Looplets: A Language for Structured Coiteration

Willow Ahrens  
MIT CSAIL  
Cambridge, MA, USA  
willow.ahrens@mit.edu

Daniel Donenfeld  
MIT CSAIL  
Cambridge, MA, USA  
danielbd@mit.edu

Fredrik Kjolstad  
Stanford University  
Stanford, CA, USA  
kjolstad@cs.stanford.edu

Saman Amarasinghe  
MIT CSAIL  
Cambridge, MA, USA  
saman@csail.mit.edu

Abstract
Real world arrays often contain underlying structure, such as sparsity, runs of repeated values, or symmetry. Specializing for structure yields significant speedups. But automatically generating efficient code for structured data is challenging, especially when arrays with different structure interact. We show how to abstract over array structures so that the compiler can generate code to coiterate over any combination of them. Our technique enables new array formats (such as 1DVBL for irregular clustered sparsity), new iteration strategies (such as galloping intersections), and new operations over structured data (such as concatenation or convolution).

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

Keywords: Coiteration, Array, Tensor, Compressed, Sparse

ACM Reference Format:

1 Introduction
Arrays (or tensors) are a powerful abstraction for representing collections of data. From scientific simulations to neural networks to image processing, array programming frameworks like NumPy, TensorFlow, and Halide help users productively process large amounts of data [2, 20, 45]. These complex frameworks are built on one of the simplest data structures in computer science: the dense array. A dense array stores every element contiguously in memory. Iterating through a dense array is as easy as incrementing a pointer in a loop.

We can greatly improve on the efficiency of dense array processing by taking advantage of the underlying regular or irregular structure often present in real-world data. Data may be sparse (mostly zero), symmetric (mirrored along the diagonal), or contain repeated values. When data is sparse, we need only store the nonzeros. When data is symmetric, we need only store one half. When data contains runs of repeated values, we need only store one value from each run. With current memory sizes, many datasets are impossible to store without these optimizations. Storage improvements lead to performance improvements when we can use mathematical properties to avoid redundant work. For example, in a sparse sum, we need only add the nonzero values.

In these representations, iteration is more complicated than dense iteration. For example, to efficiently iterate over sparse arrays, we must skip over the zeros. Several sparse formats are used in practice for different situations, such as sorted lists, hash tables, or bytemaps, and special implementations must be developed for each. The iteration problem compounds when we need to combine sparse arrays that are stored in different formats. If we wish to multiply two sparse arrays pointwise, we must determine which nonzero locations are shared. The coiteration logic is a performance bottleneck, so we need custom implementations for the combinatorial space of potential input formats.

The influential TACO framework [29] compiles specialized code for all combinations of dense and sparse arrays. Users can also add custom sparse formats by implementing TACO’s interface for iterators over nonzeros [12].

However, an iterator-over-nonzeros interface is not sufficient to express the full variety of underlying structures and iteration strategies encountered in real-world sparse arrays. Sparsity patterns may contain irregular clusters or blocks, or even regular shapes like banded or triangular matrices. When nonzeros are clustered, it is better to process the dense regions with simple dense code, rather than e.g. checking if each dense nonzero overlaps with nonzeros in other sparse arrays. TACO can model a dimension as entirely sparse (using the nonzero iterator interface) or entirely dense (using a random access interface), but cannot iterate over a single dimension as a combination of sparse and dense regions. TACO can artificially add dimensions to construct formats like BCSR (fixed-size blocks) or DIA (fixed number of diagonals), but these constructions cannot be composed unless both arrays share the same dimensional transform.
We must also move beyond mere irregular sparsity, to also express structure. Code that iterates over nonzeros does not sufficiently accelerate computations over symmetric arrays or arrays with repeated values, since these may not contain any zeros at all. Therefore, a myriad of new compilers or compiler passes have been developed in separate attempts to support different structural specializations, such as ragged arrays [18], symmetry [48], or run-length encoding [15]. These extensions represent significant implementation effort, but do not compose with one another.

In this paper, we show how to abstract over these iteration strategies using the concept of iterator protocols. An iterator protocol describes the interface that a structured array format must implement to be used in some particular implementation of an array operation. Protocols are abstract representations of structure within a sequence of values. The protocol declares which functions a format should implement, and how to interpret the data those functions produce. For example, the iterator-over-nonzeros protocol asks formats to implement iterators over coordinates and corresponding values, and declares that we should interpret the format as a vector which is zero except at the coordinates.

In short, dense array programming is comparatively easy to implement because it only needs to support one protocol: unstructured random access. The TACO compiler supports two protocols, random access and an iterator-over-nonzeros. Ragged arrays require a protocol for dense regions followed by zeros [18], and TACO’s run-length encoding extension supported a protocol for sequences of runs. The systems do not compose because each system hard-coded support for its respective protocol. As applications for array compilers become more diverse, we must support an ever-increasing number of protocols and the interactions between them. Hand-writing code for each situation is infeasible.

We introduce a language for iterator protocols. We express iterators over arrays using basic functional units called Looplets that expose underlying array structure in a customizable way. We then devise a compiler that can automatically generate efficient code for combinations of different input protocols and array expressions. We integrate the Looplet language into a new array compiler called Finch, which accelerates the design space exploration of sparse and structured loop nests. Finch can compile expressions over any combination of structured formats, protocols, mathematical simplifications, and index modifiers. Finch lowers expressions progressively, allowing us to express structural simplifications (like zero-annihilation from sparsity) through straightforward and extensible rewrite rules.

