

Finch: Sparse and Structured Array 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, arrays have revolutionized how we express computation. However, arrays in these, and almost all prominent systems, can only handle dense rectilinear integer grids. Real world arrays 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 array 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 arrays 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 array 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, graph analytics, and a high-level tensor operator fusion interface.

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 Array, Structured Array, Control Flow, Programming Language

1 INTRODUCTION

Arrays are the most fundamental abstraction in computer science. Arrays and lists are often the first-taught datastructure [4, Chapter 2.2], [55, Chapter 2.2]. Arrays are also universal across programming languages, from their introduction in Fortran in 1957 to present-day languages like Python [10], keeping more-or-less the same semantics. Modern array programming languages such as NumPy [44], SciPy [88], MatLab [66], TensorFlow [2], PyTorch [68], and Halide [70] have 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

Authors' addresses: 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, rrpate@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.

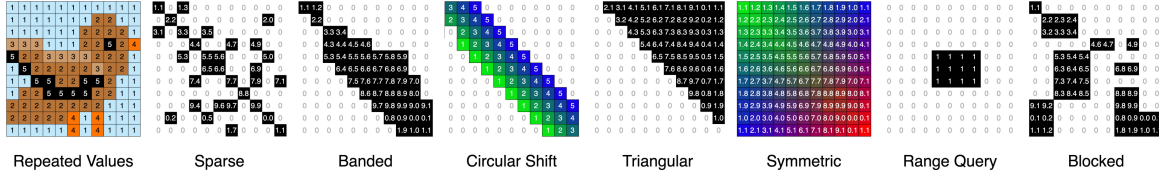


Fig. 1. A few examples of matrix structures arising in practice

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, the world is not all dense.

Our world is full of structured data.

Sparse arrays (which store only nonzero elements) describe networks, databases, and simulations [5, 12, 15, 64]. Run-length encoding describes images and masks, geometry, and databases (such as a list of transactions with the date field all the same) [39, 76]. 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) [22, 67, 72]. In the context of machine learning, combinations of sparse and blocked matrices are increasingly under consideration [27]. Even complex operators can be expressed as structured arrays. For example, a convolution with a filter can be expressed as a matrix multiplication with the Toeplitz matrix of all the circular shifts of the filter [82].

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 flexibility**. Hand-written solutions are collected in diverse libraries like MKL, OpenCV, LAPACK or SciPy [9, 20, 69, 88]. However, libraries will only ever support a subset of programs on a subset of data structure combinations. Even the most advanced libraries, such as the GraphBLAS, which support a wide variety of sparse operations over various semi-rings always lack support for other features, such as tensors, fused outputs, or runs of repeated values [21, 63]. While dense array compilers support an enormous variety of program constructs like early break and multiple left hand sides, they only support dense arrays [41, 70]. Special-purpose compilers like TACO [53], Taichi [47], StructTensor [38], or CoRa [34] which support a select subset of structured data structures (only sparse, or only ragged arrays) must compromise by greatly constraining the classes of programs which they support, such as tensor contractions. This trade-off is visualized in Tables 1 and 2.

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 [54]. The polyhedral model allows various compilers to represent dense computations on affine regions [41]. Taichi enriches single static

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

Table 1. Control flow support across various tools.

Feature / Tool	Halide	Taco	Cora	Taichi	Stur	Finch
Dense	✓	✓	✓	✓	✓	✓
Padded	✓					✓
One Sparse Operand		✓		✓		✓
Multiple Sparse Operands		✓				✓
Run-length						✓
Symmetric					✓	✓
Regular Sparse Blocks		✓				✓
Irregular Sparse Blocks						✓
Ragged			✓			✓

Table 2. Data structure support across various tools. Finch supports **both** complex programs and complex data structures.

assignment form with a specialized instruction for accessing only a single sparse structure, but it supports more control flow [47]. These systems tightly couple their control flow to narrow classes of data structures to avoid the challenges that occur when we intersect complex control flow with structured data. There are two challenges:

Optimizations are specific to the indirection and patterns in data structures: These structures break the simple mapping between array elements and where they are stored in memory. For example, sparse arrays store lists of which coordinates are nonzero, whereas run-length-encoded arrays 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, 97]. 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 arrays 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 array 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, a programmer might explore different ways to intersect only the even integers of two lists (represented as sparse vectors with sorted indices). 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]
  end
end

