looplets so the combination can be successfully simplified. The crux is that loops are decomposed into looplets, introduced via the *Unfurl* rule. Unfurl is defined in Section 3.3. As the looplets are lowered, repeated values and constants are slowly uncovered (e.g. *Run* and *Switch*, respectively). We use rewrite rules in *Simplify* to eliminate cases, unnecessary iterations, and so forth based on the information provided via looplets and via the control flow (loops, sieve, definitions). Our rewrites rely on concordization and wrapperization to reliably transform complex index expressions and control flow into loops and wrappers, our recursive lowering stage can use looplets to simplify the combination of tensor structures and control flow to eliminate unneeded work. Our rewrites also rely on tensor life cycles to guarantee their validity and avoid arbitrary mixes of reads and writes.

Finch lowers loops from the outside to the inside, focusing on a single outer loop at a time. The lowering of a single loop rewrites the entire loop body, even when the body contains multiple inner loops. It is true that a complex loop body may require a fairly invasive rewrite, but the rewrite is broken into many manageable pieces. The unfurl operation applies to all tensor access expressions involving the outer loop index, simply substituting each tensor with a corresponding looplet nest expression. The Looplet lowering rules in Figure 15 specify more granular rewrites that affect the entire loop body and involve the interaction between multiple looplets in different accesses.

```
A = Tensor(Dense(Element(0.0)))
B = Tensor(SparseList(SparseList(Element(0.0))))
C = Tensor(Dense(SparseList(Element(0.0))))
D = Tensor(SparseList(Element(0.0)))
```

```
for k = _
  A .= 0
  for i = _
    A[i] = B[i, k] * 2
  for j = _
    C[j, k] = A[i]^2 + D[k]
```

→

```
for k = _
  A .= 0
  for i = _
    A[i] = Stepper(...)[i, k] * 2
  for j = _
    Lookup(...)[j, k] =
      A[i]^2 + Stepper(...)[k]
```

Fig. 16. Unfurling accesses on the k loop.

We chose this level-by-level design to avoid combinatorial explosions handling different formats across two or more levels. Each level format describes one dimension of a tensor at a time, and Finch only lowers one loop at a time. The *Unfurl* function substitutes each level format with a looplet expression composed from a fixed set of looplets. Then, we need only consider the relationships between each looplet, and not each format. An example of *Unfurl* is given in Figure 16.

6 Example Lowering

In Figures 17-18, we illustrate the lowering of a program that sums the upper triangle of a matrix.

Input Program:

```
A = Tensor(Dense(SparseList(Element(0.0))), m, n)
s = Tensor(Element(0.0))
@finch begin
  s .= 0.0
  for j = _
    for i = _
      if i <= j
        s[] += A[i, j]
```

Step 1: Normalization

```
T = UpTriMask()
A = Tensor(Dense(SparseList(Element(0.0))), m, n)
s = Tensor(Element(0.0))
@finch begin
  @declare(s, 0.0)
  for j = 1:n
    for i = 1:m
      if T[i, j]
        s[] += A[i, j]
  @freeze(s)
```

Fig. 17. Example normalization of a Finch program. Wrapperization replaces `i <= j` with `UpTriMask()[i, j]`. Dimensionalization computes `i = 1:m`, and `j = 1:n`. The input is already concordant. Lifecycle statements are added. Normalization readies the program for recursive lowering, shown in Figure 18.

Step 2: Declaring `s` The declare statement initializes the `s` tensor.

```
s.lvl.val[1] = 0.0
@finch begin
  for j = 1:n
    for i = 1:m
      if UpTriMask()[i, j]
        s[] += A[i, j]
  @freeze(s)
```

Step 3: Unfurling `j` To process the `j` loop, we unfurl both tensors that access `j`:

```
s.lvl.val[1] = 0.0                     t = unfurl(UpTriMask()) =
@finch begin                             Lookup(
  for j = 1:n                              body(j) = UpTriMaskCol(j))
    for i = 1:m                        a = unfurl(A::DenseLevel) =
      if (t[j])[i]                       Lookup(
        s[] += (a[j])[i]                   body(j) = SubFiber(A.lvl.lvl, j))
  @freeze(s)
```

Step 4: Lower Lookups We insert a for-loop:

```
s.lvl.val[1] = 0.0
for j = 1:n
  @finch begin
    for i = 1:m
      if UpTriMaskCol(j)[i]
        s[] += SubFiber(A.lvl.lvl, j)[i]
@finch @freeze(s)
```

Step 5: Unfurling `i` Next, we process the `i` loop. Again, we unfurl both tensors:

```
                          t = unfurl(UpTriMaskCol(j)) =
                            Sequence(
                              Phase(stop = j, Run(true)),
                              Phase(Run(false)))
s.lvl.val[1] = 0.0        a = unfurl(SubFiber(
for j = 1:n                   A.lvl.lvl::SparseListLevel, j)) =
  @finch begin              Thunk(
    for i = 1:m               preamble = (q = A.lvl.lvl.ptr[j]),
      if t[i]                 Stepper(
        s[] += a[i]             seek = (i) -> (
@finch @freeze(s)                 q = binarysearch(A.lvl.lvl.idx, i)),
                                stop = A.lvl.lvl.idx[q],
                                body = Spike(
                                  body = 0,
                                  tail = A.lvl.lvl.val[q]),
                                next = (q += 1)))
```

Step 6: Lower Thunks We move the preambles out of any Thunks and unwraps them:

```
                            t = Sequence(
                              Phase(stop = j, Run(true)),
s.lvl.val[1] = 0.0            Phase(Run(false)))
for j = 1:n                 a = Stepper(
  q = A.lvl.lvl.ptr[j]        seek = (i) -> (
  @finch for i = 1:j            q = binarysearch(A.lvl.lvl.idx, i)),
    if t[i]                   stop = A.lvl.lvl.idx[q],
      s[] += a[i]             body = Spike(
@finch @freeze(s)               body = 0,
                                tail = A.lvl.lvl.val[q]),
                              next = (q += 1))
```

Step 7: Lower Sequences We insert loops for each phase:

```
s.lvl.val[1] = 0.0          t_1 = Run(true)
for j = 1:n                 t_2 = Run(false)
  q = A.lvl.lvl.ptr[j]      a = Stepper(
  @finch for i = 1:j          seek = (i) -> (
    if t_1[i]                   q = binarysearch(A.lvl.lvl.idx, i)),
      s[] += a[i]             stop = A.lvl.lvl.idx[q],
  @finch for i = j+1:m        body = Spike(
    if t_2[i]                   body = 0,
      s[] += a[i]               tail = A.lvl.lvl.val[q]),
@finch @freeze(s)             next = (q += 1))
```