A composable protocol language makes it possible to express far more data representations than prior work [12] and, critically, to mix them freely in expressions. Protocols can be easily developed for new formats, such as the PackBITTS format, which intersperses runs of repeated values with dense regions of unstructured data and is standardized as part of the TIFF image format [1]. Different protocols can also be developed for the same format. For example, a list of nonzeros might be better traversed by skipping ahead with a binary search. If there are two lists, we might want one to lead and the other to follow, or perhaps allow each to skip ahead to nonzeros in the other, a “galloping” intersection strategy [6]. Galloping is used to implement worst-case-optimal joins in databases [43, 55]. Finally, protocols can be modified to implement more complex operations. We can express padding operations by nesting an existing protocol within another, or affine indexing by shifting an existing protocol. These protocol modifications enable operations like concatenation or convolution over structured formats.

We make the following contributions:

- We propose the use of access protocols to abstract over array iteration strategies and introduce the Looplet language to express access protocols.
- We describe a compiler for the Looplet language that combines multiple local protocols into one efficient loop nest that implements an array operation.
- We show how a surprising range of iteration strategies, formats and array operations can be expressed and composed using Looplets.

To evaluate our contributions, we benchmark Finch on a diverse set of operations; sparse matrix sparse vector multiply, triangle counting, sparse input convolution, alpha blending, and all-pairs image similarity. We compare to OpenCV and TACO, and our evaluation shows that Finch is competitive on the kernels that each supports, while using less code. Finch uses 7223 lines excluding comments, versus 34805 in TACO or 900998 in OpenCV. Different protocol perform better in different cases, emphasizing the need for flexibility in iteration strategies. In some cases, Finch obtains order-of-magnitude speedups over the state of the art.

## 2 Motivating Example

When there is only one input to an operation, such as map or reduce, it is possible to hand-write specialized implementations for each potential input data structure. However, when there are multiple inputs and multiple operations, we cannot easily hand-write for all the combinations. Even when both inputs are sparse, there is no universally efficient protocol. Here, we demonstrate how the iterator-over-nonzeros interface of systems like TACO [29] cannot express the appropriate coiteration over unstructured and clustered sparsity.

Consider the dot product kernel, which combines two vectors by summing their pairwise product. The dot product can be written as \[ C = \sum_i A_i B_i \]. When both vectors are dense, we compute it with a single for-loop. When both vectors are sparse, we might consider using TACO’s two-finger merge template, which represents both vectors as iterators-over-nonzeros, and merges the nonzero coordinates to find shared
nonzeros. However, this is not a good fit when one or both of the vectors are clustered.

Perhaps the quintessential sparse format is the sparse list format (referred to as “compressed” by Kjolstad et al. [29]), which is a good fit for unpredictable sparsity patterns. This format stores the locations of only nonzero values using a sorted list, which is a natural fit for an iterator-over-nonzeros interface. However, when nonzeros are clustered they benefit from more specialized formats. For instance, banded matrices are popular in scientific computing and contain dense regions of nonzeros centered around the diagonal. To represent such structures, we can introduce a sparse band format that stores a single, variably wide block of contiguous nonzeros. The left columns of Figures 1a–1c express coiteration between these sparsity structures in the compressed iterator-over-nonzeros model, and the result of inlining our interface definitions into a two-finger-merge dot-product template. The resulting code iterates over the nonzero values of both lists, stopping once either is exhausted.

The right columns of Figures 1a–1c express coiteration in the Looplets model. Pipeline, phase, stepper, spike, run and lookup are all Looplet types that will be explained in Section 3, but they describe the semi-structured sparsity of the data in Figure 1c. The increased expressiveness of the model lets us inform the compiler that there are large zero regions before and after the dense band. This lets us skip ahead in the sparse list to the start of the dense band region. Additionally, our protocol declares that the dense band can be randomly accessed. This allows us to skip the elements in the band that are zero in the list. These two optimizations can have asymptotic effects, which are visualized for our example vectors in Figure 1c.

## 3 Looplets Language

Looplets are abstract descriptions of regular or irregular patterns in a sequence of values, together with the code needed to iterate over the sequence. Looplets represent sequences using hierarchical descriptions of the values within each region or subregion. Regions are specified by their absolute starting and ending index, referred to together as an **extent**.

Looplets are lowered by a compiler. Values can be static (i.e., known at compile time), or dynamic (i.e., known only at runtime). Looplets represent the full sequence abstractly, even if all of the values are not stored explicitly. For example, a `run` looplet represents a sequence of many of the same value, usually stored once. A `lookup` looplet represents an arbitrary sequence as a function of the index. While this function is often an array access, it could also be a function call, like \( f(i) = \sin(\pi i/7) \).

Looplets are defined with respect to the extent of the **target region** that we wish to represent (usually the range of an enclosing loop). The **spike** looplet represents a sequence of values that are all the same, followed by a single scalar value at the end of the target region. Frequently, we will want to represent a subregion, or **truncation**, of a looplet. Many looplets are self similar. A run looplet, for example, can represent any subregion of itself. Other looplets are specific to the target region. For example, a truncation of a spike that excludes the last element produces a run.
Some looplets represent the composition of other looplets. The `switch` looplet represents a choice between different looplets under different conditions. The `pipeline` looplet represents a sequence of a few different looplets, one after the other. The `stepper` looplet represents a sequence of an unbounded number of identical looplets. These looplets describe not only their sublooplets, but also the conditions and extents in which sublooplet apply.