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

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

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

```

1.1 Contributions

- (1) More complex array 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) [16, 26, 47, 81] feature a flexible structure description, our level abstraction is more capable and extensible because it uses Looplets [7] to express the structure of each level.
- (2) A rich structured array programming language with for-loops and complex control flow constructs at the same level of productivity of dense arrays. To our knowledge, the Finch programming language is the first to support if-conditions, early breaks, and multiple left hand sides over structured data, as well as complex accesses such as affine indexing or scatter/gather of sparse or structured operands.

- (3) A compiler that specializes programs to data structures automatically, facilitating an expressive language that makes it easier to search the complex space of algorithms and data structures. Finch reliably utilizes four key properties of structure, such as structural zeros, repeated values, clustered non-zeros, and singletons.
- (4) 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 array indices with continuous arrays. Finch is also used as a compiler backend for the Python PyData/Sparse library [3].
- (5) We evaluate the efficiency, flexibility, and expressibility 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 (2.51×) and SpGEMM (1.25×), to more complex applications such as graph analytics (3.53×, reducing lines of code by 4× over GraphBLAS), image processing (19.5×), and the Python Array API (28×).

This advances the state-of-the-art over the Looplets work [7]. While Looplets presented a way to merge iterators over multiple single dimensional structures, we go further by combining single-dimensional looplets into multi-dimensional tensors and operating on them in a programming language with fully-featured control flow.

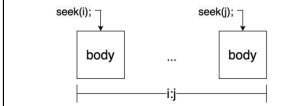
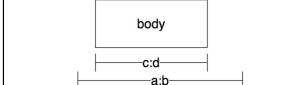
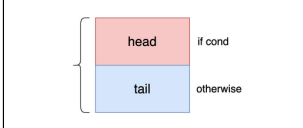
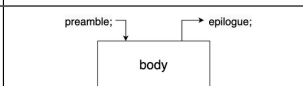
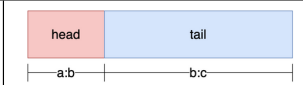
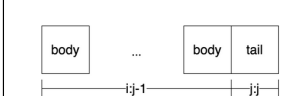
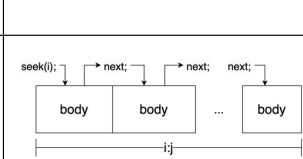
lookup (<i>seek, body</i>): The Lookup looplet represents a randomly accessible region of an iterator. The body of the lookup is understood to have one less dimension than the lookup itself, as we have already “looked up” that index in the tensor by the time we reach the body. <i>seek(i)</i> is a function that updates state to the given index.	
run (<i>body</i>): The Run looplet represents a constant region of an iterator. The body of the run is understood to have one less dimension than the lookup itself, as all of the bodies are identical.	
phase (<i>c : d, body</i>): The Phase looplet represents a restriction of the range on which a loop should execute, and allows us to succinctly express the ranges on which children of compound looplets are defined.	
switch (<i>cond, head, tail</i>): The Switch looplet allows us to specialize the body of a looplet based on a condition, evaluated in the embedding context. If the condition is true, we use ‘head’, otherwise we use ‘tail’. Switch has a high lowering priority so we can see what’s inside of it and lower that appropriately. This also lifts the condition as high as possible into the loop nest. The condition is assumed to evaluate to a Boolean.	
thunk (<i>preamble, body, epilogue</i>): The Thunk looplet allows us to cache certain computations in the state under which the body will execute. This is useful for computing and caching the results of expensive computations.	
sequence (<i>head, tail</i>): The Sequence looplet represents the concatenation of two looplets. Both arguments must be phase looplets, and are assumed to be non-overlapping, covering, and in order.	
spike (<i>body, tail</i>): The Spike looplet represents a run followed by a single value. In this paper, Spike will be considered a shorthand for sequence (phase (<i>i : j - 1, run</i> (<i>body</i>)), phase (<i>j : j, run</i> (<i>tail</i>))). In the Finch compiler, spikes are handled with special care, since they are an opportunity to align the final run to the end of the root loop extent, without using any special bounds inference.	
stepper ([<i>seek</i>], <i>next, body</i>): The stepper looplet represents a variable number of looplets, concatenated. Since our looplets may be skipped over due to conditions or various rewrites, the <i>seek</i> function allows us to fast-forward the state to the start of the root loop extent when it comes time to lower the stepper. The <i>next</i> function advances the state to the next iteration of the stepper.	

Fig. 2. The Looplet language, as understood in a correct execution of a Finch program.

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. In particular, Looplets are good at lifting code to the highest possible loop level and subdividing iteration hierarchically in coordinate space. 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.

The Looplets are described in Figure 2. We simplify the presentation to focus on the semantics, rather than precise implementation. Several looplets introduce or modify variables in the scope of the target language. It is assumed that if a looplets introduces a variable, the child looplets will not modify that variable. For more background on Looplets, we recommend the original work [7].

2.2 Fiber Trees

Fiber-tree style tensor abstractions have been the subject of extensive study [25, 26, 82]. 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. This kind of abstraction lends itself to representing sparse tensors if we vary 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, we can think of each fiber as being 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 3 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 $fiber(l, p)$. Note that the formation 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 between different fibers. Thus, the only unique data associated with $fiber(l, p)$ is the position p .

3 BRIDGING LOOPLETS AND FINCH: THE TENSOR INTERFACE

Arrays use multiple dimensions to organize data with respect to orthogonal concepts. Thus, the Finch language supports multi-dimensional arrays. Unfortunately, the Looplets 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 looplets nests for each iterator. Next, since Finch programs go beyond just single Einsums, they may read and write to the same data at

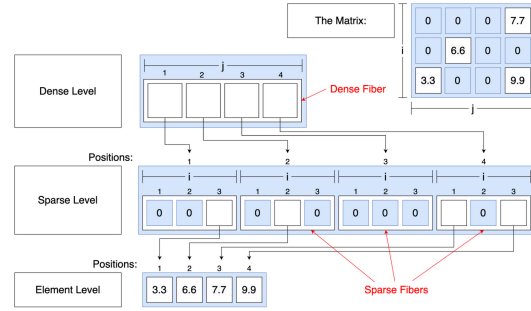


Fig. 3. Levels in the fiber tree representation of a sparse matrix in CSC format, with a dense outer level and a sparse inner level. The element level holds the leaves of the tree.

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 while we iterate 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 we are reading from it. 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.

The *unfurl* function is used to manage iteration over a subfiber. At the point when it comes time to iterate over a tensor, be in on the left or right hand side of an assignment, we call *unfurl* to precompute whatever state and datastructures are necessary to return a looplet nest that would iterate over that level of a tensor. We call *unfurl* directly before iterating over the corresponding loop, so it has access to any state variables introduced by freezing or thawing the tensor.

<i>declare</i> (<i>lvl</i> , <i>init</i> , <i>dims</i> ...): Declares the level to hold subtensors of size <i>dims</i> and an initial value of <i>init</i> . Requires the level to be in read-only mode.
<i>freeze</i> (<i>lvl</i>): Finalizes the updates in the level, and readies the level for reading. Requires the level to be in update-only mode.
<i>thaw</i> (<i>lvl</i>): Prepares the level to accept updates. Requires the level to be in read-only mode.
<i>unfurl</i> (<i>fiber</i> (<i>lvl</i> , <i>pos</i>), [<i>ext</i>], <i>mode</i> , [<i>op</i>], [<i>rhs</i>]): When <i>ext</i> is given, unfurls the fiber at position <i>pos</i> in the level <i>lvl</i> over the extent <i>ext</i> . When <i>mode</i> = read , returns a looplet nest over the values in the read-only fiber. When <i>mode</i> = update , returns a looplet nest over mutable subfibers in the update-only fiber. When applied to a scalar or leaf node, <i>ext</i> is omitted and we may also use <i>op</i> and <i>rhs</i> to update the scalar value. Often, skipping over mutable locations allows the level to know which locations must be stored. Often, a dirty bit may be used in <i>tns</i> to communicate whether the mutable subfiber has been written to, which allows the parent fiber to know whether the subfiber must be stored explicitly.
<i>assemble</i> (<i>lvl</i> , <i>pos_start</i> , <i>pos_stop</i>): Allocates subfibers in the level from positions <i>pos_start</i> to <i>pos_stop</i> . Usually, this function is only ever called on unassembled positions, but some levels (such as dense levels or bytemaps) may support reassembly.

Fig. 4. The five functions that define a level.

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. Note: it was an intentional choice to hold the parent level responsible for managing the data of the sublevels, which positions they allocate, etc. This allows the parent level to reuse allocation logic from internal index datastructures. For example, a sparse level might use a list of indices to store which nonzeros are present, and when it comes time to resize that list, it could also call $assemble$ to resize the sublevel, reducing the number of branches in the code. 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. This benefit is made clear in our case studies, where prior systems like TACO do not support all possible loop orderings and format combinations for sparse matrix multiply because they do not have a flexible enough allocation strategy, instead using a two-phase approach which requires computing a complicated closed-form kernel to iterate over the data twice to determine the number of required output nonzeros.

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 provides us with 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, 50, 89].
- **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` $\rightarrow$ `s += n * x` or code motion to lift operations out of loops [7, 31].
- **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, 38].

Sparse	Blocked	Runs	Singular	Corresponding Format
				Dense
			✓	n/a
		✓		DenseRLE
		✓	✓	n/a
✓	✓			n/a
	✓		✓	n/a
	✓	✓		n/a
	✓	✓	✓	n/a
✓				Sparse
✓			✓	SparsePinpoint
✓		✓		SparseRLE
✓		✓	✓	SparseInterval
✓	✓			SparseVBL
✓	✓		✓	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. Note that blocks and runs need not be considered together because we must store a run length for each run, so there isn't a significant storage benefit to combining them. 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 3 and Table 5.

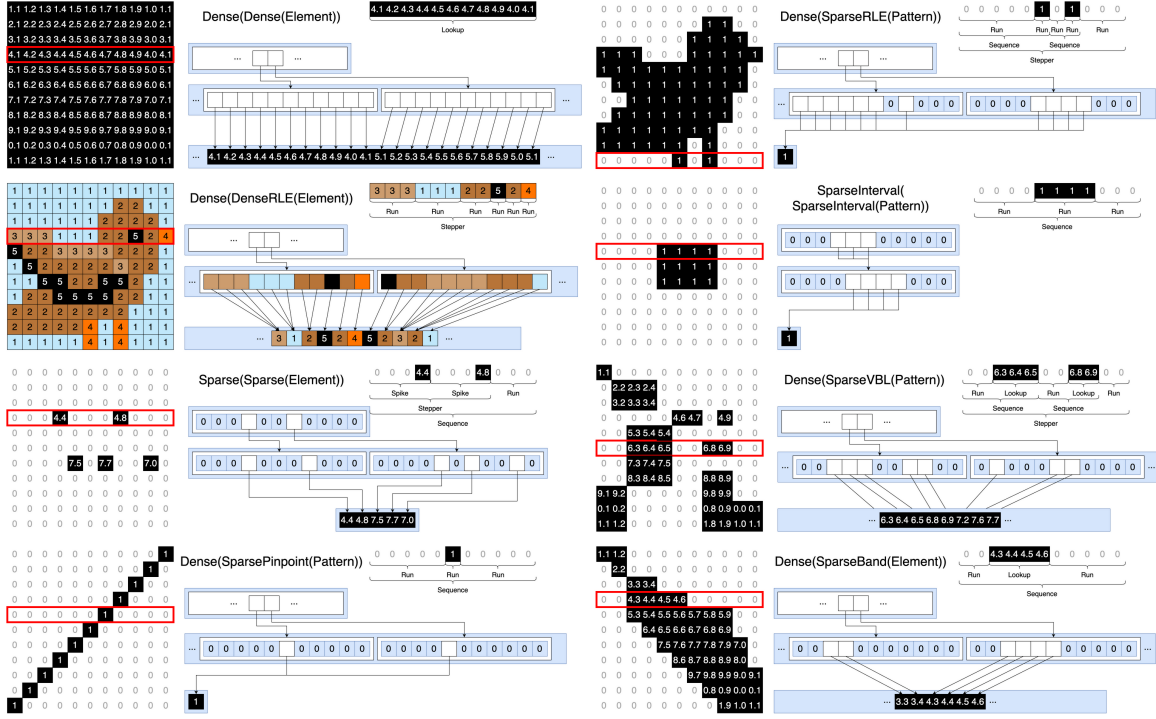


Fig. 6. Several examples of matrix structures represented using the level structures identified in Table 5. Comparing this figure to [7, Figure 3], we see that a level-by-level structural decomposition is diagrammed together with the Looplets.

3.3 Implementations of Structures

3.3.1 Sequentially Constructed Levels. We consider all combinations of these structural properties in Table 5, resulting in 8 key level formats that correspond to the 8 resulting situations. While it is impossible to write code which precisely addresses every possible structure, our level formats can be combined to express a wide variety of hierarchical structures to a sufficient granularity that we can generate code which utilizes the four properties. For example, though banded arrays are a superset of ragged arrays, representing a ragged array using a banded format does not add much overhead. The structures we consider are exhaustive in the sense that they address all combinations of sparsity, blocks, runs, and singletons in each level. We can represent a wide variety of hierarchical tensor structures by combining these level structures in a tree, as shown in Figure 6.

3.3.2 Non-sequentially Constructed Levels. To reduce the implementation burden and improve efficiency in the common case, the levels we described in the previous section only support bulk, sequential construction of formats. However, when users want to be able to write out of order (which is a common requirement arising from loop order or from the problem itself, it occurs in our SpGEMM algorithms and our histogram example in the evaluation section), we must use more complicated datastructures like hash tables and trees to support the random writes so that in-order levels can be constructed later. Because these datastructures are more complex and have a higher implementation burden and performance overhead, we only support random access construction of sparse or dense structures. We can use these two more general structures 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. Scalars don't have as much structure as clearly only two structures apply. Through these structures, scalars can also affect

Sequentially Constructed Levels	
Dense:	The dense format is the simplest format, mapping $fiber(l, p)[i] \rightarrow fiber(l.lvl, p \times l.shape + i)$. 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.
DenseRLE:	Used to represent runs of repeated values, storing two vectors, <i>right</i> and <i>ptr</i> , with q^{th} run in the p^{th} subfiber starting and ending at $right[ptr[p] + q]$ and $right[ptr[p] + q + 1] - 1$, respectively. A challenge arises for this level: it is difficult to merge duplicate runs. Such a scenario might arise when 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 we cannot read them while the sublevel is in update-only mode. Instead, we merge the duplicates during the freeze phase. We <i>freeze</i> the sublevel, <i>declare</i> a separate sublevel <i>buf</i> as a buffer to store the deduplicated subfibers, and then compare each of the subfibers in the main level, copying the deduplicated subfibers into the buffer.
SparseList:	The simplest sparse format, used to construct popular formats like CSR, CSC, DCSR, DCSC, and CSF. It stores two vectors, <i>idx</i> and <i>ptr</i> , such that $idx[ptr[p] + q]$ 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 <i>ptr</i> field. It stores a vector <i>idx</i> , such that $idx[p]$ is the nonzero index in the subfiber at position p .
SparseRLE:	Similar to DenseRLE level, but because runs are sparse, we must also store the start of each run. It stores three vectors <i>left</i> , <i>right</i> , and <i>ptr</i> , such that the q^{th} run in the p^{th} subfiber begins and ends at $left[ptr[p] + q]$ and $right[ptr[p] + q]$, respectively. Like DenseRLE, it also stores a duplicate sublevel, <i>buf</i> , for deduplication.
SparseInterval:	Similar to SparseRLE, but only stores one run per subfiber, eliminating the need for the <i>ptr</i> 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 $left[p]$ and $right[p]$ respectively.
SparseVBL:	Used to represent blocked data. It stores three vectors, <i>idx</i> , <i>ptr</i> , and <i>ofs</i> , such that $ofs[ptr[p] + q] : ofs[ptr[p] + q + 1] - 1$ are the subpositions of block q ending at index $idx[ptr[p] + q]$ in the subfiber at position p .
SparseBand:	Similar to SparseVBL, but stores only one block per subfiber, eliminating the need for the <i>ptr</i> field. It stores two vectors <i>idx</i> and <i>ofs</i> , such that $ofs[p] : ofs[p + 1] - 1$ are the subpositions of the block ending at $idx[p]$ in subfiber p .
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, <i>idx</i> and <i>ptr</i> , such that $idx[ptr[p] + q]$ is the index of the q^{th} nonzero in the subfiber at position p . Also stores a hash table <i>tbl</i> 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, <i>idx</i> and <i>ptr</i> , such that $idx[ptr[p] + q]$ 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 <i>tbl</i> , a dense array of booleans such that $tbl[shape * p + i]$ 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 <i>val</i> 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 <i>false</i> and whose stored values are all <i>true</i> .
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.
EarlyBreakScalar:	A scalar which triggers early breaks in reductions whenever an annihilator is encountered.

Table 3. 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 its default fill value and thus, whether it needs to be stored.

other tensors in crucial ways. Constant propagation through arrays is known to be a complex compiler pass [65]. Instead, we provide sparse scalars, which allow reductions but also specialize read accesses for the possible zero value. We also provide early break scalars, which modify the stepper looplets to re-specialize the loop whenever a reduction into an early break scalar hits an annihilator. We don't need to re-specialize other looplets because the stepper is the only one which repeats a non-constant number of times. We represent early break as a structural property rather than a program node because it allows us to represent the tail of a loop where one scalar has hit an

annihilator but another scalar hasn't. To our knowledge, sparse and early break scalars 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

4.1 Syntax

The syntax of Finch is displayed in Figure 7. The Finch syntax mirrors most imperative languages with for-loops and control flow. Notable statements that have been added to the language include **for**, **let**, blocks of code with **if**, and the lifecycle functions that let us declare, freeze, and thaw tensors. At the level of expressions, we support a wide array of scalar operations through literals and calls to functions whose properties and definitions are defined externally. The expressions can also interact with indices and extents. Finally, 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 four will be introduced through evaluation of **for**-loops loops and accesses, in the next section.

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 most prominently in our implementation of Gustavson's algorithm for sparse-sparse matrix multiply, which simply 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. The only restriction we impose on our syntax is that it must respect tensor life cycles. In the semantics section, we detail the specifics of how we compile our syntax to efficient code over structured data.

4.2 Semantics

We present a sketch of a small-step operational semantics, showing how to execute a Finch program in a host language. We distinguish evaluation of the language by evolving compiler state with $\langle \rangle$ and evaluation in the host language via $\langle\langle \rangle\rangle$. For example, in Figure 8, we declare the core semantics of the language and the *Define* rule (**let**) adds a definition to the compiler state whereas the *Call* rule passes evaluation off to the host. In Figure 9, we detail the Looplet evaluation semantics, which

```

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

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
ASSIGN := ACCESS <<EXPR>>= EXPR
LOOP := for INDEX = EXPR
      STMT
      end
DEFINE := let VARIABLE = EXPR
      STMT
      end
SIEVE := if EXPR
      STMT
      end
BLOCK := begin
      STMT...
      end

LITERAL := V
VALUE := S::T
TENSOR := S
INDEX := S
VARIABLE := S
EXTENT := EXPR : EXPR
CALL := EXPR(EXPR...)
ACCESS := TENSOR[EXPR...]
MODE := @mode(TENSOR)

```

Fig. 7. The syntax of the finch language. Compare this grammar to the Concrete Index Notation of TACO [53, Figure 3], noting the addition of multiple left hand sides through code blocks, access with arbitrary expressions, and explicit declaration, as well as freeze and thaw.

details how accesses within a loop can be replaced with Looplets, which are progressively rewritten to host code. Due to the use of rewrite rules, our semantics here make use of an evaluation context.

We include the sketch of our semantics to highlight several key design choices. First, we note that the life cycle rules (in combination with the definition rules) prevent programs that access tensors on both the left and the right hand side of an incrementing assignment within a loop. Tensors may only change their mode in the scope in which they were declared. The **loop**, **declare**, and **sieve** nodes introduce new scopes, and tensors without declarations are assumed to have global scope. Beyond the simple management of iterator state, these rules allow us to effectively analyze the propagation of constants, zeros, and repetitive values within a loop, eliminating problems that in a less restrictive language would require alias analysis. Second, loops enter into a looplet system via the *Unfurl* rule. In this system, repeated structures and constants are slowly uncovered as accesses are lowered in various points in the program (e.g. *Run* and *Switch*, respectively). In this process, we are able to 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).

The next section will detail how the compiler adds additional features on top of this core language. We rely on the assurance that accesses to tensor values will first unravel to Looplets and then simplify in the context of the tensor access. We will handle complex index access expressions involving indirection and arithmetic by transforming complex accesses to additional loop variables with additional tensor accesses to insert Looplets to provide the structure dictated by the complex control flow. We will also show how we can similarly exploit the structure of complex conditional expressions when the structure of the expressions is placed inside a tensor access.

5 THE FINCH COMPILER

5.1 Finch Normal Form

Our semantics in Figure 8 and Figure 9 is only well-defined on some programs. We define a particular class of programs on which our semantics are well-defined, and refer to it as **Finch Normal Form**. The properties of such a program are as follows:

- **Access with Indices:** Though Finch allows general expressions 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 normal form to programs whose loop dimensions and declaration dimensions are extents with limits defined in the scope of the corresponding loop or declaration statement.
- **Concordant:** Finch is column-major by default, and the normal form requires the order of indices in an access to 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. To make it easier to statically analyze lifecycle constraints, we restrict tensors to only change modes in the same scope in which they were defined.

The subsequent sections will 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 non-concordant programs like `for i = _; for j = _; s[] += A[i, j] end end` by adding a loop to randomly access A or adjusting the storage order of A by adding a lazy transposition wrapper.

5.2 Wrapperization

Many fancy operations on indices can be resolved by introducing equivalent **wrapper arrays** which modify the behavior of the tensors they wrap, or by introducing **mask arrays** which replace index expressions like $i \leq j$ with their equivalent masks (in this case, a triangular mask tensor). Wrappers and masks are summarized in 4.

All wrapper arrays are eventually unwrapped by the compiler as we lower them, some earlier than others. For example, the *swizzle* array wraps a tensor and permutes the indices of an access when it is unwrapped during the wrapperization pass. On the other hand, the *offset* array wraps a tensor and shifts all of the ranges declared in Figure 9 by one. The implementation burden for a wrapper is to implement a suitable program rewrite during the wrapperization procedure to unwrap the wrapper, or to implement the looplet functions of 9 with some minor modifications to shift dimensions, for example.

Mask arrays have a more straightforward implementation 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.