Step 8: Lower Runs We simply replace runs with their value:

```
  s.lvl.val[1] = 0.0
  for j = 1:n             a = Stepper(
    q = A.lvl.lvl.ptr[j]  seek = (i) -> (
     @finch for i = 1:j     q = binarysearch(A.lvl.lvl.idx, i)),
      if true             stop = A.lvl.lvl.idx[q],
        s[] += a[i]       body = Spike(
    @finch for i = j+1:m    body = 0,
      if false              tail = A.lvl.lvl.val[q]),
        s[] += a[i]       next = (q += 1))
@finch @freeze(s)
```

Step 9: Simplify The simplification pass removes the if statement in the first loop and removes the second loop:

```
                            a = Stepper(
s.lvl.val[1] = 0.0            seek = (i) -> (
for j = 1:n                     q = binarysearch(A.lvl.lvl.idx, i)),
  q = A.lvl.lvl.ptr[j]        stop = A.lvl.lvl.idx[q],
  @finch for i = 1:j          body = Spike(
      s[] += a[i]               body = 0,
@finch @freeze(s)               tail = A.lvl.lvl.val[q]),
                              next = (q += 1))
```

Step 10: Lower Steppers We process steppers with a while loop:

```
s.lvl.val[1] = 0.0
for j = 1:n
  q = A.lvl.lvl.ptr[j]
  k = 1
  while k < j                           a = Spike(
    k_2 = A.lvl.lvl.idx[q]                body = 0,
    @finch for i = k:min(k_2, j)          tail = A.lvl.lvl.val[q])
      s[] += a[i]
    q += 1
@finch @freeze(s)
```

Step 11: Lower Spikes A Spike is a Run followed by a single value:

```
s.lvl.val[1] = 0.0
for j = 1:n
  q = A.lvl.lvl.ptr[j]
  k = 1
  while k < j
    k_2 = A.lvl.lvl.idx[q]
    @finch for i = k:min(k_2, j)
      s[] += 0
    i = k_2
    @finch if i < j
      s[] += A.lvl.lvl.val[q]
    q += 1
@finch @freeze(s)
```

Step 12: Simplify We observe that addition of `0` is a no-op:

```
s.lvl.val[1] = 0.0
for j = 1:n
  q = A.lvl.lvl.ptr[j]
  k = 1
  while k < j
    k_2 = A.lvl.lvl.idx[q]
      i = k_2
      if i < j
        s[] += A.lvl.lvl.val[q]
    q += 1
  @finch @freeze(s)
```

Step 13: Lower Freeze Finally, the final `freeze` is a no-op and we obtain:

```
s.lvl.val[1] = 0.0
for j = 1:n
  q = A.lvl.lvl.ptr[j]
  k = 1
  while k < j
    k_2 = A.lvl.lvl.idx[q]
      i = k_2
      if i < j
        s[] += A.lvl.lvl.val[q]
    q += 1
```

Fig. 18. Example recursive lowering of a Finch program, continued from Figure 17. Though Finch programs look as if they are written for dense loops, Finch specializes the code during lowering so that only the necessary elements of structure need to be processed. The final program accesses only the upper triangle of A, though the original code looks as though it loops over all `i` and `j`.

7 Case Studies

We evaluate Finch on a broad set of applications to showcase it's efficiency, flexibility, and expressiveness. All experiments were run on a single core of a 12-core 2-socket 2.50GHz Intel Xeon CPU E5-2680 v3 with 128GB of memory. Finch is implemented in Julia v1.9, targeting LLVM through Julia. All timings are the minimum of 10,000 runs or 5s of measurement, whichever happens first.

Our study of sparse-sparse-matrix multiply (SpGEMM) translates classical lessons from sparse performance engineering into the language of Finch. Our study of sparse-matrix-dense-vector multiply (SpMV) highlights the benefits of structural specialization. Our studies of image morphology and graph algorithms show how Finch can express complex real-world kernels.

7.1 Sparse Matrix-Vector Multiply (SpMV)

Sparse matrix-vector multiply (SpMV) has a wide range of applications and has been thoroughly studied [66, 104]. Because SpMV is bandwidth bound, many formats have been proposed to reduce the footprint [62]. The wide range of applications results in a wide range of tensor structures, making it an effective kernel to demonstrate the utility of our programming model.

```
y .= 0
for j = _, i = _
  y[i] += A[i, j] * x[j]
```

```
y .= 0
for j = _, i = _
  y[j] += A[i, j] * x[i]
```

```
y .= 0
for j = _
  let x_j = x[j]
    y_j .= 0
    for i = _
      let A_ij = A[i, j]
        y[i] += x_j * A_ij
        y_j[] += A_ij * x[i]
    y[j] += y_j[] + D[j] * x_j
```

Fig. 19. Finch row-major, column-major and symmetric SpMV Programs. Note that the upper triangle of the input is pre-computed for the symmetric program. Reads to the canonical triangle are reused with a **define** statement, and the results are written to both relevant locations using multiple outputs.

Figure 20 displays the performance of SpMV measured relative to TACO. We varied both the data formats and SpMV algorithms in Finch (see Figure 19), and display the best-performing combination. Precisely which Finch format performed best on which matrices is shown in the figure. We compare against TACO (best of row or column-major), Julia's standard library (column major), baseline Finch (best of row or column-major with CSC format), Eigen (row-major) [46], MKL (row-major) [1], and CORA (unscheduled, row-major) [38]. Our test suite is the union of datasets from three previous papers: the matrices used by Ahrens et al. to test a variable block row format partitioning strategy [6], Kjolstad et al. to test the TACO library [58], and Leskovec et al. to evaluate graph clustering algorithms [64]. We left out two very large matrices (`Janna/Emilia_923` and `Janna/Geo_1438`); the remaining matrices had a maximum of 12 million nonzeros. We also added some synthetic matrices, $10,000 \times 10,000$ banded matrices with bandwidth 5, 30, and 100, a 1024×1024 upper triangular matrix, and a $1,000,000 \times 1,000,000$ reverse permutation matrix.