A looplet nest can introduce and manipulate its own runtime variables in the generated code. For example, the code to advance a stepper to the next looplet in a sequence might increment some counter variable. A switch looplet might use that variable in its condition. Looplets are executed in ascending index order, but some regions may be skipped. Therefore, variables introduced by a looplet are only guaranteed to be visible to child and descendant looplets, but not to looplets subsequently executed in the program.

Precise descriptions of all the looplets are given in Figure 2. Each looplet is described together with the functions that it must implement, which return either sublooplets or code that modifies the state. Because looplets are defined with respect to the region that contains them, they need only declare internal boundaries that affect child looplets.

4 Formats

Array structures are diverse, and so are approaches for array storage. Prior approaches, such as TACO level formats [12] and the FiberTree abstraction [52], popularized hierarchical taxonomies for array structure. These abstractions decompose multidimensional arrays mode-by-mode into trees, where each mode is a level of the tree, and each node within a level is a slice. For example, the popular CSR matrix format stores an outer dense level of rows, where each inner level is a sparse list of nonzero columns.

It is helpful to view an array `A` as a function mapping several indices to elements. If we were to curry our array function, then a partial application corresponds to a slice of the array. For example, if `A` were a 3-dimensional, then

\[ A(i, l_2, l_3) = A(l_1)(l_2)(l_3) \quad \text{and} \quad A(l_1) = A[i_1, :, :] . \]

In this paper, we define a `fiber` as an array function mapping a single index to a subfiber. Fibers can be thought of as abstract vectors of subfibers. The `level storage` is a datastructure responsible for storing all the fibers within a dimension, each distinguished by an integer `position`.

Looplets make it easy to efficiently support new level formats. The array tree abstraction decomposes multidimensional array formats into unidimensional ones. Looplets further decompose the remaining unidimensional structure. The format developer can use Looplets to describe the structure of a single fiber within a level. In Section 6, we show how the looplet nest is then automatically merged with other nests and lowered to efficient code. We refer to the process of constructing a looplet nest for a fiber as `unfurling`.

![Figure 2](https://example.com/figure2.png)

**Figure 2.** The looplets considered in this paper, described and displayed with a target extent of `i:j`.

By defining just a few level formats, we can express a wide variety of structured array storage formats and protocols. Figure 3 gives some examples of matrix structures and protocols. Figure 3 gives some examples of matrix structures and protocols that allow us to efficiently traverse them.

Our approach relaxes the constraints of prior work on array structure. Rather than force array datastructures to
Looplets: A Language for Structured Coiteration  
CGO ’23, February 25 – March 1, 2023, Montréal, QC, Canada

**Figure 3.** A variety of example structures, corresponding level formats, and protocols expressed as looplet nests. Matrices are row major, and outer levels are dense. The row under consideration is highlighted in red.
express a set of points that are nonzero, our format can express any structured function from an index or indices to a position or value. The hierarchical level format approach is enabled by the Looplets language, but not required. Other approaches might also be implemented with Looplets.

5 Extended Concrete Index Notation

Concrete index notation (CIN) is a high level language for array computations. CIN was first introduced as part of the TACO compiler [28]. It was later extended to include multi statements (to express multiple outputs) [30] and protocol declarations (to customize the access protocol independently from the array format) [3].

5.1 Concrete Index Notation Background

Our new grammar for CIN is shown in Figure 4. At the heart of CIN is the assignment statement, which represents an update to a single array element. We can either increment or overwrite the element by a pointwise expression composed of functions over other array access expressions. Arrays are accessed by index variables. The forall statement repeats the assignment statements for each value of an index variable. The where statement lets us compute an array in one statement, then use it in the other. The producer (right hand) side computes arrays for the consumer (left hand) side to use. This gives rise to the notion of array results. Each statement in concrete index notation returns one or more results. The result of an assignment is the array being modified, and the result of a forall is the result of its body. The result of a where statement is the result of its consumer. Result arrays are initialized as soon as they enter the scope, and are finalized when they leave scope. Thus, arrays are initialized in the outermost where statement which contains them on the right hand side, or at the start of the program. The multi statement allows us to compute multiple outputs at once, and it combines the results of its constituent statements.

5.2 Concrete Index Notation Extensions

We extend CIN with additional constructs to express looplets. We allow arrays to be accessed by any expression, not just indices. Critically, index expressions allow us to specify different protocols for accessing index variables. Instead of returning a fixed looplet nest, users can specify the kind of nest that should be used. For example, a user might choose between random-access, iterating over nonzeros, or galloping over some index variable. Index expressions also enable index modifiers, which affect the protocol of the index they modify. Users might declare that they should iterate over a slice of an array, or perhaps shift the index by some expression. If an index expression is opaque to the compiler, it represents a random access. The sieve statement guards another statement and only executes it when the condition is true. We support random access by introducing a loop over all possible access locations and using a sieve to only execute the single iteration corresponding to the target location. During lowering, our code generator may need to introduce pass statements, which are no-ops that do nothing other than remember which array outputs they aren’t writing to. Pass statements return their arrays unmodified.

6 Lowering

This section describes our looplet merging and lowering algorithm. Our compiler, Finch, lowers concrete index notation recursively, node by node, emitting code until a forall is reached. To lower a forall, Finch unfurls all arrays accessed at the outermost level by the forall index, transforming them into looplets. In our progressive lowering approach, each type of looplet has a corresponding compiler pass that is capable of evaluating it. Many lowerers produce subexpressions that are lowered recursively. We lower whichever looplets we can at each step. Unlowered looplets are truncated or ignored for later stages, as applicable.