```

for i=_, j=_
  if i <= j
    s[] += A[i, j]
  end
end
↓
for i=_, j=_
  if UpTriMask()[i, j]
    s[] += A[i, j]
  end
end
↓
for i = 1:n
  for j = 1:i
    s[] += A[i, j]
  end
end

```

Fig. 10. Wrapperization

$$\begin{array}{c}
\frac{\langle val, (e, t, d) \rangle \rightarrow val' \quad var \notin d}{\langle \mathbf{define}(var, val, body), (e, t, d) \rangle \rightarrow \langle body, (e[var \mapsto val'], t, \{\}) \rangle} \text{Define} \qquad \frac{\langle args_i, (e, t) \rangle \Rightarrow vals_i \quad \langle f, (e, t) \rangle \Rightarrow g}{\langle \mathbf{call}(f, args \dots), (e, t) \rangle \rightarrow \langle\langle g(vals \dots), t \rangle\rangle} \text{Call} \\
\\
\frac{}{\langle \mathbf{literal}(val), (e, t, d) \rangle \rightarrow val} \text{Literal} \qquad \frac{}{\langle \mathbf{variable}(name), (e, t, d) \rangle \rightarrow e(\mathbf{variable}(name))} \text{Variable} \qquad \frac{}{\langle \mathbf{index}(name), (e, t, d) \rangle \rightarrow e(\mathbf{index}(name))} \text{Index} \\
\\
\frac{\langle body, s \rangle \rightarrow s'}{\langle \mathbf{block}(body, tail \dots), s \rangle \rightarrow \langle \mathbf{block}(tail \dots), s' \rangle} \text{Block} \qquad \frac{}{\langle \mathbf{value}(ex, type), (e, t, d) \rangle \Rightarrow \langle\langle ex, t \rangle\rangle} \text{Value} \\
\\
\frac{\langle cond, (e, t, d) \rangle \Rightarrow true \quad \langle body, (e, t, \{\}) \rangle \rightarrow (e', t', d')}{\langle \mathbf{sieve}(cond, body), (e, t, d) \rangle \rightarrow (e', t', d)} \text{SieveTrue} \qquad \frac{\langle cond, s \rangle \Rightarrow false}{\langle \mathbf{sieve}(cond, body), s \rangle \rightarrow s} \text{SieveFalse} \\
\\
\frac{s = (e, t, d) \quad tns \notin d \quad e(tns) = tns' \quad \langle init, s \rangle \Rightarrow init' \quad \forall i \langle init, dims_i \rangle \Rightarrow dims'_i}{\langle \mathbf{declare}(tns, init, dims), s \rangle \rightarrow (e[\mathbf{mode}(tns) \mapsto \mathbf{update}], \langle\langle \mathbf{declare}(tns', init', dims' \dots), t \rangle\rangle, d \cup \{tns\})} \text{Declare} \\
\\
\frac{s = (e, t, d) \quad e(\mathbf{mode}(tns)) = \mathbf{update} \quad tns \in d \quad e(tns) = tns'}{\langle \mathbf{freeze}(tns), s \rangle \rightarrow (e[\mathbf{mode}(tns) \mapsto \mathbf{read}], \langle\langle \mathbf{freeze}(tns'), t \rangle\rangle, d)} \text{Freeze} \\
\\
\frac{s = (e, t, d) \quad e(\mathbf{mode}(tns)) = \mathbf{read} \quad tns \in d \quad e(tns) = tns'}{\langle \mathbf{thaw}(tns), s \rangle \rightarrow (e[\mathbf{mode}(tns) \mapsto \mathbf{update}], \langle\langle \mathbf{thaw}(tns'), t \rangle\rangle, d)} \text{Thaw} \\
\\
\frac{e(tns) = tns' \quad \langle op, s \rangle \rightarrow op' \quad \langle rhs, s \rangle \rightarrow rhs' \quad e(\mathbf{mode}(tns)) = \mathbf{update} \quad \langle\langle unfurl(tns', \mathbf{update}, op', rhs'), t \rangle\rangle \rightarrow t'}{\langle E[\mathbf{assign}(\mathbf{access}(tns), op, rhs)], (e, t, d) \rangle \rightarrow (e, t', d)} \text{Assign}
\end{array}$$

Fig. 8. Basic evaluation semantics, roughly defining most of these language constructs to function similarly to their classical definitions. The state s of the finch compiler 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 . This means that rules which modify t are running in the host language. Several looplets introduce variables into the embedding language, which may be read when evaluating the **value** node. All of the lifecycle functions are designed to be implemented and executed in the host language, but these semantics enforce that each of these functions may update state in the host language and flip the mode of the tensor between **read** and **update**.

$$\begin{array}{c}
E := [\cdot] | \mathbf{loop}(idx, ext, E|body) | \mathbf{block}(E|head, E|tail) | \mathbf{sieve}(E|cond, E|body) | \mathbf{assign}(E|lhs, op, E|rhs) \\
\mathbf{declare}(var, E|rhs, E|body) | \mathbf{call}(E|f, E|args...) | \mathbf{access}(tns, E|idxs...) \\
\frac{e(tns) \mapsto tns' \quad e(\mathbf{mode}(tns)) \mapsto m \quad \langle\langle unfurl(tns', ext, m), t \rangle\rangle \Rightarrow tns''}{\langle\mathbf{loop}(i, ext, E[\mathbf{access}(tns, j..., i)]), s\rangle \rightarrow \langle\mathbf{loop}(i, ext, E[\mathbf{access}(tns'', j..., i)]), s\rangle} \text{Unfurl} \\
\frac{\langle\mathbf{loop}(i, ext, E[\mathbf{access}(\mathbf{run}(body), j..., i)]), s\rangle \rightarrow \langle\mathbf{loop}(i, ext, E[\mathbf{access}(body, j...)]), s\rangle}{\langle\mathbf{loop}(i, ext, E[\mathbf{access}(\mathbf{run}(body), j..., i)]), s\rangle \rightarrow \langle\mathbf{loop}(i, ext, E[\mathbf{access}(body, j...)]), s\rangle} \text{Run} \quad \frac{e(i) = i' \quad \langle\langle seek(i'), t \rangle\rangle \rightarrow t'}{\langle E[\mathbf{access}(\mathbf{lookup}(seek, body), j..., i)], (e, t, d) \rangle \rightarrow \langle E[\mathbf{access}(body, j...)], (e, t', d) \rangle} \text{Lookup} \\
\frac{\langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{assign}(\mathbf{access}(\mathbf{run}(body), j..., i), op, rhs)]), s\rangle \rightarrow \langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{sieve}(i == a, \mathbf{assign}(\mathbf{access}(body, j...), op, rhs)]), s\rangle}{\langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{assign}(\mathbf{access}(\mathbf{run}(body), j..., i), op, rhs)]), s\rangle \rightarrow \langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{sieve}(i == a, \mathbf{assign}(\mathbf{access}(body, j...), op, rhs)]), s\rangle} \text{AcceptRun} \\
\frac{\langle\langle cond, t \rangle\rangle \Rightarrow true}{\langle E[\mathbf{access}(\mathbf{switch}(cond, head, tail), i...)], s \rangle \rightarrow \langle E[\mathbf{access}(head, i...)], s \rangle} \quad \frac{\langle\langle cond, t \rangle\rangle \Rightarrow false}{\langle E[\mathbf{access}(\mathbf{switch}(cond, head, tail), i...)], s \rangle \rightarrow \langle E[\mathbf{access}(tail, i...)], s \rangle} \text{Switch} \\
\frac{\langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{access}(\mathbf{phase}(\mathbf{extent}(c, d), body), j..., i)]), s \rangle \rightarrow \langle\mathbf{loop}(i, \mathbf{extent}(\max(a, c), \min(b, d)), E[\mathbf{access}(body, j..., i)]), s \rangle}{\langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{access}(\mathbf{phase}(\mathbf{extent}(c, d), body), j..., i)]), s \rangle \rightarrow \langle\mathbf{loop}(i, \mathbf{extent}(\max(a, c), \min(b, d)), E[\mathbf{access}(body, j..., i)]), s \rangle} \text{Phase} \\
\frac{\langle\langle preamble, t \rangle\rangle \rightarrow t' \quad \langle E[body], (e, t', d) \rangle \rightarrow (e', t'', d) \quad \langle\langle epilogue, t'' \rangle\rangle \rightarrow t''}{\langle E[\mathbf{thunk}(preamble, body, epilogue)], (e, t', d) \rangle \rightarrow (e', t'', d)} \text{Thunk} \\
\frac{\langle\mathbf{loop}(i, ext, E[\mathbf{access}(head, j..., i)]), s \rangle \rightarrow s'}{\langle\mathbf{loop}(i, ext, E[\mathbf{access}(\mathbf{sequence}(head, tail), j..., i)]), s \rangle \rightarrow \langle\mathbf{loop}(i, ext, E[\mathbf{access}(tail, j..., i)]), s' \rangle} \text{Sequence} \quad \frac{\langle node, algebra \rangle \rightarrow node'}{\langle E[node], s \rangle \rightarrow \langle E[node'], s \rangle} \text{Simplify} \\
\frac{\langle\langle seek(a), t \rangle\rangle \rightarrow t'}{\langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{access}(\mathbf{stepper}(seek, body, next), j..., i)]), (e, t, d) \rangle \rightarrow \langle\mathbf{loop}(i, \mathbf{extent}(a, b), E[\mathbf{access}(\mathbf{stepper}(body, next), j..., i)]), (e, t', d) \rangle} \text{StepperSeek} \\
\frac{\langle\mathbf{loop}(i, ext, E[\mathbf{access}(body, j..., i)]), (e, t, d) \rangle \rightarrow (e', t', d) \quad \langle\langle next, t' \rangle\rangle \rightarrow t''}{\langle\mathbf{loop}(i, ext, E[\mathbf{access}(\mathbf{stepper}(body, next), j..., i)]), s \rangle \rightarrow \langle\mathbf{loop}(i, ext, E[\mathbf{access}(\mathbf{stepper}(body, next), j..., i)]), (e', t'', d) \rangle} \text{StepperNext} \\
\frac{\langle\mathbf{loop}(i, \mathbf{extent}(a, b), body), s \rangle \rightarrow \langle\mathbf{block}(\mathbf{define}(i, a, body), \mathbf{sieve}(a < b, \mathbf{loop}(i, \mathbf{extent}(a + 1, b), body))), s \rangle}{\langle\mathbf{loop}(i, \mathbf{extent}(a, b), body), s \rangle \rightarrow \langle\mathbf{block}(\mathbf{define}(i, a, body), \mathbf{sieve}(a < b, \mathbf{loop}(i, \mathbf{extent}(a + 1, b), body))), s \rangle} \text{Loop} \\
\frac{e(tns) \mapsto tns' \quad e(\mathbf{mode}(tns)) \mapsto \mathbf{read} \quad \langle\langle unfurl(tns', \mathbf{read}), t \rangle\rangle \rightarrow tns''}{\langle E[\mathbf{access}(tns)], s \rangle \rightarrow \langle E[tns''], s \rangle} \text{Access}
\end{array}$$

Fig. 9. Looplet evaluation semantics. Basic evaluation semantics, roughly defining most of these language constructs to function similarly to their classical definitions. The state of the compiler is described in the previous figure. 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 variables introduced by a looplet may be modified by the looplet itself (steppers often increment some state variables), we forbid child looplets from modifying any state variables that they didn't introduce. This allows us to treat the **value** node as a constant. Note: the *Simplify* rule references *algebra*, which is our variable defining a set of straightforward simplification rules. These rules include simple properties like $x * 0 \rightarrow 0$ to more complicated ones such as constant propagation. We omit the full set of rules for brevity, but point the curious reader to [7, Figure 5] for some examples.

Transformation Example	Description
$A[i + a] \rightarrow \text{offset}(A, 1)[i]$	Creates an OffsetArray such that $\text{offset}(\text{tns}, \text{delta}...)[i...] == \text{tns}[i... + \text{delta}...]$.
$A[i + x] \rightarrow \text{toeplitz}(A, 1)[i, x]$	Creates a ToeplitzArray, adding 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.
$A[a:b)(i) \rightarrow \text{window}(A, a:b)[i]$	Creates a WindowedArray, representing a view into another tensor. This wrapper returns a different size of tensor.
$A[i] \rightarrow \text{permissive}(A)[i]$	Creates a PermissiveArray, allowing for out-of-bounds access or padding. This tensor returns no dimensions as its size.
$A[p(i)] \rightarrow \text{swizzle}(A, \text{perm})[i]$	A lazily transposed array, $\text{swizzle}(A, \text{perm})[\text{idx}...] is transformed to A[\text{idx}[\text{perm}]...] during wrapperization.$
$i < j \rightarrow \text{UpTriMask}()[i, j - 1]$	Upper triangular mask, true if $i < j$.
$i \geq j \rightarrow \text{LoTriMask}()[i, j]$	Lower triangular mask, true if $i \geq j$.
$l \leq i < j \rightarrow \text{Bandmask}()[i, l, h - 1]$	Banded mask, true for elements within a specified band.
$i == j \rightarrow \text{DiagMask}()[i, j]$	Diagonal mask, true if $i == j$.
$i \neq j \rightarrow \text{!(DiagMask}()[i, j])$	Inverse diagonal mask, true if $i \neq j$.
$\text{chunkmask}(b)$	Chunk mask, for chunked tensor access. True if $b \times (j - 1) < i \leq b \times j$.

Table 4. Wrapper arrays, masks, and some example indexing sugar they enable.

5.3 Dimensionalization

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 [54]. 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 [41, 70]. We refer to the process of discovering suitable dimensions as **dimensionalization**.

In Finch, we use a straightforward dimensionalization algorithm on 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. Our algorithm uses the following principles, assuming dimension types form a lattice:

- (1) We assign dimensions to indices.
- (2) Using an index in an access “hints” that the index should have the corresponding dimension.
- (3) Loop dimensions are equal to the “meet” of all hints in the loop body and any existing dimensions in the loop bounds. The meet usually asserts that dimensions match, but may also e.g. propagate info about parallelization
- (4) The $_$ symbol represents a dimensionless quantity at the bottom of the dimension lattice.
- (5) We assign dimensions to declarations.
- (6) Left hand side (updating) accesses “hint” the size of their tensor
- (7) The dimensions of a declaration are the “meet” of all hints from the declaration to the first read.
- (8) The new dimensions of the declared tensor are used when the tensor is on the right hand side (reading) access.