Finch introduces tradeoffs between the benefits of specialization and the branching it induces. Specialization is most effective for common cases that can be highly simplified, such as the zero region of sparse matrices. For example, we found it was faster to pre-compute the upper triangle of our symmetric matrix, rather than calculate it on the fly using a mask expression such as `i < j`, which changed the exit condition of the inner loop. The option to de-specialize certain conditional expressions is another example of how Finch can widen the design space for structured operators.

Our result shows that different formats perform better on different matrices, and that Finch can be used to exploit these formats. In general, SpMV performance was superior for the level format that best paralleled the structure of the tensor. The best Finch format had a geomean speedup of 1.27 over TACO. Matrices with block structure like `exdata_1`, `TSOPF_RS_b678_c1`, and `heart3` performed best with the SparseBlockList format with speedups of 2.75, 1.80, and 1.20 relative to TACO. Matrices with banded structure performed the best with the SparseBand format. In particular, on the `large_band` and the `medium_band matrices`, our banded format showed a speedup of 2.50 and 2.02 relative to TACO. On our triangular matrix, Finch had a speedup of 3.04 over TACO,

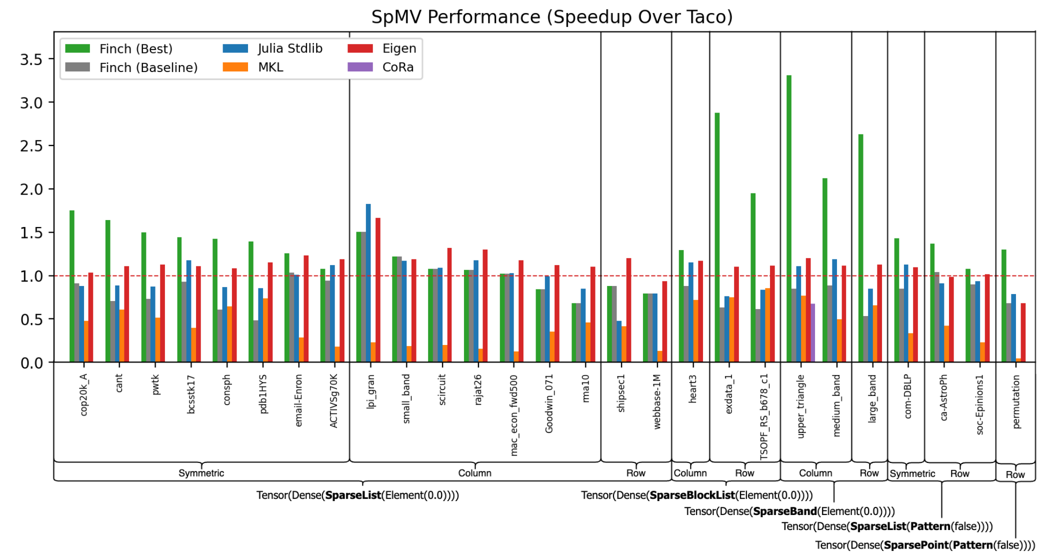


Fig. 20. SpMV performance, organized by the best performing Finch format. Programs are from Figure 19. "Finch (Best)" is the fastest among the formats we tested. "Finch (Baseline)" is the faster of row or column major "SparseList".

outperforming even CORA, which was designed for ragged tensors but targeted more towards cache blocking than to the specific structure of the tensor. Similarly, using a SparsePoint format obtained a speedup of 1.30 by avoiding a loop over nonzeros (since there is only ever one nonzero in the SparsePoint level). The Pattern leaf level performed better than the Element leaf level on Boolean graph matrices. On `ca-AstroPh`, SparseList-Pattern format resulted in a speedup of 1.17 over TACO, while SparseList-Element only achieved 1.04. Our results clearly demonstrate the utility of being able to vary both the algorithm and the format to match the structure of the tensor.

Though MKL is closed-source, using `perf` on `mac_econ_fwd_500`, we found that MKL had noticeably higher branch mispredictions than expected (23%, as compared to TACO's 1%), and that many instructions were vectorized with AVX (23% as compared to TACO's 0.03%), indicating a vectorized row-major implementation. Such an implementation should underperform since the matrix has only 6 nonzeros per row and the inner loop would iterate only once between setup and cleanup of the vector registers. Taco and Eigen performed similarly, both emitting simple loops over non-zeros. This sometimes has a slight advantage over Finch, which uses the coordinate as the loop variable. Still, Finch's structural specification showed a clear advantage on our test inputs.

7.2 Sparse-Sparse Matrix Multiply (SpGEMM)

We compute the $M \times N$ sparse matrix C as the product of $M \times K$ and $K \times N$ sparse matrices A and B. There are three main approaches to SpGEMM [102, Section 2.2]. The inner-products algorithm takes dot products of corresponding rows and columns, while the outer-products algorithm sums the outer products of corresponding columns and rows. Gustavson's algorithm sums the rows of B scaled by the corresponding nonzero columns in each row of A. Inner-products is known to be asymptotically less efficient than the others, as we must do a merge operation to compute each of the MN entries in the output [8]. We will show that our ability to implement these latter methods exceeds that of TACO, translating to asymptotic benefits.

```
@finch begin
  C .= 0
  for j=_
    for i=_
      for k=_
        C[i, j] += AT[k, i] * B[k, j]
  return C
```

```
w = Tensor(SparseByteMap(Element(0)))
@finch begin
  C .= 0
  for j=_
    w .= 0
    for k=_
      for i=_
        w[i] += A[i, k] * B[k, j]
    for i=_
      C[i, j] = w[i]
```

```
w = Tensor(SparseHash(SparseHash(Element(0))))
@finch begin
  w .= 0
  for k=_
    for j=_
      for i=_
        w[i, j] += A[i, k] * BT[j, k]
  C .= 0
  for j=_, i=_
    C[i, j] = w[i, j]
```

Fig. 21. Inner Products, Gustavson's, and Outer Products matrix multiply in Finch

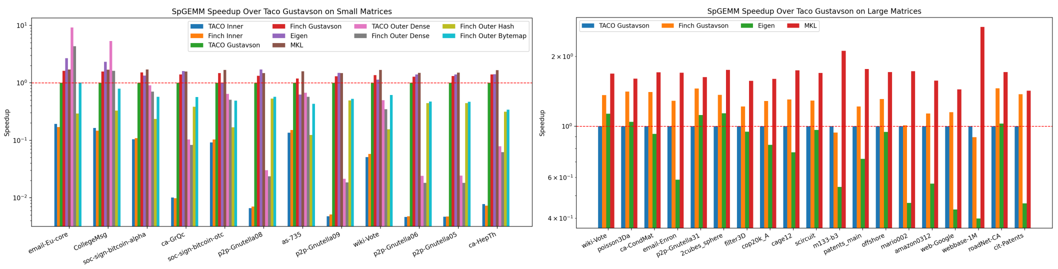


Fig. 22. A comparison of several matrix multiply algorithms in Finch, Taco, Eigen, and MKL. The top images show results on the same dataset as [102]. We use only Gustavson's algorithm on larger matrices.