6.1 Looplet Lowerers

Since looplets are defined with respect to a forall statement, its index, and its bounds, each looplet lowerer operates on a forall loop expression. The looplets themselves are treated as vectors being accessed in the body of the forall loop.

Lookups. The simplest way to evaluate a forall loop is to execute each iteration in turn, and evaluate the body of the loop after binding the index value. If all of the looplets in the body are lookups, dynamic values, or scalars with respect to the loop index, Finch emits a for-loop and lower the body after substituting a dynamic index value:

```
A = Lookup(body())[j^2]
B = Lookup(body())[data[j]]
ifinch[0] i in 1:1 (C[i] := 2 * x * A[i] + B(i))
for _j = 1:1
  @finch(C[j] := 2 * x * $(i_1^2) + $(data[i_1]))
end
```

Figure 4. Our extended concrete index notation grammar. Here, value is used for values (like arrays) in the scope surrounding the program, and $(value)$ is an escape sequence. The token @finch is used to denote a CIN program execution within a larger piece of code.
When one array is zero within a region, implementations
Thus, when spikes are truncated by other looplet passes,
they produce a cases statement depending on whether the
one, Finch skips the loop and just evaluates the body once.

Two subexpressions, one for the spike bodies (runs of variable
compiler passes over intermediate expressions.
encoded data. Users can even write their own simplifying
constants is useful for optimizing operations over run-length-
cents within the expression. For example, Finch recognizes
or constant propagation, some rules might operate on state-
broadens the accessibility of our system. Users might ex-
rewrite rules. Using rewrite rules to express optimizations
subregion, simplifying optimizations can be expressed as

Because Finch lowers separate looplet expressions for each
subregion, simplifying optimizations can be expressed as
rewrite rules. Using rewrite rules to express optimizations
broadens the accessibility of our system. Users might ex-
press arbitrary rewrites for the interaction between custom
value types and custom functions, such as semirings or be-
yond \[14, 26, 40\]. Figure 5 gives some examples of the kinds of
rules we use. In addition to simple rules like zero-annihilation
or constant propagation, some rules might operate on state-
ments within the expression. For example, Finch recognizes
that adding a constant \(c\) to the same output \(n\) times is equi-
valent to adding \(c \cdot n\), saving \(O(n)\) work. Removing loops over
constants is useful for optimizing operations over run-length-
encoded data. Users can even write their own simplifying
compiler passes over intermediate expressions.

\[
\begin{align*}
(a, \ldots, (b, \ldots, c, \ldots)) & \Rightarrow (a, \ldots, b, \ldots, c, \ldots), \\
(a, \ldots, 0, b, \ldots) & \Rightarrow (a, \ldots, b, \ldots), \\
0 & \Rightarrow 0 \Rightarrow 0 = 0 \\
\end{align*}
\]

Figure 5. A selection of rewrite rules used in Finch to de-
clare mathematical properties enabling sparse and structural
optimizations. Users can add custom rules for the kinds of
computations in their domain.

**Runs and Rewriting.** The run lowerer unwraps runs
into their scalar values and simplifies the resulting program.
For example, sparse computing relies on the fact that
\(x = 0\). When one array is zero within a region, implementa-
tions can ignore arrays it multiplies.

Because Finch lowers separate looplet expressions for each
subregion, simplifying optimizations can be expressed as
rewrite rules. Using rewrite rules to express optimizations
broadens the accessibility of our system. Users might ex-
press arbitrary rewrites for the interaction between custom
value types and custom functions, such as semirings or be-
yond \[14, 26, 40\]. Figure 5 gives some examples of the kinds of
rules we use. In addition to simple rules like zero-annihilation
or constant propagation, some rules might operate on state-
ments within the expression. For example, Finch recognizes
that adding a constant \(c\) to the same output \(n\) times is equi-
valent to adding \(c \cdot n\), saving \(O(n)\) work. Removing loops over
constants is useful for optimizing operations over run-length-
encoded data. Users can even write their own simplifying
compiler passes over intermediate expressions.

\[
\begin{align*}
A & \Rightarrow Run(body = x) \\
B & \Rightarrow Run(body = 0) \\
@finch((OV \ i \in \ start:stop CC) \Rightarrow A[I] + B[I])) \\
@finch((OV \ i \in \ start:stop CC) \Rightarrow 0 + 3x) \\
@finch((OV \ i \in \ start:stop pass C) \\
@finch((OV \ i \in \ start:stop pass C) \\
@finch((OV \ i \in \ start:stop pass C) \\

**Spikes.** Given an expression with spikes, Finch constructs
two subexpressions, one for the spike bodies (runs of variable
length) and one for the tails (of length one). Finch truncates
other looplets to match each case. When a loop has length
one, Finch skips the loop and just evaluates the body once.