```

#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]
    end
  end
end
↓
C := 0
for i = 1:3
  for j = 1:5
    for k = 1:4
      C[i, j] += A[i, k] * B[k, j]
    end
  end
end

```

Fig. 11. Dimensionalization.

For example, in Figure 11, the second dimension of A must match the first dimension of B . Also, the first dimension of A must match the i loop dimension, 1:3. Finch will also resize declared tensors to match indices used in writes, so C is resized to (1:3, 1:5). If no dimensions are specified elsewhere, then Finch will use the dimension of the declared tensor.

Dimensionalization occurs after wrapper arrays are de-sugared. You can therefore exempt a mode in an access from dimensionalization by wrapping the corresponding index in `~` to produce a "PermissiveArray" (e.g. `A[~i]`).

5.4 Concordization

After wrapperization, all arrays are normalized to column-major ordering and the unrecognized expressions are left in the access expressions. At that point, we run a pass over the code to make the program concordant. We make expressions concordant by inserting loops with one iteration. Examples are given in Figure 12.

5.5 Life Cycle Automation

Finally, we introduce a pass to avoid needed to manually insert the `@freeze` or `@thaw` macros. The pass will insert these statements at the appropriate place in the program if they have not been inserted already, easing the burden on the programmer and bridging between structured and dense languages.

6 CASE STUDIES

We evaluate Finch on a broad set of applications to showcase its efficiency, flexibility, and expressibility. All of our implementations highlight the benefits of data structure and algorithm co-design. Our implementation of sparse-sparse-matrix multiply (SpGEMM) translates classical lessons from sparse performance engineering into the language of Finch, using temporaries and randomly-accessible workspace formats to efficiently implement the three main approaches. Our study of sparse-matrix-dense-vector multiply (SpMV) highlights the benefits of precise structural specialization. Our studies of image morphology and graph applications show how Finch's programming model can express more complex real-world kernels. Finally, we explain how flexible operators, formats, and indexing expressions in Finch have supported a flexible implementation of the Python Array API, supporting fused execution.

All experiments were run on a single core of a 12-core 2-socket Intel Xeon E5-2695 v2 running at 2.40GHz 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. The Finch compiler is publicly available online at <https://github.com/willow-ahrens/Finch.jl>.

6.1 Sparse Matrix-Vector Multiply (SpMV)

Sparse matrix-vector multiply (SpMV) has a wide range of applications and has been thoroughly studied [61, 99]. Because SpMV is bandwidth bound, many formats have been proposed to reduce the footprint [58]. The wide range of applications unsurprisingly results in a wide range of array structures, making it an effective kernel to demonstrate the utility of our programming model. We varied both the data formats and the SpMV algorithm. Our formats are shown in Table 15, and our programs are listed in Figure 14. In varying the program, we consider both row and column-major SpMV

```

for i = _
  for j = _
    s[] += A[i, j]
  end
end
↓
for i = _
  for j = _
    for k = _
      if i == k
        s[] += A[k, j]
      end
    end
  end
end
end

for i = _
  A[I[i]] += 1
end
↓
for i = _
  for j = _
    if j == I[i]
      A[j] += 1
    end
  end
end
end

```

Fig. 12. Examples of concordization, transforming accesses to normal column major.

```

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

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

```

Fig. 13. Life cycle automation.

```

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

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]
  end
end
#D is the diagonal
y[j] += y_j[] + D[j] * x_j
end

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]
  end
end
#D is the diagonal
y[j] += y_j[] + D[j] * x_j
end

```

Fig. 14. Finch row-major, column-major and symmetric SpMV Programs

programs, as well as a symmetric SpMV. Finch enables us to exploit symmetry effectively. Our program restricts our attention to the canonical triangle (using masks), reuses the reads to the canonical triangle of the symmetric matrix (using a **define** statement), and writes the results to both relevant locations (using multiple outputs). All of our programs apply to all of our level formats, enabling exploitation of multiple structural patterns concurrently (e.g. sparsity *and* symmetry).

Figure 16 displays speedup relative to TACO, SuiteSparseGraphBLAS, and Julia’s standard library. We test using sparse matrices from a large selection of datasets spanning several previous papers: the datasets used by Vuduc et al. to test the OSKI interface [90], Ahrens et al. to test a variable block row format partitioning strategy [6], and Kjolstad et al. to test the TACO library [54]. Additionally, we included the SNAP graph collection to test with Boolean matrices. We also created some synthetic matrices containing bands of varying sizes. We tested using the row-major and column-major Finch programs in Figure 14 as well as the symmetric program where applicable; the performance displayed for Finch on each dataset is the fastest among the formats and programs we tested. For TACO, Column-major SpMV consistently performs better than row-major SpMV (an average of 1.36x better) so we use column-major SpMV in TACO as our baseline.

Format	Style of Matrix
Dense(SparseList(Element(0.0)))	real, sparse
Dense(SparseList(Pattern()))	Boolean, sparse
Dense(SparseVBL(Element(0.0)))	real, blocked
Dense(SparseBand(Element(0.0)))	real, banded

Fig. 15. SpMV Tensor Formats

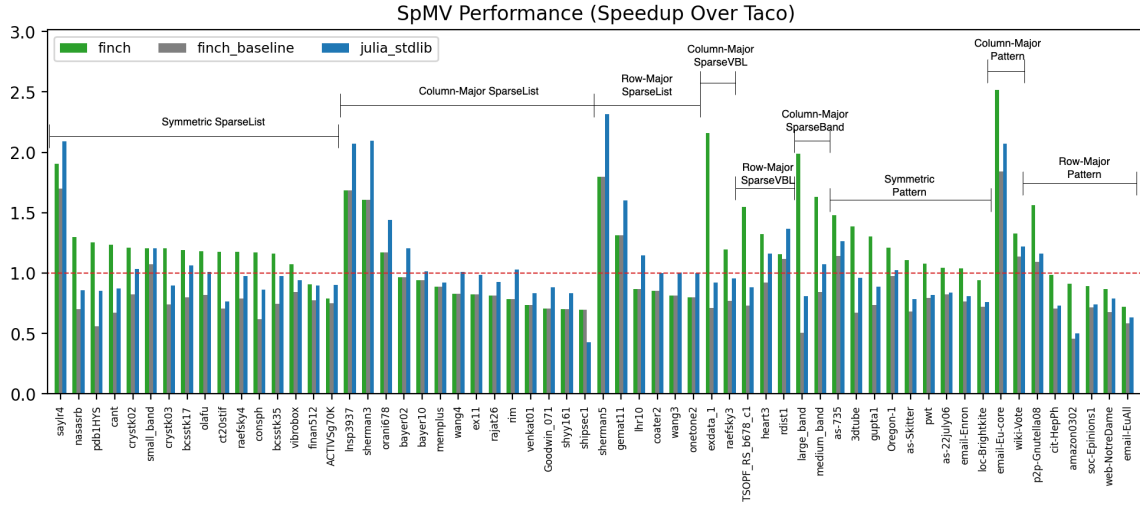


Fig. 16. Performance of SpMV by Finch format. The performance displayed for Finch on each dataset is the fastest among the formats we tested. Programs are from Figure 14 and formats are from Figure 15. “finch_baseline” is the faster of row major or column major “SparseList”.

We found that the SpMV performance was superior for the level format that best paralleled the structure of the tensor. Matrices with a clear blocked structure like `exdata_1`, `TSOPF_RS_b678_c1`, and `heart3` performed notably well with the `SparseVBL` format with speedups of 2.16, 1.55, and 1.30 relative to TACO, while the baseline format had minor slowdowns relative to TACO. Furthermore, the synthetic banded matrices we constructed performed the best with the `SparseBand` matrix, in particular with the `large_band` and the `medium_band` matrices having a speedup of 1.98 and 1.64 relative to TACO, while the baseline format had minor slowdowns relative to TACO. The `Pattern` format performed better than the `Element` format for representing the leaf values in the matrices when these values were Boolean, such as matrices in the SNAP collection which represent graph datasets are Boolean. For example, the `SparseList-Pattern` for `email-Eu-core` resulted in a speedup

of 2.51, while the SparseList-Element format resulted in a slowdown of 1.84 over TACO. Every symmetric matrix in the SparseList and SparseList-Pattern formats has better performance when we use a Finch SpMV program that takes advantage of this symmetry. Symmetric SpMV with the SparseList level format in Finch results in an average of 1.27x speedup over TACO and symmetric SpMV with the SparseList-Pattern format in Finch results in an average speedup of 1.21x over TACO. Notably, there is a 1.91x speedup for the HB/saylr4 matrix over TACO.

6.2 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 [97, 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 $O(MN)$ entries in the output [8]. We will show that our ability to implement these latter methods exceeds that of TACO, translating to asymptotic benefits.

Figure 17 implements all three approaches in Finch, and Figure 18 compares the performance of Finch to TACO. Note that these algorithms mainly differ in their loop order, but that different data structures can be used to support the various access patterns induced. In our Finch implementation of outer products, we use a sparse hash table, as it is fully-sparse and randomly accessible. Since, TACO

```

@finch begin
  C .= 0
  for j=1:n
    for i=1:m
      for k=1:k
        C[i, j] += AT[k, i] * B[k, j]
      end
    end
  end
  return C
end

w = Tensor(SparseByteMap(Element{0}))
@finch begin
  C .= 0
  for j=1:n
    w .= 0
    for k=1:k
      for i=1:m
        w[i] += A[i, k] * B[k, j]
      end
    end
    for i=1:m
      C[i, j] = w[i]
    end
  end
end

w = Tensor(SparseHash(SparseHash(Element{0})))
@finch begin
  w .= 0
  for k=1:k
    for j=1:n
      for i=1:m
        w[i, j] += A[i, k] * BT[j, k]
      end
    end
  end
  C .= 0
  for j=1:n, i=1:m
    C[i, j] = w[i, j]
  end
end

```

Fig. 17. Inner Products, Gustavson’s, and Outer Products matrix multiply in Finch

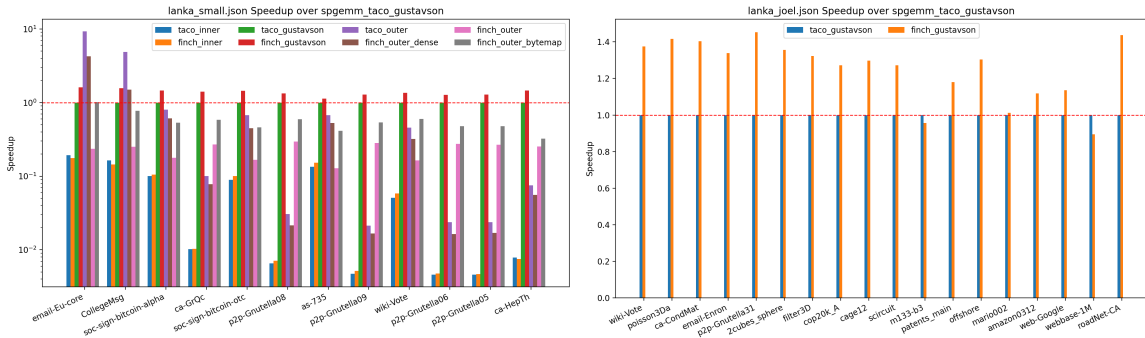


Fig. 18. A comparison of several matrix multiply algorithms between Finch and Taco. On left, we use smaller matrices, ordered from small to big dimension. Note that inner-products necessarily requires $O(MN)$ work and TACO’s outer-products format is dense. Finch can use a sparse outer products format and thus has an asymptotic advantage that becomes evident as the output dimensions grow. On right, we use only Gustavson’s algorithm and compare on larger matrices.

does not support multidimensional sparse workspaces, its outer products uses a dense intermediate, which leads to an asymptotic slow down shown in Figure 18. Similarly, although a sparse bytemap has a dense memory footprint, we use it in our Finch implementation of Gustavson’s for the smaller $O(N)$ intermediate. We note that the bytemap format in TACO’s Gustavson’s implementation is hard-wired, whereas Finch’s programming model allows us to write algorithms with explicit temporaries and transpositions. Without such hard wiring, TACO would have to use a dense intermediate to support random writes, which TACO would then propagate to the output, turning it dense and leading to the same asymptotic results as in the case of outer products. As depicted in Figure 18, 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 on larger matrices, with an average speedup of 1.25.

6.3 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. [94], including both road networks and scale-free graphs (bounded node degree vs. power law node degree).

Direction-optimization [14] is crucial for achieving high BFS performance in such scenarios, switching between push and pull traversals 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.

Figure 20 compares performance to Graphs.jl, a Julia library, and the LAGraph Library, which implements graph algorithms with sparse linear algebra using GraphBLAS [63]. For the BFS algorithm, direction-optimization notably enhances performance for scale-free graphs. Although GraphBLAS uses hardwired optimizations, Finch is competitive. On Bellman-Ford (with path lengths and shortest-path tree), Finch’s support for multiple outputs, sparse inputs, and masks leads to superior performance over GraphBLAS (average speedup of 3.53). We did not include GAP-road as it timed out. In Appendix B, we display the code for BFS and Bellman-Ford in Finch (57 and 50 LOC) and LAGraph (215 and 227 LOC), and invite readers to compare the clarity of the algorithms.