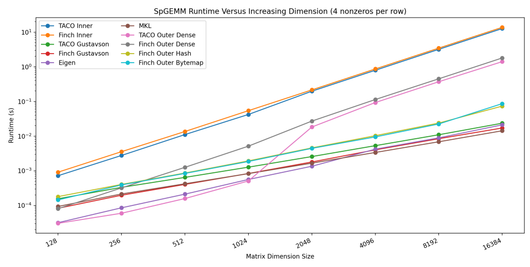


Fig. 23. An asymptotic comparison of several matrix multiply algorithms on increasingly large Erdős-Rényi matrices with an average of 4 nonzeros per row. Gustavsons and sparse output outer products have an asymptotic advantage over inner products or dense output outer products as the problem size grows.

Figure 21 implements all three approaches in Finch, and Figure 22 compares the performance of Finch to TACO, Eigen, and MKL on the matrices of Zhang et al. [102]. While these algorithms differ mainly in their loop order, different data structures must be used to support the various access patterns. In our Finch implementation of outer products, we use a sparse hash table, as it is fully-sparse and randomly accessible. Since TACO does not support multidimensional sparse workspaces, its outer products uses a dense intermediate, which leads to an asymptotic slow down shown in Figure 22. Although a sparse bytemap has a dense memory footprint, we use it in our Finch implementation of Gustavson's for the smaller 1-dimensional intermediate. We note that the bytemap format in TACO's Gustavson's implementation is a hard-wired optimization, whereas Finch's programming model allows us to write algorithms with explicit temporaries, formats, and transpositions. As depicted in Figure 22, Finch achieves comparable performance with TACO on smaller matrices when we use the same datastructures, and significant improvements when we use better datastructures. Finch outperforms TACO overall, with a geomean speedup of 1.30. Finch and TACO both outperform Eigen by a significant margin, as Eigen is designed for usability and generality but is not heavily optimized. Finch is competitive with, but slightly slower than MKL. We cannot comment extensively on MKL's performance as we cannot access source code. We suspect MKL uses Gustavson's algorithm with a optimized sorting routine.

Figure 23 includes a scaling study to show the asymptotic impact of algorithms and output formats as the SpGEMM problem size grows. We consider uniformly random $N \times N$ matrices with a fraction of $p = 4/N$ nonzeros. Inner-products runs in expected time $\Theta(N^3 p) = \Theta(4N^2)$. Outer-products with a sparse output format runs in expected time $\Theta(N^3 p^2) = \Theta(16N)$, which is an asymptotic improvement as the matrix gets sparser. Outer-products with a dense output format runs in expected time $\Theta(N^3 p^2 + N^2) = \Theta(N^2)$, which is an asymptotic disadvantage when the number of nonzeros per row (Np) is small. Our plot shows that Finch's outer products outperforms TACO's outer products, as Finch supports sparse output formats where TACO does not. Finch is the first tensor compiler to support all three strategies with both sparse and dense output formats.

7.3 Graph Analytics

We used Finch to implement both Breadth-first search (BFS) and Bellman-Ford single-source shortest path. Our BFS implementation and graphs datasets are taken from Yang et al. [99], including both road networks and scale-free graphs (bounded node degree vs. power law node degree).

Direction-optimization [16] is crucial for achieving high BFS performance in such scenarios, switching between push and pull loop orders to efficiently explore graphs. Push traversal visits the neighbors of each frontier node, while pull traversal visits every node and checks to see if it has a neighbor in the frontier. The advantage of pull traversal is that we may terminate our search once we find a node in the frontier, saving time in the event the push traversal were to visit most of the graph anyway. Early break is the critical part of control flow in this algorithm, though the algorithms also require different loop orders, multiple outputs, and custom operators. Finch performs well because it can directly express algorithms comparable to competitive libraries.

```
V = Tensor(Dense(Element(false)))
P = Tensor(Dense(Element(0)))
F = Tensor(SparseByteMap(Pattern()))
_F = Tensor(SparseByteMap(Pattern()))
A = Tensor(Dense(SparseList(Pattern())))
AT = Tensor(Dense(SparseList(Pattern())))

function bfs_push(_F, F, A, V, P)
  @finch begin
    _F .= false
    for j=_, k=_
      if F[j] && A[k, j] && !(V[k])
        _F[k] |= true
        P[k] <<choose(0)>>= j
    return _F
```

```
function bfs_pull(_F, F, AT, V, P)
  p = ShortCircuitScalar{0}()
  @finch begin
    _F .= false
    for k=_
      if !V[k]
        p .= 0
        for j=_
          if F[follow(j)] && AT[j, k]
            p[] <<choose(0)>>= j
        if p[] != 0
          _F[k] |= true
          P[k] = p[]
    return _F
```

```
_D = Tensor(Dense(Element(Inf)), n)
D = Tensor(Dense(Element(Inf)), n)
function bellmanford(A, _D, D, _F, F)
  @finch begin
    F .= false
    for j = _
      if _F[j]
        for i = _
          let d = _D[j] + A[i, j]
            D[i] <<min>>= d
            F[i] |= d < _D[i]
```

Fig. 24. Graph Applications written in Finch. Note that parents are calculated separately for Bellman-Ford. The *choose*(z) operator is a GraphBLAS concept which returns any argument that is not z.

Figure 25 compares performance to Graphs.jl, a Julia library, and the LAGraph Library, which uses sparse linear algebra via GraphBLAS [68]. On BFS, Finch is competitive even with the hardwired optimizations of GraphBLAS, a geomean slowdown of 1.22. Direction-optimization enhances performance for scale-free graphs. On Bellman-Ford (with path lengths and shortest-path tree), Finch's support for multiple outputs, sparse inputs, and masks leads to a geomean speedup of 2.47 over GraphBLAS. Our artifact lists code for BFS and Bellman-Ford in Finch (53 and 54 LOC) and LAGraph (215 and 227 LOC), and we invite readers to compare the clarity of the algorithms [9].