Recall that spike looplets depend on the target region.
Thus, when spikes are truncated by other looplet passes,
they produce a cases statement depending on whether the
new target range includes the final tail element. If that last
element is included, the subrange still represents a spike.
Otherwise, it is simplified to a run:

\[
\begin{align*}
A & \Rightarrow Spike(body = 0, tail(I) = Adata[I]) \\
B & \Rightarrow Spike(body = 0, tail(I) = Bdata[I]) \\
@finch(OV \ i \in \ start:stop CC) \Rightarrow A[I] + B[I]) \\
\end{align*}
\]

**Switches.** The switch lowerer produces a separate expres-
sion for each combination of phases from the switch looplets
in an expression. Each combination is lowered separately
and emitted in an if-else-block:

\[
\begin{align*}
A & \Rightarrow Switch( \\
\text{Case} & \Rightarrow (x > 1), body = 1), \\
\text{Case} & \Rightarrow (true), body = 2), \\
\end{align*}
\]

**Pipelines.** The pipeline lowerer produces a separate expres-
sion for each combination of phases from the pipeline
looplets in an expression. The ranges of all the phases in each
combination are intersected, and other looplets are truncated
to match. Note that many of these combinations will have an
empty intersection. Consider the graph where phase combi-
nations are nodes and edges represent transitions from one
phase to the next within a pipeline. If we lower combination-
spired phases in an order that linearizes the graph, then
earlier phase combinations will always be executed before
later ones. Since each edge advances a pipeline, the graph is
acyclic and we can construct such an order:

\[
\begin{align*}
A & \Rightarrow Pipeline( \\
\text{Phase} & \Rightarrow (s_A, body = 1), \\
\text{Phase} & \Rightarrow (body = 2), \\
\text{Phase} & \Rightarrow (s_A, body = 1), \\
\text{Phase} & \Rightarrow (s_B, body = 3), \\
\text{Phase} & \Rightarrow (true), body = 4), \\
\text{Phase} & \Rightarrow (O(V \ i \in \ start:stop CC) \Rightarrow A[I] = B[I])) \\
\text{Phase} & \Rightarrow (O(V \ i \in \ start:stop CC) \Rightarrow 0 + 3x)) \\
\text{Phase} & \Rightarrow (O(V \ i \in \ start:stop pass C) \\
\text{Phase} & \Rightarrow (O(V \ i \in \ start:stop pass C) \\
\text{Phase} & \Rightarrow (O(V \ i \in \ start:stop pass C) \\

**Steppers.** Steppers represent an arbitrary number of child
looplets. Finch first uses the stepper seek function to set each
stepper’s current child to intersect with the starting index
of the current target region. Finch then uses a while loop
to lower steppers, with each step evaluating as large a range
as possible without crossing any stepper’s child boundaries.
At the beginning of the loop body, Finch computes the target
region by intersecting the extent of each steppers’ current
child. Then, Finch truncates the stepper children to the computed region, producing the loop body expression of the step. Finally, Finch emits the next statements from each stepper, which are responsible for advancing the state of the child looplet to represent the next child, if necessary.

Because the while loop maintains the current starting index for the step, only the ending index is needed from each stepper. This single index is the stride. The seek function often contains a binary search, and the next function usually increments a variable, but they can be more general if needed:

```
a = Stepper(stride = idx[p], body = Run(val[p]), next = p += 1)
b = Stepper(stride = jdx[q], body = Run(wal[q]), next = q += 1)
p = q = 1
#finch(∀ i ∈ start:stop (C[:,i] += A[i] * B[i]))
⇓
p = q = 1
step = start
while step < stop
  stride = min(idx[p], jdx[q])
  #finch(∀ i ∈ start:stride (C[:,i] += val[p] * wal[q]))
  p = stride = idx[p]
  q = stride = jdx[q]
  step = stride
end
```

**Jumpers.** Finch lowers jumpers similarly to steppers. Instead of using the smallest declared extent, jumpers use the largest. Whether the smallest or largest extent is chosen, it will correspond to the exact range of at least one child looplet, which can be lowered verbatim. Choosing the largest extent enables powerful optimizations. If the largest looplet is a run of zeros, multiple child looplets that it multiplies can be skipped. The body of a jumper should be able to process more than one child looplet, but usually includes a switch that specializes to the case where only a single child is needed.

As an example, a jumper over spikes usually declares the length of the current spike but, if another spike is longer, processes multiple spikes with a stepper. The jumper allows us to implement leader-follower or galloping intersections:

```
a = Jumper(stride = idx[p], body = Run(val[p]), next = p += 1)
b = Jumper(stride = jdx[q], body = Run(wal[q]), next = q += 1)
p = q = 1
#finch(∀ i ∈ start:stop (C[:,i] += A[i] * B[i]))
⇓
p = q = 1
step = start
while step < stop
  stride = max(idx[p], jdx[q])
  #finch(∀ i ∈ start:stride (C[:,i] += val[p] * wal[q]))
  p = stride = idx[p]
  q = stride = jdx[q]
  step = stride
end
```

**Shifts.** Shift looplets do not need a special compiler pass, but instead shift the declared extents of their arguments during other passes. Any looplet such as a run, scalar, or spike which results in a terminal scalar value with respect to the loop index can safely discard the shift looplet wrapper.

```
6.2 Choosing Lowerers

The same expression may contain several different kinds of looplets, each lowered with a corresponding compiler pass. When many different lowerers are needed, we use pairwise tiebreaking rules. If looplets contain other looplets, the outer looplet needs to be lowered before the looplets it contains. Thus, we can restrict our attention to the outermost looplets when picking a compiler pass.

A tiebreaking rule between two lowerers asserts that the winner can handle any program (or looplets) the loser could handle. Our tiebreaking rules also represent a heuristic for the order in which looplets should be lowered. Finch chooses lowering passes in the following order of descending priority:

```
Switch > Run > Spike > Pipeline > Jumper > Stepper > Lookup
```

Our reasoning is as follows: We always lower switch looplets first in order to examine their cases. We lower runs and spikes whenever we see them, to simplify expressions as early as possible. Then, we lower pipelines before the looping constructs to hoist the control flow outside of loops. We lower Jumpers before steppers to give them leader privileges. Finally, if our expression is just lookups, there’s nothing left to do but emit a simple for-loop. As future work, we plan to investigate modifying the looplet lowering order by giving some looplets customizable numeric priorities.

7 Index Modifiers

Looplets enable new functionality previously unsupported by sparse array compilers. We show how the combination of few simple *index modifiers* can be combined to implement kernels like concatenation and convolution over sparse inputs. These index modifiers change the behavior of their corresponding mode by wrapping or modifying the looplets that mode would unfurl into.

As an example, consider the special windowing array window(i, j)[k] = i+k−1 with dimension 1:j−1. A window can be used to represent a slice of an input array. As such, A[window(3,5)[k]] would behave like the slice A[3:5][k]. We can construct a protocol for A[window(i,j)[k]] as

```
Shift(delta=i, body=truncate(unfurl(A), 1:j))
```

Letting offset(i)[j] = j−1 be a special array that shifts the dimension of the parent, we can construct the protocol

```
Shift(delta=i, body=unfurl(A))
```

Finally, we can also introduce a padding array, which allows out-of-bounds access. We will use the special Julia value missing, which is used to denote missing data and propagates. For example, we have A[missing] = missing and f(x, missing) = missing. The function coalesce returns it’s first non-missing argument. When i is in bounds, we define permit[i] = i, but when i is out of bounds, we define permit[i] = missing. Our protocol for A[permit[i]] is