```

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
      end
    end
    return _F
  end
end

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
          end
        end
        if p[] != 0
          _F[k] |= true
          P[k] = p[]
        end
      end
    end
    return _F
  end
end

_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]
          end
        end
      end
    end
  end
end

```

Fig. 19. 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*.

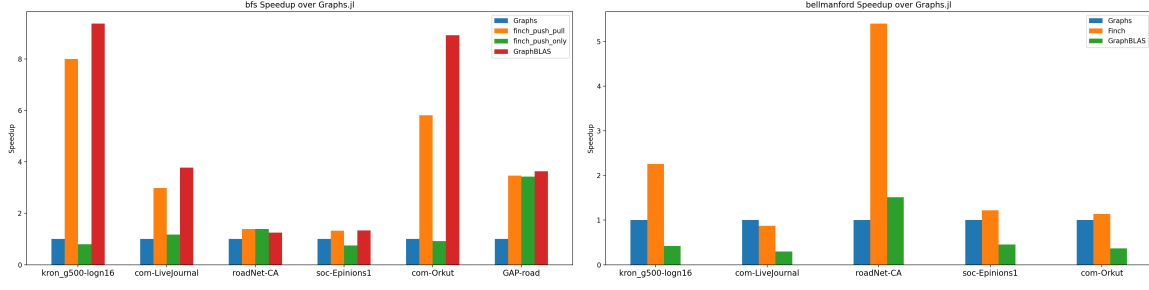


Fig. 20. Performance of graph apps across various tools. `finch_push_only` exclusively utilizes push traversal, while `finch_push_pull` applies direction-optimization akin to GraphBLAS. Finch’s support for push/pull traversal and early break facilitates direction-optimization. Among GraphBLAS’s five variants for Bellman-Ford, we selected LAGraph_BF_full1a, consistently the fastest with our graphs.

6.4 Image Morphology

Some image processing pipelines stand to benefit from structured data processing [31]. We focus on binary image morphology and the logical transformation of binary images and masks. We consider two operations: binary erosion (computing a mask), and a masked histogram (using a mask to avoid work). We use images are all binary, either by design or having been thresholded.

Finch allows us 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 (*SparseRLE(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.

Similarly, since Finch let’s us choose our algorithm 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 [35]. 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 might understand the filter as a structured tensor of circular shifts, or we might 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 21 displays example erosion algorithms for bitwise or run-length-encoded algorithms.

We compared against OpenCV. We used four datasets. We randomly selected 100 images from the MNIST [59] and Omniglot [57] character recognition datasets, as well as a dataset of human line drawings [33]. We also hand-selected a subset of mask images (these images were less homogeneous, so we listed them in Appendix C) from a digital image processing textbook [40]. 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 SparseRLE 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). 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, allowing us to implement the bitwise kernel and then mask it.

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

Masked Histogram:

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

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)
    end
  end
  for x = _
    if mask[x, y]
      let t1 = 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) | (t1 >> (8 * sizeof(UInt) - 1)))
      output[x, y] = res
    end
  end
end
```

Fig. 21. Two approaches to erosion in Finch. The *coalesce* function defines the out of bounds value. On left, the naive approach. On *SparseRLE(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.

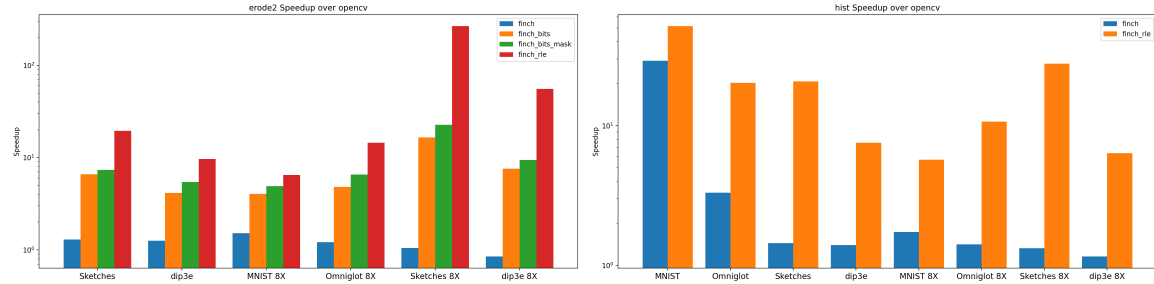


Fig. 22. Performance of Finch on image morphology tasks. On left, we run 2 iterations of erosion. On right, we run a masked histogram.

We also implemented a histogram kernel. We used an indirect access into the output to implement this (Figure 21), something not many sparse frameworks support. We compare to OpenCV since the OpenCV histogram function also accepts a mask. If we use *SparseRLE(Pattern())* for our mask, we can reduce the branching in the masked kernel and get better performance. The improvements with SparseRLE 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).

6.5 Implementing NumPy’s Array API in Finch

In the past decade, the adoption of the Python Array API [44] has allowed for a proliferation array programming systems, but existing implementations of this and similar APIs for structured data suffer from either incompleteness or inefficiency. These APIs have hundreds of required functions, from mapreduce to slicing and windowing. Compilers with simpler interfaces (such as TACO’s simple Einsum) need a massive amount of glue code outside of the normal compilation path to support the full API, leading to a large implementation burden. As the API’s were designed for only one structure (Dense), the burden only grows with the introduction of as additional structures such as sparsity, as these structures interact with the complex interface and with each other.

Further, implementing each operation on its own is insufficient. Operator fusion is required to avoid asymptotic performance degradation on sparse kernels, as we show in Figure 25. A flexible compiler like Finch, which can produce efficient code for arbitrary operations between inputs in a wide variety of formats, is the missing piece needed to implement the full array API for structured data with competitive performance. To handle the expansiveness of the Array API while preserving opportunities for fusion and whole workflow optimization, we pursued a lazy evaluation strategy mediated by a high-level query language. This is implemented by 1) Finch Logic, a minimal, high-level language for expressing array operations and 2) the Physical Interpreter, which executes Finch Logic as a sequence of Finch programs.

6.5.1 Finch Logic and the Physical Interpreter. Finch Logic is used as a common representation to implement and fuse the operations in the Array API. The syntax is designed to be minimal and expressive, taking inspiration from relational algebra with the goal of being able to express most of the operations in the Array API.

```

EXPR := table(TENSOR, FIELDS...) |
      ALIAS | FIELD | mapjoin(OP, EXPRS...) |
      aggregate(OP, INIT, EXPR, FIELDS...) |
      reorder(EXPR, FIELDS...) |
      relabel(EXPR, FIELDS...) |
      reformat(TENSOR, EXPR)
#TENSOR is a tensor value
#OP is a function
#INIT is an initial value
#FIELD is a name for an index
#ALIAS is a name for a tensor
PLAN := plan(QUERIES...)
QUERY := query(ALIAS, EXPR)

```

Fig. 23. Finch Logic Syntax.

Finch Logic. The syntax of Finch Logic is described in Figure 23. Our `mapjoin` represents pointwise application, while `aggregate` represents reductions. We name the result of each computation with `query`, and join several expressions in a `plan`. `reorder` reorders the indices while `relabel` relabels them. `reformat` hints at materialization into its argument.

Finch’s support for arbitrary user-defined functions allows us to support complex operations such as `argmin`, `norm`, and `where`, all over structured data. Finch’s support for wrapper arrays allows us to express indexing operations. These situations are shown in Figure 24.

Heuristic Optimization. Once we express our operation in Finch Logic, we use a quick heuristic to fuse the expressions. We heuristically split nested aggregate queries into separate queries, and fuse all of the `mapjoins` into corresponding aggregates. We then heuristically choose a loop order for each query with `reorder` nodes, and introduce separate query nodes to transpose so that the resulting expressions are concordant. Our heuristic assigns a format to each output by recursively evaluating the properties of structure (sparsity, repetition, etc.) and checking whether they are preserved under the operation in question.

Finch Interpreter. Once the program is in a standard form where each query has a specified format with `reformat` and at most one aggregate, the Finch Interpreter executes each query, in order, through a straightforward lowering process.

6.5.2 Evaluation. To demonstrate the performance of our array implementation, we evaluate it on 1) triangle counting 2) SDDMM 3) and element-wise operations. Further, we compare against DuckDB as a state of the art system which implements a form of kernel fusion through pipelined

```

sum(A, dims=2) = aggregate(+, relabel(A, i_1, ..., i_d), i_2)
matmul(A, dims=2) = aggregate(+, mapjoin(*, relabel(A, i, j), relabel(B, j, k)))
argmin(A) = aggregate(minby, (Inf, len(A)), mapjoin(tuple, relabel(A, i), i), i)
norm2(A) = aggregate(scaled_plus, (0, 0), mapjoin(tuple, mapjoin(sign, A), mapjoin(abs, A)), fields(A)...)
where(C, A, B) = mapjoin(ifelse, relabel(C, i), relabel(A, i), relabel(B, i))
slice(A, i:j) = table(WindowedArray(A, i:j), k)
cat(A, B) = mapjoin(coalesce, table(PermissiveArray(A), i), table(OffsetArray(B, length(A)), i))

```

Fig. 24. Implementations of a few common functions in Finch Logic. Here, `minby(x, y)` compares `x[1]` and `y[1]` and returns the smaller `x` or `y`. `scaled_plus` rescales whichever argument is smaller, then adds the values with the same scale. `ifelse(c, a, b)` returns `a` when `c` is true, `b` otherwise. Finch can understand declared properties and rewrite rules for all of these functions and types, extending these algorithms to operate on structured arguments.

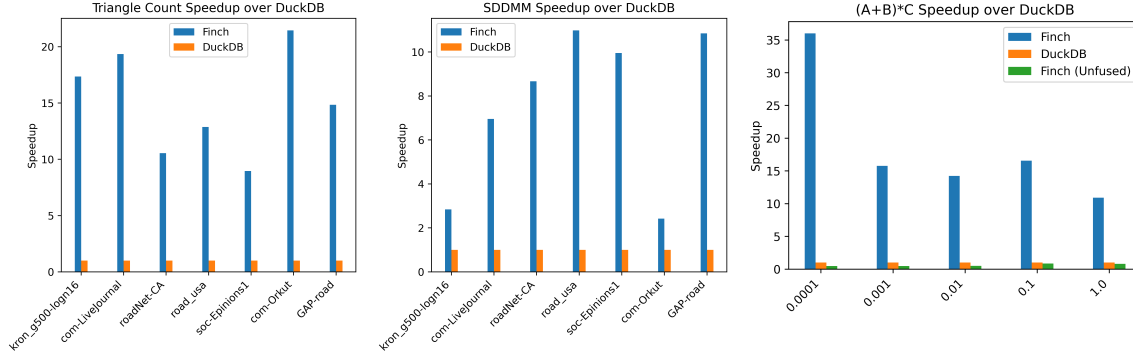


Fig. 25. Performance of Finch Logic for common kernels.

query execution. To do this, we express each of these kernels as a single select, join, groupby query. For the element-wise case, we provide an unfused Finch method to show the impact of fusion. For triangle counting, we use the same set of graph matrices as in Figure 20. For SDDMM, we use this set of graph matrices for the sparse matrix, and we produce random dense matrices with embedding dimension 25. Lastly, for the elementwise operations, we use uniformly sparse matrices with dimension 10000 by 10000. A/B have sparsity .1, and we vary the sparsity of C in the X axis.

Across all three of these kernels, we see that the high level interface for Finch provides a major improvement over DuckDB, ranging from $1.2x$ – $28x$. For triangle counting and SDDMM, this improvement stems from Finch’s efficient intersection of the nonzero indices as opposed to DuckDB’s use of binary join plans¹. For element-wise operations, this improvement stems from Finch’s better handling of expressions which combine index intersection and union and its compressed data representation.

7 RELATED WORK

The related work on array languages and libraries spans several areas, from libraries to languages, from dense to structured computation.

Libraries for Dense Data: Many libraries specialize in dense computations. Perhaps the most well-known example is NumPy [44], and a classic example is the BLAS, though several BLAS routines are specialized to symmetric, hermitian, and triangular matrices [9]. Many research projects have advanced on BLAS, such as BatchedBlas and BLIS [32, 86].