7.4 Image Morphology

Some image processing pipelines stand to benefit from structured formats [34]. We consider two operations in binary image morphology: binary erosion (computing a mask), and a masked histogram (using a mask to avoid work). Our images are all binary, either by design or thresholding.

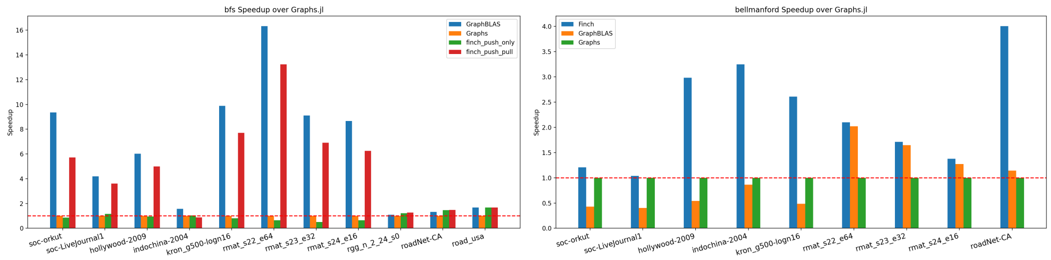


Fig. 25. Performance of graph apps across various tools. finch_push_only exclusively utilizes push traversal, while finch_push_pull applies direction-optimization akin to GraphBLAS. Finch's support for push/pull traversal and early break facilitates direction-optimization. Among GraphBLAS's five variants for Bellman-Ford, we selected LAGraph_BF_full1a, consistently the fastest with our graphs. We did not include Bellman-Ford results for graphs with high diameter as they timed out (> 1 hour).

Finch allows us to choose our datastructure, so we may choose to use either a dense representation with bytes (`Dense(Element(0x00))`), a bit-packed representation (`Dense(Element(UInt64))`), or a run-length encoded representation that represents runs of true or false (`SparseRunList(Pattern())`). All of these have their advantages. The dense representation induces the least overhead, the bit-packed representation can take advantage of bitwise binary ops, and the run-length encoded version only uses memory and compute when the pattern changes.

Finch also allows us to choose our algorithm, and we can implement erosion in a few ways. The erosion operation turns off a pixel unless all of it's neighbors are on. This can be used to shrink the boundaries of a mask, and remove point instances of noise [39]. This introduces three instances of structure in the control flow: the mask, the padding of inputs, and the convolutional filter. We focused on the filter. We can understand the filter as a structured tensor of circular shifts, or we can understand each shifted view of the data in an unrolled stencil computation as a structured tensor, or a two part stencil where we compute the horizontal then vertical part of the stencil. We experimented with these options and found that the last approach performed best, due to fitting the storage formats while reducing the amount of work with intermediate temporaries. Figure 26 displays example erosion algorithms for bitwise or run-length-encoded algorithms.

We compared against OpenCV on four datasets. We randomly selected 100 images from the MNIST [63] and Omniglot [61] character recognition datasets, as well as a dataset of human line drawings [36]. We also hand-selected a subset of mask images (these images were less homogeneous, so we listed them in our artifact [9]) from a digital image processing textbook [44]. All images were thresholded, and we also include versions of the images that have been magnified before thresholding, to induce larger constant regions. In our erosion task, the SparseRunList format performs the best as it is asymptotically faster and uses less memory, leading to a 19.5X speedup over OpenCV on the sketches dataset, which becomes arbitrarily large as we magnify the images (here shown as 266X). Finch achieves these speedups by exploiting structured sparsity to do less work than OpenCV's more naive dense implementation, which unnecessarily reads most of an image or mask. However, we believe the 51.6x on MNIST is due to calling overhead in OpenCV. The bitwise kernels were effective as well, and would be more effective on datasets with less structure. A strength of Finch is that it supports structured datasets, even over bitwise operations.

We also implemented a histogram kernel. We used an indirect access into the output to implement this (Figure 26), something not many sparse frameworks support. We compare to OpenCV since the OpenCV histogram function also accepts a mask. If we use `SparseRunList(Pattern())` for our mask,

Wordwise Erosion:

```
output .= false
for y = _
  tmp .= false
  for x = _
    tmp[x] = coalesce(input[x, ~(y-1)], true) & input[x, y] &
   ↪  coalesce(input[x, ~(y+1)], true)
  for x = _
    output[x, y] = coalesce(tmp[~(x-1)], true) & tmp[x] &
   ↪  coalesce(tmp[~(x+1)], true)
```

Masked Histogram:

```
bins .= 0
for x=_
  for y=_
    if mask[y, x]
      bins[div(img[y, x], 16) + 1] += 1
```

Bitwise Erosion:

```
output .= 0
for y = _
  tmp .= 0
  for x = _
    if mask[x, y]
      tmp[x] = coalesce(input[x, ~(y-1)], 0xFFFFFFFF) & input[x,
     ↪  y] & coalesce(input[x, ~(y+1)], 0xFFFFFFFF)
  for x = _
    if mask[x, y]
      let tl = coalesce(tmp[~(x-1)], 0xFFFFFFFF), t = tmp[x], tr =
     ↪  coalesce(tmp[~(x+1)], 0xFFFFFFFF)
        let res = ((tr << (8 * sizeof(UInt) - 1)) | (t >> 1)) & t &
       ↪  ((t << 1) | (tl >> (8 * sizeof(UInt) - 1)))
          output[x, y] = res
```

Fig. 26. Two approaches to erosion in Finch. The *coalesce* function defines the out of bounds value. On left, the naive approach. On `SparseRunList(Pattern())` inputs, this only performs operations at the boundaries of constant regions. On right, a bitwise approach, using a mask to limit work to nonzero blocks of bits.