```
Pipeline(Phase(stride=0, Run(missing)),
  Phase(stride=length(A), body=unfurl(A)),
  Phase(Run(missing)))
```

Pipe b
We implemented was very sparse (exactly 10 nonzeros), the follower protocol (10% nonzeros), a leader protocol for $A$. “Sparse” corresponds to a standard two-finger merge. Very sparse, VBL led to large speedups over TACO. Evaluation

When $x$ frequently contains dense blocks or bands, we also tried our matrices are from the scientific computing domain, and adaptively combine the benefits of leading and following. Together, these primitives greatly expand the range of functionality that the concrete index notation can express. For example, we can concatenate $A$ and $B$ to produce $C$ with:

$$\forall i \ 1[i] = \text{coalesce}(A[\text{permit}[i]], B[\text{permit}[\text{size}(A)][i]])$$

Similarly, we can express one-dimensional convolution with a vector $A$ and a length-3 filter $F$ to produce $B$ with:

$$\forall i \ j \ B[i] = \text{coalesce}(A[\text{permit}[i]], 0) * F[\text{permit}[j]]$$

8 Evaluation

We implemented Finch in Julia. All timings are the minimum of at least 10,000 runs or 5 seconds of measurement. We used a single processor of an Intel® Xeon® CPU E5-2680 v3 running at 2.50GHz with AVX2 instructions and 32KB of L1 cache. Both TACO and Finch used 32-bit integer indices.

8.1 SpMSpV

Many of the new functionalities introduced by Finch involve coiteration between two structured operands. To emphasize the effects of different coiteration strategies, we begin with a comparison between SpMSpV approaches in Figure 6. Our kernel was $\forall i \ j \ y_{\text{ref}}[i] = A[i] * x[j]$. We iterate over $j$ in the inner loop to test coiteration capabilities, repeatedly merging $x$ with every row of $A$. We tested on matrices in the Harwell-Boeing collection [16]. When $x$ was more dense (10% nonzeros), a leader protocol for $A$ performed well, as it visited each element of $A$ and fast-forwarded $x$. When $x$ was very sparse (exactly 10 nonzeros), the follower protocol for $A$ performed better. A galloping approach was able to adaptively combine the benefits of leading and following. Our matrices are from the scientific computing domain, and frequently contain dense blocks or bands, so we also tried the VBL format. VBL processes the index of each band, rather than the index of each element in each band. When $x$ was very sparse, VBL led to large speedups over TACO.

8.2 Triangle Counting

Galloping intersections can greatly accelerate the triangle counting kernel, where no loop ordering can avoid an intersection in an inner loop and operands often have unpredictable power-law sparsity distributions of nonzeros within the rows and columns. Our kernel is

$$\forall i \ j \ k \ \forall \ A[i, j] = \text{round}(\text{UInt8}, \alpha * A[i, j] + \beta * C[i, j])$$

where we use a binary search to seek to the start of each considered window of $A$. The binary search allows our kernel to scale linearly with sparsity of the input. We consider zero-padded arrays, but we could express other padding schemes, such as circular padding by adding copies of $A$ on each side of the original. Our sparse implementation begins to outperform the dense one at around 5% sparsity, and results in a 7.8x speedup at 1% sparsity.

8.3 Convolution

Protocols enable new sparse functionality. Figure 8 compared our sparse convolution kernel to OpenCV. Our Finch kernel for a masked 11x11 convolution was

$$\forall i \ j \ k \ A[i, j] = \text{coalesce}(A[\text{permit}[i]], \text{coalesce}(A[\text{permit}[i]], B[\text{permit}[i]]), 0) * F[\text{permit}[j]]$$

8.4 Alpha Blending

Finch is competitive with frameworks that move beyond sparsity. Figure 9 compares against both OpenCV and TACO’s RLE extensions [15] for an alpha blending kernel

$$\forall i \ j \ A[i, j] = \text{round}((\text{UInt8}, \alpha * A[i, j] + \beta * C[i, j])$$

The Omniglot dataset has $105 \times 105$ images with grayscale handwritten characters from 50 different languages [35]. The Humansketches dataset has $1111 \times 1111$ images with hand

---

1 https://github.com/willow-ahrens/Finch.jl
drawn grayscale sketches [17]. Finch’s RLE format is competitive on both datasets, though there was not enough structure in Omniglot for an RLE approach to outperform OpenCV. OpenCV represents each 256-valued pixel as a single byte, and can process 32 pixels at once in an AVX register.

**Figure 9.** We compare TACO’s prototype RLE extensions [15] with Finch’s sparse and RLE approaches on an alpha blending task. Results are the mean of 10 images.

### 8.5 All-Pairs Image Similarity

Finch enables unique optimizations. Figure 10 considers the effectiveness of different strategies on an all-pairs image similarity kernel. We consider only pairwise comparisons, rather than batch approaches, to focus on coiteration and maintain relevance to k-means clustering methods. The kernel is

\[
\forall k \quad R[k] \leftarrow A[k, 1] \wedge \forall k \quad k \quad (R[k] = \sqrt{R[k] + R[l] - 2 \times o[l]}) \quad \text{where} \quad \forall k \quad 1 \quad o[l] \leftarrow A[k, 1] \wedge \forall k \quad k \quad A[k, 1] \wedge \forall k \quad 1 \quad A[l, 1])
\]

Where \( A \) contains linearized images in each row. The MNIST and EMNIST datasets contains 28x28 images of handwritten digits and letters [13, 36]. Results compute distances between 256 images. Finch’s VBL format can take advantage of the white background and clustered nonzeros in most of these images. However, the Omniglot dataset has noisier backgrounds which are better captured by run-length-encoding. Additionally, when both accesses to \( A \) contain a run, we can apply the last rule of Figure 5 to sum the whole run at once. As with alpha blending, the images were not sparse enough for these approaches to beat vectorized OpenCV. Future work might investigate whether quantization induces more structure, or whether our approach can produce vectorized code.

**Figure 10.** Speedups on all pairs image similarity.

### 9 Related Work

Dense array frameworks like NumPy [20], BLAS [11], and APL [25] greatly influenced subsequent language design.

The Halide compiler popularized array optimization through the use of scheduling commands, or semantics-preserving program transformations [38, 45, 46]. Implementing a scheduling language in Finch is future work.

Array compilers that support irregular sparsity include TACO [28, 29], MT1 [7, 9, 10], MLIR [8], COMET [54], Etch [34], SIPR [44], Tiramisu [5] and CHiLL-I/E [51].

Some approaches specialize for particular sparsity patterns. The OSKI library [56] includes specializations for block sparsity, and the BLAS for triangular matrices. TACO supports fixed-size blocks and bands [12]. TESA specializes for bitmap or quantized sparsity [60], CORA for ragged arrays [18], and FIDIL or TAICHI for spatial decompositions [22, 24]. SparseTIR combines multiple formats [58]. Zhao et. al. support coiteration for conjuctive \( * \) leader-follower loops over a wide variety of formats, but do not support disjunction \( + \) or mutual lookahead (galloping) coiterations [59].

Most sparse frameworks support only numeric types and the \( + \) or \( * \) operators, but TACO [21], GraphBLAS [14, 26, 40], and Cyclops [49] support arbitrary element types and operators, making them more productive programming models.

Previously mentioned sparse compilers consider sparse arrays as a set of nonzeros to be processed, precluding other compression-based optimizations. Compilers like StreamIt [53] and an extension to TACO [15] support direct computation on losslessly compressed datasets. The BLAS and Cyclops framework both optimize for dense symmetry [11, 50].

Many approaches model sparse computation with database queries, including the Bernoulli compiler [31–33] and a TACO autoscheduler [3]. Run-length encoding is a popular format for column stores [41]. Queries can be modeled and optimized as iterators [42].

In functional programming, stream fusion is a related technique which can fuse lazily constructed streams [27, 39].

Several sparse compilers have been extended to better adapt computation to multicore, GPU, or distributed architectures [5, 47, 49, 57]. Hsu et. al. investigate sparse compilation for spatial-dataflow accelerators [23]. Future work includes targeting new architectures with Looplets.

### 10 Conclusion

Historically, specializations for the array structures in different fields have been handled separately. For example, scientific computing explored block sparsity, image processing explored compression techniques, and databases explored worst-case optimal joins. Our work takes a step towards unifying these techniques across disciplines, providing opportunities to transfer optimizations and better specialize to data with heterogeneous structures.

### Acknowledgments

This work was supported by NSF Grant IIP-2044424 and the Applications Driving Architectures (ADA) Center, a JUMP Center cosponsored by SRC and DARPA.
A Finch CGO 2023 Artifact

A.1 Abstract
This artifact has two scripts that build and run all of the experiments described in the paper. We require an x86 host, and runs Debian bullseye. The artifact is implemented as a Julia package Finch.jl that implements the looplets described in the paper, and a set of benchmark scripts to run the benchmarks described in the paper. The specific runtimes measured depend on the host computer, however we expect the relative performance between Finch, TACO and OpenCV to roughly match the results presented in the paper.

A.2 Artifact check-list (meta-information)
- **Operating System**: We tested on Ubuntu 18 and MacOS 12.5, but we expect it to work on most modern mac or Linux distributions.
- **Compilation**: This artifact compares to two versions of TACO which are included as source code and built. Additionally, we build OpenCV and several wrapper scripts to call taco.
- **Data set**: We use matrices from the Harwell-Boeing collection, graphs from the SNAP network dataset collection, and several image datasets, MNIST, EMNIST, Omniglot, and Humansketches (these are automatically downloaded).
- **Software**: The artifact requires CMake (we used 3.10), gcc/g++ (we used 7.5), Python (we used 2.7/3.9), and git (we used 2.17). We used Julia 1.8.2 (which is automatically downloaded). We have several Julia dependencies whose exact versions are recorded in the Manifest.toml file (these are also automatically downloaded).
- **Hardware**: The artifact requires x86 processor.
- **Execution**: The experiments should be ran single threaded pinned to a single socket. Turbo Boost was turned off to avoid thermal throttling. The experiments from the paper take 2 hours in total.
- **Metrics**: Execution time is reported. All timings are the minimum of at least 10,000 runs or 5 seconds of measurement.
- **Output**: The output is long format data including the execution time stored in JSON files. We also provide scripts to plot the data in the same format as the paper.
- **How much disk space required (approximately)?** 4GB
- **Publicly available?**: Yes
- **Code licenses (if publicly available)?**: The code has been released under the MIT license.
- **Archived?**: After evaluation, we plan to distribute our repository on Zenodo. All of the datasets we used are publicly accessible with considerations for access in perpetuity.

A.3 How Delivered
Our artifact is distributed by a direct download [4] or by cloning the repository from GitHub, using the following command:

```bash
git clone -b cgo23-artifact https://github.com/willowahrens/FinchBenchmarks.git
```

Notice that we use the cgo23-artifact branch of the repository.

A.4 Installation
1. First install CMake, gcc, g++, Python, Python3, and git. Results were originally collected using gcc 7.5.0, however a more modern version should work. The gcc compiler only affects TACO and OpenCV, not Finch.
2. Run the build.sh script to download Julia 1.8.2 and build OpenCV and TACO (this uses the included Makefile).

A.5 Experiment Workflow
There are five experiments described in the paper in sections 9.1 through 9.5, with associated scripts to collect and analyze results. You can run all these commands with the run.sh script. This script runs all of the experiments and generates the graphs shown in the paper. Running the data collection scripts automatically downloads the appropriate datasets.

The experiments are named as follows: - all_pairs (for all-pairs image similarity) - alpha (for alpha blending) - conv (for sparse convolution) - smpspv (for sparse matrix sparse multiply) - triangle (for triangle counting)

Each experiment has several associated scripts that are all prefixed by it’s name. We use alpha as an example.

**alpha.jl** is a Julia script that runs the experiments. It can be invoked as

```julia
julia --project=. alpha.jl RESULT_FILE
```

where `RESULT_FILE` is the name of the JSON output in long format.

**alpha.sh** is a bash script that runs **alpha.jl** after setting appropriate environment variables to keep dataset and julia package downloads inside the toplevel directory. This script produces **alpha_results.json**, the results.

**alpha_plot.jl** is a Julia script that reads the results file and generates plots. It can be invoked as

```julia
julia --project=. alpha_plot.jl RESULT_FILE
```

where `RESULT_FILE` is the name of the output plot (with a .png extension).

**alpha_plot.sh** is a bash script that runs **alpha.jl** after setting similar appropriate environment variables. This script produces **alpha_plot.png**, the plot of the results you just collected.

**alpha_results_reference.json** contains the results we used to generate the plots for the paper. You can point the plotting scripts at this file to reproduce the results in the
paper exactly (you could also just look at the plots in the paper).

There are separate binaries for the OpenCV and TACO experiments, which are called from the Julia files with the appropriate environment variables set.

A.5.1 Short Lists. To save time, spmspv.jl and triangle.jl run a subset of the results in the paper consisting of the 5 smallest and 5 best performing datasets. To run the full versions, pass long as an additional final argument to these scripts.

A.6 Evaluation and Expected Result

Running the artifact as described above produces the raw execution timing, and the relative performance plotted in graphs which match the paper. You can verify that each experiment matches it’s corresponding figure in the paper. We also include reference results which were used to generate the figures in the paper.

A.7 Experiment Customization

It is possible to customize the benchmark scripts to use additional datasets. The matrices used in SpMSpV and Triangle counting are downloaded using the MatrixDepot Julia package. The main functions of these scripts can be modified to use a different list of matrix names recognized by MatrixDepot.

Finch uses the RewriteTools package to express rewrite rules, which can be added to enable simplifications on user-defined types. RewriteTools is an adaptation of the SymbolicUtils package [19].

The alpha.jl and all_pairs.jl datasets use images downloaded using the TensorDepot Julia package. Other datasets included in this package can be used, as long as they are a 3-tensor with the first index used as the index of the image, and the next two indices represent the rows and column of the image. The permutedims function can be used to permute the dimensions if they do not match. Other datasets can be added to TensorDepot for easy integration into the test harness, or they can be downloaded directly.

Finch can also be used as a standalone sparse tensor compiler. More details and documentation is available at https://github.com/willow-ahrens/Finch.jl. Very briefly, any @finch macro expression in the benchmark suite is calling the Finch compiler and using Looplets to compile sparse kernels. The @fiber macro sets up level formats with the abbreviations of d for a dense level, s1 for a sparse list of nonzeros, sv for a VBL level, and rl for an RLE level.

References

Looplets: A Language for Structured Coiteration


Received 2022-09-02; accepted 2022-11-07