Libraries for Structured Data: Many libraries support BLAS plus a few sparse array types, typically CSR, CSC, BCSR, Banded, and COO. Examples include SciPy [88], PETSc [5], Armadillo [71], OSKI [90], Cyclops [79], MKL [1], and Eigen [42]. There are even libraries for very specific kernels and format combinations, such as SPLATT [77] (MTTKRP on CSF). Several of these libraries also feature some graph or mesh algorithms built on sparse matrices. The GraphBLAS [52] 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 [63]. Similarly, the MapReduce and Hadoop platforms support operations on indexed collections [29], and have been used to support graph algorithms in the GBASE library [51]. Several machine learning frameworks support some sparse arrays and operations, most notably TorchSparse [83, 84].

¹This matches with findings in the database literature showing that worst-case optimal joins (which are very similar to our kernel execution) are more efficient than binary joins for these queries [91].

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 [70] and its various descendant such as TVM [24], Exo [49], Elevate [43], and ATL [60]. 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 [41], Tiramisu [11], CHiLL [23], Pluto [19], and AlphaZ [96] offer similar capabilities in terms of control flow though they often support more irregular regions than the polyhedral framework supports. These are based on ISL [87]. 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 [54], but has been extended several times to support (single dimensional) local tensors [53], imperfectly nested loops [30], breaks via semi-rings [45], windowing and tiling [74], and convolution [92], and compilation in MLIR [17], 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 [26], and tree like structures [25], as separate extensions. SparseTIR is a similar system supporting combined sparse formats (including block structures) [95]. The SDQL language offers a similar level of control flow [75], 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 [73] and separately by another system that describes static symmetries and other structures as predicates [38]. The Taichi language focuses on a single sparse data structure made from dense blocks, bit-masks, and pointers [47]. The sparse polyhedral framework builds on CHiLL for the purpose of generating inspector/executor optimizations [81] 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 [98]. 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 [28, 56]. However, many SQL or dataframe systems rely on b-trees, columnar, or hash tables, with only a few systems, such as Vectorwise [18], LaraDB [48], GMAP [85], or SciDB [80] building physical layouts with other constructs based in array programming. However, array based databases are a new focus given the rise of mixed ML/DB pipelines [13, 62]. 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 [36, 37].

Other Architectures: Sparse compilers have been extended to many architectures. An extension of TACO supports GPU [74], Cyclops [78, 79] and SPDISTAL [93] support distributed memory, and the Sparse Abstract Machine [46] supports custom hardware. We believe that supporting control flow is the first step towards architectural support beyond unstructured sparsity.

8 CONCLUSION

Finch automatically specializes flexible control flow to diverse data structures, facilitating productive algorithmic exploration, flexible array 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

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] Martin 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 2023)*. 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 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. <https://doi.org/10.1145/3519939.3523442>
- [9] 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>
- [10] 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>
- [11] 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.
- [12] 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).
- [13] Peter Baumann, Dimitar Misev, Vlad Merticariu, and Bang Pham Huu. 2021. Array databases: Concepts, standards, implementations. *Journal of Big Data* 8 (2021), 1–61. Publisher: Springer.
- [14] Scott Beamer, Krste Asanovic, 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.
- [15] 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.
- [16] 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. Publisher: ACM New York, NY, USA.
- [17] 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>
- [18] PA Boncz and M Zukowski. 2012. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27.
- [19] 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.
- [20] Gary Bradski, Adrian Kaehler, and others. 2000. OpenCV. *Dr. Dobb's journal of software tools* 3, 2 (2000).

- [21] 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.
- [22] 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.
- [23] 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).
- [24] 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>
- [25] 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>
- [26] 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>
- [27] 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.
- [28] Chris J Date. 1989. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc.
- [29] 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>
- [30] 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>
- [31] 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>
- [32] 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>
- [33] 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>
- [34] 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
- [35] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. 1996. Hypermedia image processing reference. *England: John Wiley & Sons Ltd* (1996), 118–130.
- [36] 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.
- [37] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, 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>
- [38] 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>
- [39] Solomon Golomb. 1966. Run-length encodings (corresp.). *IEEE transactions on information theory* 12, 3 (1966), 399–401. Publisher: Citeseer.
- [40] Rafael C. Gonzalez and Richard E. Woods. 2006. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., USA.
- [41] 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. Publisher: World Scientific.
- [42] Gaël Guennebaud, Benoît Jacob, and others. 2010. Eigen v3. <http://eigen.tuxfamily.org>
- [43] Bastian Hagedorn, Johannes Lenfers, Thomas Köhler, 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>

- [44] 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.
- [45] 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>
- [46] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjolstad. 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>
- [47] 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>
- [48] 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>
- [49] 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>
- [50] 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
- [51] 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.
- [52] 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>
- [53] 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)*. 180–192. <https://doi.org/10.1109/CGO.2019.8661185>
- [54] 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>
- [55] Donald E. Knuth. 1997. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Addison-Wesley Professional. Google-Books-ID: x9AsAwAAQBAJ.
- [56] 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 Griebel, and Sergei Gorlatch (Eds.). Springer, Berlin, Heidelberg, 318–327. <https://doi.org/10.1007/BFb0002751>
- [57] 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.
- [58] 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.
- [59] 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.
- [60] 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>
- [61] 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

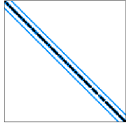
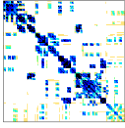
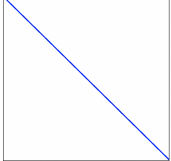
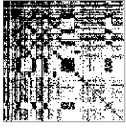
- Computing Machinery, New York, NY, USA, 339–350. <https://doi.org/10.1145/2751205.2751209>
- [62] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2018. Scalable linear algebra on a relational database system. *ACM SIGMOD Record* 47, 1 (2018), 24–31. Publisher: ACM New York, NY, USA.
 - [63] 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.
 - [64] 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>
 - [65] Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, and Simone Campanoni. [n. d.]. Representing Data Collections in an SSA Form. ([n. d.]).
 - [66] 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>
 - [67] Dianne P O'Leary. 2009. *Scientific computing with case studies*. SIAM.
 - [68] 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>
 - [69] 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].
 - [70] 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>
 - [71] 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> Publisher: The Open Journal.
 - [72] Yousef Saad. 2003. *Iterative methods for sparse linear systems* (2nd ed.). SIAM, Philadelphia.
 - [73] Maximilian Schleich, Amir Shaikhha, and Dan Suci. 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>
 - [74] 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>
 - [75] 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>
 - [76] 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>
 - [77] 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>
 - [78] 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.
 - [79] 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.
 - [80] 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.
 - [81] 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.
 - [82] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. Publisher: Ieee.

- [83] 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
- [84] 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.
- [85] 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.
- [86] 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>
- [87] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302.
- [88] 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.
- [89] 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.
- [90] 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.
- [91] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free join: Unifying worst-case optimal and traditional joins. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–23. Publisher: ACM New York, NY, USA.
- [92] 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 (2023).
- [93] 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.
- [94] 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>
- [95] 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>
- [96] 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.
- [97] 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>
- [98] 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. Publisher: ACM New York, NY.
- [99] 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 20 February 2007; revised 12 March 2009; accepted 5 June 2009

A SPMV DATASETS

Table 1. SpMV Sample Matrices

	HB/saylr4	Norris/heart3	large_band	SNAP/as-735
Spy				
Group / Name	HB/saylr4	Norris/heart3	large_band	SNAP/as-735
Dimensions	3,564 x 3,564 (22,316)	2,339 x 2,339 (680,341)	10,000 x 10,000 (1,999,900)	7,716 x 7,716 (26,467)
Best Finch Format	Symmetric SparseList	SparseVBL	SparseBand	Symmetric SparseList-Pattern

B GRAPH ALGORITHM LISTINGS

B.1 Finch Breadth-First Search

```

function bfs_finch_kernel(edges, edgesT, source=5, alpha = 0.01)
  (n, m) = size(edges)
  edges = pattern!(edges)
  @assert n == m
  F = Tensor(SparseByteMap(Pattern()), n)
  _F = Tensor(SparseByteMap(Pattern()), n)
  @finch F[source] = true
  F_nnz = 1
  V = Tensor(Dense(Element(false)), n)
  @finch V[source] = true
  P = Tensor(Dense(Element(0)), n)
  @finch P[source] = source
  while F_nnz > 0
    if F_nnz/m > alpha # pull
      p = ShortCircuitScalar{0}()
      _F .= false
      for k=_
        if !V[k]
          p .= 0
          for j=_
            if F[follow(j)] && AT[j, k]
              p[] <<choose(0)>>= j
            end
          end
          if p[] != 0
            _F[k] |= true
            P[k] = p[]
          end
        end
      end
    else # push
      _F .= false
      for j=_, k=_
        if F[j] && A[k, j] && !(V[k])
          _F[k] |= true
          P[k] <<choose(0)>>= j
        end
      end
    end
    c = Scalar{0}
    @finch begin
      for k=_
        let _f = _F[k]
        V[k] |= _f
        c[] += _f
      end
    end
    (F, _F) = (_F, F)
    F_nnz = c[]
  end
  return P
end

```

B.2 GraphBLAS Breadth-First Search

```
//-----
// LAGr_BreadthFirstSearch: breadth-first search dispatch
//-----

// LAGraph, (c) 2019-2022 by The LAGraph Contributors, All Rights Reserved.
// SPDX-License-Identifier: BSD-2-Clause
//
// For additional details (including references to third party source code and
// other files) see the LICENSE file or contact permission@sei.cmu.edu. See
// Contributors.txt for a full list of contributors. Created, in part, with
// funding and support from the U.S. Government (see Acknowledgments.txt file).
// DM22-0790

// Contributed by Scott McMillan, SEI Carnegie Mellon University

//-----

// Breadth-first-search via push/pull method if using SuiteSparse:GraphBLAS
// and its GxB extensions, or a push-only method otherwise. The former is
// much faster.

// This is an Advanced algorithm. SuiteSparse can use a push/pull method if
// G->AT and G->out_degree are provided. G->AT is not required if G is
// undirected. The vanilla method is always push-only.

#include "LG_alg_internal.h"

int LAGr_BreadthFirstSearch
(
    // output:
    GrB_Vector *level,
    GrB_Vector *parent,
    // input:
    const LAGraph_Graph G,
    GrB_Index src,
    char *msg
)
{
    #if LAGRAPH_SUITESPARSE
        return LG_BreadthFirstSearch_SSGrB (level, parent, G, src, msg);
    #else
        return LG_BreadthFirstSearch_vanilla (level, parent, G, src, msg);
    #endif
}

//-----
// LG_BreadthFirstSearch_SSGrB: BFS using Suitesparse extensions
//-----

// LAGraph, (c) 2019-2022 by The LAGraph Contributors, All Rights Reserved.
// SPDX-License-Identifier: BSD-2-Clause
//
// For additional details (including references to third party source code and
// other files) see the LICENSE file or contact permission@sei.cmu.edu. See
// Contributors.txt for a full list of contributors. Created, in part, with
// funding and support from the U.S. Government (see Acknowledgments.txt file).
// DM22-0790

// Contributed by Timothy A. Davis, Texas A&M University

//-----

// This is an Advanced algorithm. G->AT and G->out_degree are required for
// this method to use push-pull optimization. If not provided, this method
// defaults to a push-only algorithm, which can be slower. This is not
// user-callable (see LAGr_BreadthFirstSearch instead). G->AT and
// G->out_degree are not computed if not present.

// References:
//
// Carl Yang, Aydin Buluc, and John D. Owens. 2018. Implementing Push-Pull
// Efficiently in GraphBLAS. In Proceedings of the 47th International
// Conference on Parallel Processing (ICPP 2018). ACM, New York, NY, USA,
// Article 89, 11 pages. DOI: https://doi.org/10.1145/3225058.3225122
//
// Scott Beamer, Krste Asanovic and David A. Patterson, The GAP Benchmark
// Suite, http://arxiv.org/abs/1508.03619, 2015. http://gap.cs.berkeley.edu/
```

```

// revised by Tim Davis (davis@tamu.edu), Texas A&M University

#define LG_FREE_WORK      \
{                          \
    GrB_free (&w) ;        \
    GrB_free (&q) ;        \
}

#define LG_FREE_ALL      \
{                          \
    LG_FREE_WORK ;        \
    GrB_free (&pi) ;      \
    GrB_free (&v) ;        \
}

#include "LG_internal.h"

int LG_BreadthFirstSearch_SGrB
(
    GrB_Vector *level,
    GrB_Vector *parent,
    const LAGraph_Graph G,
    GrB_Index src,
    char *msg
)
{
    //-----
    // check inputs
    //-----

    LG_CLEAR_MSG ;
    GrB_Vector q = NULL ;           // the current frontier
    GrB_Vector w = NULL ;           // to compute work remaining
    GrB_Vector pi = NULL ;          // parent vector
    GrB_Vector v = NULL ;           // level vector

    #if !LAGRAPH_SUITESPARSE
        LG_ASSERT (false, GrB_NOT_IMPLEMENTED) ;
    #else

        bool compute_level = (level != NULL) ;
        bool compute_parent = (parent != NULL) ;
        if (compute_level) (*level) = NULL ;
        if (compute_parent) (*parent) = NULL ;
        LG_ASSERT_MSG (compute_level || compute_parent, GrB_NULL_POINTER,
            "either level or parent must be non-NULL") ;

        LG_TRY (LAGraph_CheckGraph (G, msg)) ;

        //-----
        // get the problem size and cached properties
        //-----

        GrB_Matrix A = G->A ;

        GrB_Index n, nvals ;
        GRB_TRY (GrB_Matrix_nrows (&n, A)) ;
        LG_ASSERT_MSG (src < n, GrB_INVALID_INDEX, "invalid source node") ;

        GRB_TRY (GrB_Matrix_nvals (&nvals, A)) ;

        GrB_Matrix AT = NULL ;
        GrB_Vector Degree = G->out_degree ;
        if (G->kind == LAGraph_ADJACENCY_UNDIRECTED ||
            (G->kind == LAGraph_ADJACENCY_DIRECTED &&
             G->is_symmetric_structure == LAGraph_TRUE))
        {
            // AT and A have the same structure and can be used in both directions
            AT = G->A ;
        }
        else
        {
            // AT = A' is different from A.  If G->AT is NULL, then a push-only
            // method is used.
            AT = G->AT ;
        }

        // direction-optimization requires G->AT (if G is directed) and
        // G->out_degree (for both undirected and directed cases)
        bool push_pull = (Degree != NULL && AT != NULL) ;