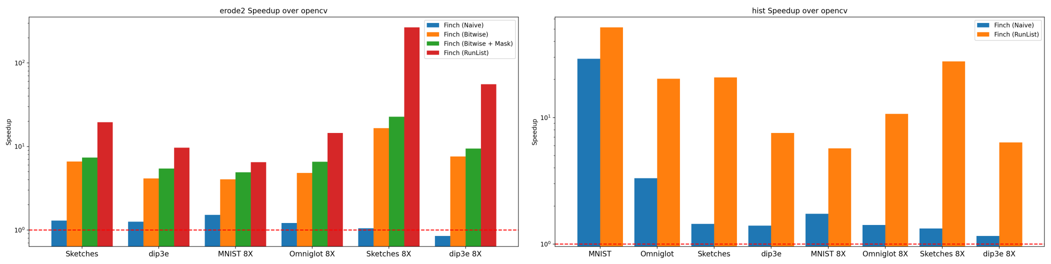


Fig. 27. Performance of Finch on image morphology tasks. On left, we run 2 iterations of erosion. On right, we run a masked histogram. We display the geomean speedup within each dataset.

we can reduce the branching in the masked kernel and get better performance. The improvements with SparseRunList are seen in the histogram task too, as it allows us to mask off contiguous regions of computation, instead of individual pixels, reducing the branches and leading to a significant speedup (20.3x on Omniglot and 20.8x on sketches). In a low compute task such as a histogram, skipping many reads for the mask via structured sparsity can lead to huge speedups.

8 Related Work

The related work spans several areas, from libraries to languages, from dense to structured data.

Libraries for Dense Data: Many libraries specialize in dense computations. Perhaps the most well-known example is NumPy [48], and a classic example is the BLAS, though several BLAS routines are specialized to symmetric, hermitian, and triangular matrices [10]. Many research projects have advanced on BLAS, such as BatchedBlas and BLIS [35, 92].

Libraries for Structured Data: Many libraries support BLAS plus a few sparse tensor types, typically CSR, CSC, BCSR, Banded, and COO. Examples include SciPy [94], PETSc [5], Armadillo [77], OSKI [96], Cyclops [85], MKL [1], and Eigen [46]. There are even libraries for very specific kernels and format combinations, such as SPLATT [83] (MTTKRP on CSF). Several of these libraries also feature some graph or mesh algorithms built on sparse matrices. The GraphBLAS [56] supports primitive semiring operations (operations beyond $(+, *)$, such as $(min, +)$ multiplication) which can be composed to enable graph algorithms, some of which are collected in LAGraph [68]. Similarly, the MapReduce and Hadoop platforms support operations on indexed collections [32], and have been used to support graph algorithms in the GBASE library[55]. Several machine learning frameworks support some sparse tensors and operations, most notably TorchSparse[89, 90].

Compilers for Dense Data: Outside of general purpose compilers, many compilers have been developed for optimizing dense data on a variety of control flow. Perhaps the most well known example is Halide [76] and its various descendant such as TVM [27], Exo [53], Elevate [47], and ATL [65]. These languages typically support most control flow except for an early break though some don't support arbitrary reading/writing or even indirect accesses. Several polyhedral languages, such as Polly [45], Tiramisu [12], CHiLL [26], Pluto [22], and AlphaZ [101] offer similar capabilities in terms of control flow though they often support more irregular regions. The density of this research represents the density of support for dense computation.

Compilers for Structured Data: Several compilers exist for several types of structured data, often featuring separate languages for the storage of the structured data and the computation. The TACO compiler originally supported just plain Einsum computations [58], but has been extended several times to support (single dimensional) local tensors [57], imperfectly nested loops [33], breaks via semi-rings [49], windowing and tiling [80], and convolution [97], and compilation in MLIR [20], all as separate extensions. Similarly, TACO originally support just dense and CSF like N dimensional structures, but was extended independently to support COO like structures [29], and tree like structures [28], as separate extensions. SparseTIR is a similar system supporting combined sparse formats (including block structures) [100]. The SDQL language offers a similar level of control flow [81], but only on sparse hash tables. Similarly, SDQL has been extended with a system that allows one to specify formats as queries on a set of base storage types [79] and separately by another system that describes static symmetries and other structures as predicates [42]. The Taichi language focuses on a single sparse data structure made from dense blocks, bit-masks, and pointers [51]. The sparse polyhedral framework builds on CHiLL for the purpose of generating inspector/executor optimizations [87] though the branch of this work that specifies sparse formats separately from the computation (otherwise they are inlined into the computation manually) seems to apply mainly to Einsums [103]. Second to last, SQL's classical physical/logical distinction is the classic program/format distinction, and SQL supports a huge variety of control flow constructs [31, 60]. However, many SQL or dataframe systems rely on b-trees, columnar, or hash tables, with only a few systems, such as Vectorwise [21], LaraDB [52], GMAP [91], or SciDB [86] building physical layouts with other constructs based in tensor programming. However, tensor based databases are a new focus given the rise of mixed ML/DB pipelines [15, 67]. Lastly, SPIRAL focuses on recursively defined datastructures and recursively define linear algebra, and can therefore express a structure and computation that none of the systems mentioned above can: a Cooley–Tukey FFT [40, 41].

Other Architectures: Sparse compilers have been extended to many architectures. An extension of TACO supports GPU [80], Cyclops [84, 85] and SPDistal [98] support distributed memory, and the Sparse Abstract Machine [50] supports custom hardware. We believe that supporting control flow is the first step towards architectural support beyond unstructured sparsity.

9 Conclusion

Finch automatically specializes flexible control flow to diverse data structures, facilitating productive algorithmic exploration, flexible tensor programming, and efficient high-level interfaces for a wider variety of applications than ever before.

Acknowledgments

Intel and NSF PPoSS Grant CCF-2217064; DARPA PROWESS Award HR0011-23-C-0101; NSF SHF Grant CCF-2107244; DoE PSAAP Center DE-NA0003965; DARPA SBIR HR001123C0139

10 Data Availability Statement

The most recent version of the Finch compiler is currently available as open-source software at https://github.com/finch-tensor/Finch.jl. The benchmarks used to construct this paper are available as an artifact on Zenodo at https://doi.org/10.5281/zenodo.14597754. The reviewed version of the artifact is https://doi.org/10.5281/zenodo.14735207. They can also be found in the `oopsla-25-artifact` branch of the FinchBenchmarks repository on GitHub at https://github.com/finch-tensor/FinchBenchmarks. The artifact contains a copy of the Finch.jl compiler version v1.1.0, our benchmarking code, and instructions to replicate all results of the paper.

References

[1] 2024. Developer Reference for Intel® oneAPI Math Kernel Library for Fortran. (April 2024). https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-fortran/2024-0/overview.html

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[3] Hameer Abbasi. 2023. Plans for new sparse compilation backend · pydata/sparse · Discussion #618. https://github.com/pydata/sparse/discussions/618

[4] Harold Abelson and Gerald Jay Sussman. 1996. *Structure and Interpretation of Computer Programs*. The MIT Press. https://library.oapen.org/handle/20.500.12657/26092 Accepted: 2019-01-17 23:55.