```

```

// determine the semiring type
GrB_Type int_type = (n > INT32_MAX) ? GrB_INT64 : GrB_INT32 ;
GrB_Semiring semiring ;

if (compute_parent)
{
    // use the ANY_SECONDINT* semiring: either 32 or 64-bit depending on
    // the # of nodes in the graph.
    semiring = (n > INT32_MAX) ?
        GxB_ANY_SECONDINT64 : GxB_ANY_SECONDINT32 ;

    // create the parent vector. pi(i) is the parent id of node i
    GRB_TRY (GrB_Vector_new (&pi, int_type, n)) ;
    GRB_TRY (GxB_set (pi, GxB_SPARSITY_CONTROL, GxB_BITMAP + GxB_FULL)) ;
    // pi (src) = src denotes the root of the BFS tree
    GRB_TRY (GrB_Vector_setElement (pi, src, src)) ;

    // create a sparse integer vector q, and set q(src) = src
    GRB_TRY (GrB_Vector_new (&q, int_type, n)) ;
    GRB_TRY (GrB_Vector_setElement (q, src, src)) ;
}
else
{
    // only the level is needed, use the LAGraph_any_one_bool semiring
    semiring = LAGraph_any_one_bool ;

    // create a sparse boolean vector q, and set q(src) = true
    GRB_TRY (GrB_Vector_new (&q, GrB_BOOL, n)) ;
    GRB_TRY (GrB_Vector_setElement (q, true, src)) ;
}

if (compute_level)
{
    // create the level vector. v(i) is the level of node i
    // v (src) = 0 denotes the source node
    GRB_TRY (GrB_Vector_new (&v, int_type, n)) ;
    GRB_TRY (GxB_set (v, GxB_SPARSITY_CONTROL, GxB_BITMAP + GxB_FULL)) ;
    GRB_TRY (GrB_Vector_setElement (v, 0, src)) ;
}

// workspace for computing work remaining
GRB_TRY (GrB_Vector_new (&w, GrB_INT64, n)) ;

GrB_Index nq = 1 ;           // number of nodes in the current level
double alpha = 8.0 ;
double beta1 = 8.0 ;
double beta2 = 512.0 ;
int64_t n_over_beta1 = (int64_t) (((double) n) / beta1) ;
int64_t n_over_beta2 = (int64_t) (((double) n) / beta2) ;

//-----
// BFS traversal and label the nodes
//-----

bool do_push = true ;        // start with push
GrB_Index last_nq = 0 ;
int64_t edges_unexplored = nvals ;
bool any_pull = false ;      // true if any pull phase has been done

// {!mask} is the set of unvisited nodes
GrB_Vector mask = (compute_parent) ? pi : v ;

for (int64_t nvisited = 1, k = 1 ; nvisited < n ; nvisited += nq, k++)
{
    //-----
    // select push vs pull
    //-----

    if (push_pull)
    {
        if (do_push)
        {
            // check for switch from push to pull
            bool growing = nq > last_nq ;
            bool switch_to_pull = false ;
            if (edges_unexplored < n)
            {
                // very little of the graph is left; disable the pull
                push_pull = false ;
            }
        }
    }
}

```

```

    }
    else if (any_pull)
    {
        // once any pull phase has been done, the # of edges in the
        // frontier has no longer been tracked. But now the BFS
        // has switched back to push, and we're checking for yet
        // another switch to pull. This switch is unlikely, so
        // just keep track of the size of the frontier, and switch
        // if it starts growing again and is getting big.
        switch_to_pull = (growing && nq > n_over_beta1) ;
    }
    else
    {
        // update the # of unexplored edges
        // w<q>=Degree
        // w(i) = outdegree of node i if node i is in the queue
        GRB_TRY (GrB_assign (w, q, NULL, Degree, GrB_ALL, n,
            GrB_DESC_RS)) ;
        // edges_in_frontier = sum (w) = # of edges incident on all
        // nodes in the current frontier
        int64_t edges_in_frontier = 0 ;
        GRB_TRY (GrB_reduce (&edges_in_frontier, NULL,
            GrB_PLUS_MONOID_INT64, w, NULL)) ;
        edges_unexplored -= edges_in_frontier ;
        switch_to_pull = growing &&
            (edges_in_frontier > (edges_unexplored / alpha)) ;
    }
    if (switch_to_pull)
    {
        // switch from push to pull
        do_push = false ;
    }
}
else
{
    // check for switch from pull to push
    bool shrinking = nq < last_nq ;
    if (shrinking && (nq <= n_over_beta2))
    {
        // switch from pull to push
        do_push = true ;
    }
}
any_pull = any_pull || (!do_push) ;
}

//-----
// q = kth level of the BFS
//-----

int sparsity = do_push ? GxB_SPARSE : GxB_BITMAP ;
GRB_TRY (GxB_set (q, GxB_SPARSITY_CONTROL, sparsity)) ;

// mask is pi if computing parent, v if computing just level
if (do_push)
{
    // push (saxpy-based vxm): q[!mask] = q'*A
    GRB_TRY (GrB_vxm (q, mask, NULL, semiring, q, A, GrB_DESC_RSC)) ;
}
else
{
    // pull (dot-product-based mxv): q[!mask] = AT*q
    GRB_TRY (GrB_mxv (q, mask, NULL, semiring, AT, q, GrB_DESC_RSC)) ;
}

//-----
// done if q is empty
//-----

last_nq = nq ;
GRB_TRY (GrB_Vector_nvals (&nq, q)) ;
if (nq == 0)
{
    break ;
}

//-----
// assign parents/levels
//-----

if (compute_parent)

```

```

    {
        // q(i) currently contains the parent id of node i in tree.
        // pi{q} = q
        GRB_TRY (GrB_assign (pi, q, NULL, q, GrB_ALL, n, GrB_DESC_S)) ;
    }
    if (compute_level)
    {
        // v{q} = k, the kth level of the BFS
        GRB_TRY (GrB_assign (v, q, NULL, k, GrB_ALL, n, GrB_DESC_S)) ;
    }
}

//-----
// free workspace and return result
//-----

if (compute_parent) (*parent) = pi ;
if (compute_level) (*level) = v ;
LG_FREE_WORK ;
return (GrB_SUCCESS) ;
#endif
}

```

B.3 Finch Bellman-Ford

```

function bellmanford_finch_kernel(edges, source=1)
    (n, m) = size(edges)
    @assert n == m
    dists_prev = Tensor(Dense(Element(Inf)), n)
    dists_prev[source] = 0
    dists = Tensor(Dense(Element(Inf)), n)
    active_prev = Tensor(SparseByteMap(Pattern()), n)
    active_prev[source] = true
    active = Tensor(SparseByteMap(Pattern()), n)
    parents = Tensor(Dense(Element(0)), n)
    for iter = 1:n
        @finch begin
            for j = _
                if active_prev[j]
                    dists[j] <<min>>= dists_prev[j]
                end
            end
        end
        @finch begin
            active .= false
            for j = _
                if active_prev[j]
                    for i = _
                        let d = dists_prev[j] + edges[i, j]
                            dists[i] <<min>>= d
                            active[i] |= d < dists_prev[i]
                        end
                    end
                end
            end
            if countstored(active) == 0
                break
            end
            dists_prev, dists = dists, dists_prev
            active_prev, active = active, active_prev
        end
        @finch begin
            for j = _
                for i = _
                    let d = edges[i, j]
                        if d < Inf && dists[j] + d <= dists[i]
                            parents[i] <<choose(0)>>= j
                        end
                    end
                end
            end
        end
        return (dists=dists, parents=parents)
    end
end

```

B.4 GraphBLAS Bellman-Ford

```

//-----
// LAGraph_BF_full11a.c: Bellman-Ford single-source shortest paths, returns tree,

```

```

// while diagonal of input matrix A needs not to be explicit 0
//-----

// LAGraph, (c) 2019-2022 by The LAGraph Contributors, All Rights Reserved.
// SPDX-License-Identifier: BSD-2-Clause
//
// For additional details (including references to third party source code and
// other files) see the LICENSE file or contact permission@sei.cmu.edu. See
// Contributors.txt for a full list of contributors. Created, in part, with
// funding and support from the U.S. Government (see Acknowledgments.txt file).
// DM22-0790

// Contributed by Jinhao Chen and Timothy A. Davis, Texas A&M University
//-----

// This is the fastest variant that computes both the parent & the path length.

// LAGraph_BF_full1a: Bellman-Ford single source shortest paths, returning both
// the path lengths and the shortest-path tree.

// LAGraph_BF_full performs a Bellman-Ford to find out shortest path, parent
// nodes along the path and the hops (number of edges) in the path from given
// source vertex s in the range of [0, n) on graph given as matrix A with size
// n*n. The sparse matrix A has entry A(i, j) if there is an edge from vertex i
// to vertex j with weight w, then A(i, j) = w.

// LAGraph_BF_full1a returns GrB_SUCCESS if it succeeds. In this case, there
// are no negative-weight cycles in the graph, and d, pi, and h are returned.
// The vector d has d(k) as the shortest distance from s to k. pi(k) = p+1,
// where p is the parent node of k-th node in the shortest path. In particular,
// pi(s) = 0. h(k) = hop(s, k), the number of edges from s to k in the shortest
// path.

// If the graph has a negative-weight cycle, GrB_NO_VALUE is returned, and the
// GrB_Vectors d(k), pi(k) and h(k) (i.e., *pd_output, *ppi_output and
// *ph_output respectively) will be NULL when negative-weight cycle detected.

// Otherwise, other errors such as GrB_OUT_OF_MEMORY, GrB_INVALID_OBJECT, and
// so on, can be returned, if these errors are found by the underlying
// GrB_* functions.

//-----

#define LG_FREE_WORK \
{ \
    GrB_free(&d); \
    GrB_free(&dmasked); \
    GrB_free(&dless); \
    GrB_free(&Atmp); \
    GrB_free(&BF_Tuple3); \
    GrB_free(&BF_LMIN_Tuple3); \
    GrB_free(&BF_PLUSrhs_Tuple3); \
    GrB_free(&BF_LT_Tuple3); \
    GrB_free(&BF_LMIN_Tuple3_Monoid); \
    GrB_free(&BF_LMIN_PLUSrhs_Tuple3); \
    LAGraph_Free ((void*)&I, NULL); \
    LAGraph_Free ((void*)&J, NULL); \
    LAGraph_Free ((void*)&w, NULL); \
    LAGraph_Free ((void*)&W, NULL); \
    LAGraph_Free ((void*)&h, NULL); \
    LAGraph_Free ((void*)&pi, NULL); \
}

#define LG_FREE_ALL \
{ \
    LG_FREE_WORK ; \
    GrB_free (pd_output); \
    GrB_free (ppi_output); \
    GrB_free (ph_output); \
}

#include <LAGraph.h>
#include <LAGraphX.h>
#include <LG_internal.h> // from src/utility

typedef void (*LAGraph_binary_function) (void *, const void *, const void *) ;

//-----
// data type for each entry of the adjacent matrix A and "distance" vector d;
// <INFINITY,INFINITY,INFINITY> corresponds to nonexistence of a path, and