[5] Shrirang Abhyankar, Getnet Betrie, Daniel A Maldonado, Lois C Mcinnes, Barry Smith, and Hong Zhang. [n. d.]. PETSc DMNetwork: A Scalable Network PDE-Based Multiphysics Simulator. ([n. d.]).

[6] Willow Ahrens and Erik G. Boman. 2021. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. https://doi.org/10.48550/arXiv.2005.12414 arXiv:2005.12414 [cs].

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

[8] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

[9] Willow Ahrens, Radha Patel, Teodoro Fields Collin, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and Structured Tensor Programming with Control Flow: The Artifact. https://doi.org/10.5281/zenodo.14735207

[10] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719604

[11] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability (IRE-AIEE-ACM '57 (Western))*. Association for Computing Machinery, New York, NY, USA, 188–198. https://doi.org/10.1145/1455567.1455599

[12] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: a polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Washington, DC, USA, 193–205.

[13] S Balay, S Abhyankar, Mark F Adams, J Brown, P Brune, K Buschelman, L Dalcin, A Dener, V Eijkhout, W Gropp, and others. 2020. *PETSc Users Manual (Rev. 3.13)*. Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[14] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. 2010. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics* 14 (Jan. 2010), 7:3.7–7:3.24. https://doi.org/10.1145/1498698.1564507

[15] Peter Baumann, Dimitar Misev, Vlad Merticariu, and Bang Pham Huu. 2021. Array databases: Concepts, standards, implementations. *Journal of Big Data* 8 (2021), 1–61.

[16] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[17] Robert M Bell and Yehuda Koren. 2007. Lessons from the Netflix prize challenge. *Acm Sigkdd Explorations Newsletter* 9, 2 (2007), 75–79. Publisher: ACM New York, NY, USA.

[18] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–12.

[19] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. *arXiv:1209.5145 [cs]* (Sept. 2012). http://arxiv.org/pdf/1209.5145.pdf arXiv: 1209.5145.

[20] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Transactions on Architecture and Code Optimization* 19, 4 (Sept. 2022), 50:1–50:25. https://doi.org/10.1145/3544559

[21] PA Boncz and M Zukowski. 2012. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27.

[22] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

[23] Gary Bradski, Adrian Kaehler, and others. 2000. OpenCV. *Dr. Dobb's journal of software tools* 3, 2 (2000).

[24] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C. In *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 643–652.

[25] Keaton J. Burns, Geoffrey M. Vasil, Jeffrey S. Oishi, Daniel Lecoanet, and Benjamin P. Brown. 2020. Dedalus: A flexible framework for numerical simulations with spectral methods. *Physical Review Research* 2, 2 (April 2020), 023068. https://doi.org/10.1103/PhysRevResearch.2.023068 _eprint: 1905.10388.

[26] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. A framework for composing high-level loop transformations. *Technical Report 08–897, USC Computer Science Technical Report* (2008).

[27] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[28] Stephen Chou and Saman Amarasinghe. 2022. Compilation of dynamic sparse tensor algebra. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 175:1408–175:1437. https://doi.org/10.1145/3563338

[29] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 123:1–123:30. https://doi.org/10.1145/3276493

[30] Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. 2022. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*. PMLR, 4690–4721.

[31] Chris J Date. 1989. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc.

[32] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[33] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. SparseLNR: accelerating sparse tensor computations using loop nest restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing*. ACM, Virtual Event, 1–14. https://doi.org/10.1145/3524059.3532386

[34] Daniel Donenfeld, Stephen Chou, and Saman Amarasinghe. 2022. Unified Compilation for Lossless Compression and Sparse Computing. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 205–216. https://doi.org/10.1109/CGO53902.2022.9741282

[35] J. Dongarra, S. Hammarling, N. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon. 2017. The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems. *Procedia Computer Science* 108 (Jan. 2017), 495–504. https://doi.org/10.1016/j.procs.2017.05.138

[36] Mathias Eitz, James Hays, and Marc Alexa. 2012. How do humans sketch objects? *ACM Transactions on Graphics* 31, 4 (July 2012), 44:1–44:10. https://doi.org/10.1145/2185520.2185540

[37] Pratik Fegade. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. https://doi.org/10.5281/zenodo.6326456

[38] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 721–747. https://proceedings.mlsys.org/paper_files/paper/2022/file/afe8a4577080504b8bec07bbe4b2b9cc-Paper.pdf

[39] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. 1996. Hypermedia image processing reference. *England: John Wiley & Sons Ltd* (1996), 118–130.

[40] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain-Specific Languages*. Springer, 385–409.

[41] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Puschel, James C. Hoe, and Jose M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (Nov. 2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[42] Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling Structured Tensor Algebra. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 229:204–229:233. https://doi.org/10.1145/3622804

[43] Solomon Golomb. 1966. Run-length encodings (corresp.). *IEEE transactions on information theory* 12, 3 (1966), 399–401. Publisher: Citeseer.

[44] Rafael C. Gonzalez and Richard E. Woods. 2006. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., USA.

[45] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[46] Gaël Guennebaud, Benoît Jacob, and others. 2010. Eigen v3. http://eigen.tuxfamily.org

[47] Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–29. https://doi.org/10.1145/3408974

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

[49] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 128:1–128:29. https://doi.org/10.1145/3485505

[50] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. https://doi.org/10.1145/3582016.3582051

[51] 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* 38, 6 (Nov. 2019), 201:1–201:16. https://doi.org/10.1145/3355089.3356506

[52] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR'17)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3070607.3070608

[53] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[54] Eun-Jin Im and Katherine Yelick. 2001. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *Computational Science — ICCS 2001 (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 127–136. https://doi.org/10.1007/3-540-45545-0_22

[55] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2011. GBASE: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1091–1099.

[56] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose 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

[57] Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO)*. CGO, 180–192. https://doi.org/10.1109/CGO.2019.8661185

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

[59] Donald E. Knuth. 1997. *The Art of Computer Programming: Fundamental Algorithms, Volume 1.* Addison-Wesley Professional. Google-Books-ID: x9AsAwAAQBAJ.

[60] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing (Lecture Notes in Computer Science)*, Christian Lengauer, Martin Griebl, and Sergei Gorlatch (Eds.). Springer, Berlin, Heidelberg, 318–327. https://doi.org/10.1007/BFb0002751

[61] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. 2015. Human-level concept learning through probabilistic program induction. *Science* 350, 6266 (Dec. 2015), 1332–1338. https://doi.org/10.1126/science.aab3050 Publisher: American Association for the Advancement of Science.

[62] Daniel Langr and Pavel Tvrdík. 2016. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (Feb. 2016), 428–440. https://doi.org/10.1109/TPDS.2015.2401575 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[63] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324. https://doi.org/10.1109/5.726791 Conference Name: Proceedings of the IEEE.

[64] Jure Leskovec, Kevin J. Lang, and Michael Mahoney. 2010. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web (WWW '10)*. Association for Computing Machinery, New York, NY, USA, 631–640. https://doi.org/10.1145/1772690.1772755

[65] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified tensor-program optimization via high-level scheduling rewrites. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 55:1–55:28. https://doi.org/10.1145/3498717

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

[67] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2018. Scalable linear algebra on a relational database system. *ACM SIGMOD Record* 47, 1 (2018), 24–31. Publisher: ACM New York, NY, USA.

[68] Tim Mattson, Timothy A Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. 2019. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 276–284.

[69] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems (RecSys '13)*. Association for Computing Machinery, New York, NY, USA, 165–172. https://doi.org/10.1145/2507157.2507163

[70] Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, and Simone Campanoni. 2024. Representing Data Collections in an SSA Form. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Edinburgh, United Kingdom, 308–321. https://doi.org/10.1109/CGO57630.2024.10444817

[71] Cleve Moler and Jack Little. 2020. A history of MATLAB. *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020), 81:1–81:67. https://doi.org/10.1145/3386331

[72] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (March 2018), 16:1–16:40. https://doi.org/10.1145/3180143

[73] Dianne P O'Leary. 2009. *Scientific computing with case studies.* SIAM.

[74] 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*, Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[75] Christos Psarras, Henrik Barthels, and Paolo Bientinesi. 2022. The Linear Algebra Mapping Problem. Current state of linear algebra languages and libraries. *ACM Trans. Math. Software* 48, 3 (Sept. 2022), 1–30. https://doi.org/10.1145/3549935 arXiv:1911.09421 [cs].

[76] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[77] Jason Rumengan, Terry Yue Zhuo, and Conrad Sanderson. 2021. PyArmadillo: a streamlined linear algebra library for Python. *Journal of Open Source Software* 6, 66 (2021), 3051. https://doi.org/10.21105/joss.03051

[78] Yousef Saad. 2003. *Iterative methods for sparse linear systems* (2nd ed.). SIAM, Philadelphia.

[79] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proceedings of the ACM on Management of Data* 1, 1 (May 2023), 37:1–37:27. https://doi.org/10.1145/3588717

[80] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 158:1–158:30. https://doi.org/10.1145/3428226

[81] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 89:1–89:33. https://doi.org/10.1145/3527333

[82] Jia Shi. 2020. Column Partition and Permutation for Run Length Encoding in Columnar Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2873–2874. https://doi.org/10.1145/3318464.3384413

[83] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS '15)*. IEEE Computer Society, Washington, DC, USA, 61–70. https://doi.org/10.1109/IPDPS.2015.27

[84] Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *arXiv:1512.00066 [cs]* (Nov. 2015). http://arxiv.org/abs/1512.00066 arXiv: 1512.00066.

[85] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 813–824. https://doi.org/10.1109/IPDPS.2013.112 ISSN: 1530-2075.

[86] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. 2013. SciDB: A database management system for applications with complex analytics. *Computing in Science & Engineering* 15, 3 (2013), 54–62. Publisher: IEEE.

[87] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (Nov. 2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721 Conference Name: Proceedings of the IEEE.

[88] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. Efficient Processing of Deep Neural Networks. *Synthesis Lectures on Computer Architecture* 15, 2 (June 2020), 1–341. https://doi.org/10.2200/S01004ED1V01Y202004CAC050 Publisher: Morgan & Claypool Publishers.

[89] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. 2022. TorchSparse: Efficient Point Cloud Inference Engine. *Proceedings of Machine Learning and Systems* 4 (April 2022), 302–315. https://proceedings.mlsys.org/paper_files/paper/2022/hash/c48e820389ae2420c1ad9d5856e1e41c-Abstract.html

[90] Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. 2023. Torchsparse++: Efficient training and inference framework for sparse convolution on gpus. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 225–239.

[91] Odysseas G Tsatalos, Marvin H Solomon, and Yannis E Ioannidis. 1996. The GMAP: A versatile tool for physical data independence. *The VLDB Journal* 5 (1996), 101–118. Publisher: Springer.

[92] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3 (June 2015), 14:1–14:33. https://doi.org/10.1145/2764454

[93] Todd Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. https://doi.org/10.5441/002/ICDT.2014.13

[94] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 3 (March 2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2 Bandiera_abtest: a Cc_license_type: cc_by Cg_type: Nature Research Journals Number: 3 Primary_atype: Reviews Publisher: Nature Publishing Group Subject_term: Biophysical chemistry;Computational biology and bioinformatics;Technology Subject_term_id: biophysical-chemistry;computational-biology-and-bioinformatics;technology.

[95] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. 2002. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 26–26. https://doi.org/10.1109/SC.2002.10025 ISSN: 1063-9535.

[96] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16 (Jan. 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071 Publisher: IOP Publishing.

[97] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. 2023. Unified Convolution Framework: A compiler-based approach to support sparse convolutions. *Proceedings of Machine Learning and Systems* 5 (March 2023), 666–679. https://proceedings.mlsys.org/paper_files/paper/2023/hash/ccf7262fb986e4367ccd3903960c57a0-Abstract-mlsys2023.html

[98] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: compiling distributed sparse tensor computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*. IEEE Press, Dallas, Texas, 1–15.

[99] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Implementing Push-Pull Efficiently in GraphBLAS. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3225058.3225122

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

[101] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 17–31.

[102] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 687–701. https://doi.org/10.1145/3445814.3446702

[103] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral specification and code generation of sparse tensor contraction with co-iteration. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2022), 1–26.

[104] Weijie Zhou, Yue Zhao, Xipeng Shen, and Wang Chen. 2020. Enabling Runtime SpMV Format Selection through an Overhead Conscious Method. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (Jan. 2020), 80–93. https://doi.org/10.1109/TPDS.2019.2932931 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

Received 2024-10-15; accepted 2025-02-18