```

```

// the value <0, 0, NULL> corresponds to a path from a vertex to itself
//-----

typedef struct
{
    double w; // w corresponds to a path weight.
    GrB_Index h; // h corresponds to a path size or number of hops.
    GrB_Index pi; // pi corresponds to the penultimate vertex along a path.
                  // vertex indexed as 1, 2, 3, ... , V, and pi = 0 (as nil)
                  // for u=v, and pi = UINT64_MAX (as inf) for (u,v) not in E
}
BF_Tuple3_struct;

//-----
// binary functions, z=f(x,y), where Tuple3xTuple3 -> Tuple3
//-----

void BF_LMIN3
(
    BF_Tuple3_struct *z,
    const BF_Tuple3_struct *x,
    const BF_Tuple3_struct *y
)
{
    if (x->w < y->w
        || (x->w == y->w && x->h < y->h)
        || (x->w == y->w && x->h == y->h && x->pi < y->pi))
    {
        if (z != x) { *z = *x; }
    }
    else
    {
        *z = *y;
    }
}

void BF_PLUSrhs3
(
    BF_Tuple3_struct *z,
    const BF_Tuple3_struct *x,
    const BF_Tuple3_struct *y
)
{
    z->w = x->w + y->w ;
    z->h = x->h + y->h ;
    z->pi = (x->pi != UINT64_MAX && y->pi != 0) ? y->pi : x->pi ;
}

void BF_LT3
(
    bool *z,
    const BF_Tuple3_struct *x,
    const BF_Tuple3_struct *y
)
{
    (*z) = (x->w < y->w
        || (x->w == y->w && x->h < y->h)
        || (x->w == y->w && x->h == y->h && x->pi < y->pi)) ;
}

// Given a n-by-n adjacency matrix A and a source vertex s.
// If there is no negative-weight cycle reachable from s, return the distances
// of shortest paths from s and parents along the paths as vector d. Otherwise,
// returns d=NULL if there is a negative-weight cycle.
// pd_output is pointer to a GrB_Vector, where the i-th entry is d(s,i), the
// sum of edges length in the shortest path
// ppi_output is pointer to a GrB_Vector, where the i-th entry is pi(i), the
// parent of i-th vertex in the shortest path
// ph_output is pointer to a GrB_Vector, where the i-th entry is h(s,i), the
// number of edges from s to i in the shortest path
// A has weights on corresponding entries of edges
// s is given index for source vertex
GrB_Info LAGraph_BF_full1a
(
    GrB_Vector *pd_output, //the pointer to the vector of distance
    GrB_Vector *ppi_output, //the pointer to the vector of parent
    GrB_Vector *ph_output, //the pointer to the vector of hops
    const GrB_Matrix A, //matrix for the graph
    const GrB_Index s //given index of the source
)
{

```

```

GrB_Info info;
char *msg = NULL ;
// tmp vector to store distance vector after n (i.e., V) loops
GrB_Vector d = NULL, dmasked = NULL, dless = NULL;
GrB_Matrix Atmp = NULL;
GrB_Type BF_Tuple3;

GrB_BinaryOp BF_LMIN_Tuple3;
GrB_BinaryOp BF_PLUSrhs_Tuple3;
GrB_BinaryOp BF_LT_Tuple3;

GrB_Monoid BF_LMIN_Tuple3_Monoid;
GrB_Semiring BF_LMIN_PLUSrhs_Tuple3;

GrB_Index nrows, ncols, n, nz; // n = # of row/col, nz = # of nnz in graph
GrB_Index *I = NULL, *J = NULL; // for col/row indices of entries from A
GrB_Index *h = NULL, *pi = NULL;
double *w = NULL;
BF_Tuple3_struct *W = NULL;

if (pd_output != NULL) *pd_output = NULL;
if (ppi_output != NULL) *ppi_output = NULL;
if (ph_output != NULL) *ph_output = NULL;

LG_ASSERT (A != NULL && pd_output != NULL &&
  ppi_output != NULL && ph_output != NULL, GrB_NULL_POINTER) ;

GRB_TRY (GrB_Matrix_nrows (&nrows, A)) ;
GRB_TRY (GrB_Matrix_ncols (&ncols, A)) ;
GRB_TRY (GrB_Matrix_nvals (&nz, A));
LG_ASSERT_MSG (nrows == ncols, -1002, "A must be square") ;
n = nrows;
LG_ASSERT_MSG (s < n, GrB_INVALID_INDEX, "invalid source node") ;

//-----
// create all GrB_Type GrB_BinaryOp GrB_Monoid and GrB_Semiring
//-----
// GrB_Type
GRB_TRY (GrB_Type_new(&BF_Tuple3, sizeof(BF_Tuple3_struct)));

// GrB_BinaryOp
GRB_TRY (GrB_BinaryOp_new(&BF_LT_Tuple3,
  (LAGraph_binary_function) (&BF_LT3), GrB_BOOL, BF_Tuple3, BF_Tuple3));
GRB_TRY (GrB_BinaryOp_new(&BF_LMIN_Tuple3,
  (LAGraph_binary_function) (&BF_LMIN3), BF_Tuple3, BF_Tuple3, BF_Tuple3));
GRB_TRY (GrB_BinaryOp_new(&BF_PLUSrhs_Tuple3,
  (LAGraph_binary_function)(&BF_PLUSrhs3),
  BF_Tuple3, BF_Tuple3, BF_Tuple3));

// GrB_Monoid
BF_Tuple3_struct BF_identity = (BF_Tuple3_struct) { .w = INFINITY,
  .h = UINT64_MAX, .pi = UINT64_MAX };
GRB_TRY (GrB_Monoid_new_UDT(&BF_LMIN_Tuple3_Monoid, BF_LMIN_Tuple3,
  &BF_identity));

//GrB_Semiring
GRB_TRY (GrB_Semiring_new(&BF_LMIN_PLUSrhs_Tuple3,
  BF_LMIN_Tuple3_Monoid, BF_PLUSrhs_Tuple3));

//-----
// allocate arrays used for tuples
//-----
#if 1
LAGRAPH_TRY (LAGraph_Malloc ((void **) &I, nz, sizeof(GrB_Index), msg)) ;
LAGRAPH_TRY (LAGraph_Malloc ((void **) &J, nz, sizeof(GrB_Index), msg)) ;
LAGRAPH_TRY (LAGraph_Malloc ((void **) &w, nz, sizeof(double), msg)) ;
LAGRAPH_TRY (LAGraph_Malloc ((void **) &W, nz, sizeof(BF_Tuple3_struct),
  msg)) ;

//-----
// create matrix Atmp based on A, while its entries become BF_Tuple3 type
//-----

GRB_TRY (GrB_Matrix_extractTuples_FP64(I, J, w, &nz, A));
int nthreads, nthreads_outer, nthreads_inner ;
LG_TRY (LAGraph_GetNumThreads (&nthreads_outer, &nthreads_inner, msg)) ;
nthreads = nthreads_outer * nthreads_inner ;
printf ("nthreads %d\n", nthreads) ;
int64_t k;
#pragma omp parallel for num_threads(nthreads) schedule(static)
for (k = 0; k < nz; k++)

```

```

{
    W[k] = (BF_Tuple3_struct) { .w = w[k], .h = 1, .pi = I[k] + 1 };
}
GRB_TRY (GrB_Matrix_new(&Atmp, BF_Tuple3, n, n));
GRB_TRY (GrB_Matrix_build_UDT(Atmp, I, J, W, nz, BF_LMIN_Tuple3));
LAGraph_Free ((void**)&I, NULL);
LAGraph_Free ((void**)&J, NULL);
LAGraph_Free ((void**)&W, NULL);
LAGraph_Free ((void**)&w, NULL);

#else

todo: GraphBLAS could use a new kind of unary operator, not z=f(x), but

[z,flag] = f (a[ij, i, j, k, nrows, ncols, nvals, etc, ...])
flag: keep or discard. Combines GrB_apply and GxB_select.

builtins:
f(...) =
    i, bool is true
    j, bool is true
    i+j*nrows, etc.
    k
    tril, triu (like GxB_select): return a[ij, and true/false boolean

    z=f(x,i). x: double, z:tuple3, i:GrB_Index with the row index of x
    // z = (BF_Tuple3_struct) { .w = x, .h = 1, .pi = i + 1 };

GrB_apply (Atmp, op, A, ...)

in the BFS, this is used:
op: z = f ( ... ) = i
    to replace x(i) with i

#endif

//-----
// create and initialize "distance" vector d, dmasked and dless
//-----
GRB_TRY (GrB_Vector_new(&d, BF_Tuple3, n));
// make d dense
GRB_TRY (GrB_Vector_assign_UDT(d, NULL, NULL, (void*)&BF_identity,
    GrB_ALL, n, NULL));
// initial distance from s to itself
BF_Tuple3_struct d0 = (BF_Tuple3_struct) { .w = 0, .h = 0, .pi = 0 };
GRB_TRY (GrB_Vector_setElement_UDT(d, &d0, s));

// creat dmasked as a sparse vector with only one entry at s
GRB_TRY (GrB_Vector_new(&dmasked, BF_Tuple3, n));
GRB_TRY (GrB_Vector_setElement_UDT(dmasked, &d0, s));

// create dless
GRB_TRY (GrB_Vector_new(&dless, GrB_BOOL, n));

//-----
// start the Bellman Ford process
//-----
bool any_dless= true; // if there is any newly found shortest path
int64_t iter = 0; // number of iterations

// terminate when no new path is found or more than V-1 loops
while (any_dless && iter < n - 1)
{
    // execute semiring on dmasked and A, and save the result to dmasked
    GRB_TRY (GrB_vxm(dmasked, GrB_NULL, GrB_NULL,
        BF_LMIN_PLUSrhs_Tuple3, dmasked, Atmp, GrB_NULL));

    // dless = d .< dtmp
    GRB_TRY (GrB_eWiseMult(dless, NULL, NULL, BF_LT_Tuple3, dmasked, d,
        NULL));

    // if there is no entry with smaller distance then all shortest paths
    // are found
    GRB_TRY (GrB_reduce (&any_dless, NULL, GrB_LOR_MONOID_BOOL, dless,
        NULL));
    if(any_dless)
    {
        // update all entries with smaller distances
        //GRB_TRY (GrB_apply(d, dless, NULL, BF_Identity_Tuple3,
        //    dmasked, NULL));
        GRB_TRY (GrB_assign(d, dless, NULL, dmasked, GrB_ALL, n, NULL));
    }
}

```

```

        // only use entries that were just updated
        //GRB_TRY (GrB_Vector_clear(dmasked));
        //GRB_TRY (GrB_apply(dmasked, dless, NULL, BF_Identity_Tuple3,
        //    d, NULL));
        //try:
        GRB_TRY (GrB_assign(dmasked, dless, NULL, d, GrB_ALL, n, GrB_DESC_R));
    }
    iter ++;
}

// check for negative-weight cycle only when there was a new path in the
// last loop, otherwise, there can't be a negative-weight cycle.
if (any_dless)
{
    // execute semiring again to check for negative-weight cycle
    GRB_TRY (GrB_vxm(dmasked, GrB_NULL, GrB_NULL,
        BF_IMIN_PLUSrhs_Tuple3, dmasked, Atmp, GrB_NULL));

    // dless = d .< dtmp
    GRB_TRY (GrB_eWiseMult(dless, NULL, NULL, BF_LT_Tuple3, dmasked, d,
        NULL));

    // if there is no entry with smaller distance then all shortest paths
    // are found
    GRB_TRY (GrB_reduce (&any_dless, NULL, GrB_LOR_MONOID_BOOL, dless,
        NULL));
    if (any_dless)
    {
        // printf("A negative-weight cycle found. \n");
        LG_FREE_ALL;
        return (GrB_NO_VALUE);
    }
}

//-----
// extract tuple from "distance" vector d and create GrB_Vectors for output
//-----

LAGRAPH_TRY (LAGraph_Malloc ((void **) &I, n, sizeof(GrB_Index), msg));
LAGRAPH_TRY (LAGraph_Malloc ((void **) &W, n, sizeof(BF_Tuple3_struct),
    msg));
LAGRAPH_TRY (LAGraph_Malloc ((void **) &w, n, sizeof(double), msg));
LAGRAPH_TRY (LAGraph_Malloc ((void **) &h, n, sizeof(GrB_Index), msg));
LAGRAPH_TRY (LAGraph_Malloc ((void **) &pi, n, sizeof(GrB_Index), msg));

// todo: create 3 unary ops, and use GrB_apply?

GRB_TRY (GrB_Vector_extractTuples_UDT (I, (void *) W, &n, d));

for (k = 0; k < n; k++)
{
    w[k] = W[k].w;
    h[k] = W[k].h;
    pi[k] = W[k].pi;
}
GRB_TRY (GrB_Vector_new(pd_output, GrB_FP64, n));
GRB_TRY (GrB_Vector_new(ppi_output, GrB_UINT64, n));
GRB_TRY (GrB_Vector_new(ph_output, GrB_UINT64, n));
GRB_TRY (GrB_Vector_build (*pd_output, I, w, n, GrB_MIN_FP64));
GRB_TRY (GrB_Vector_build (*ppi_output, I, pi, n, GrB_MIN_UINT64));
GRB_TRY (GrB_Vector_build (*ph_output, I, h, n, GrB_MIN_UINT64));
LG_FREE_WORK;
return (GrB_SUCCESS);
}

```

C MASK IMAGES

We interpreted the following images from “Digital Image Processing” [1] as masks:

FigP1012.png

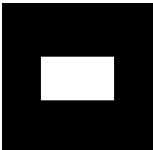
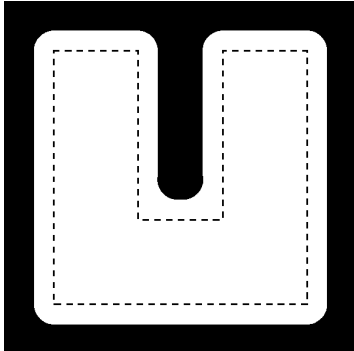


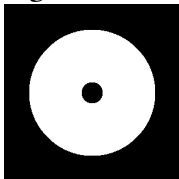
Fig1008_a_step_edge.png



FigP0905_d.png



FigP0528_c_doughnut.png



FigP0616_b.png

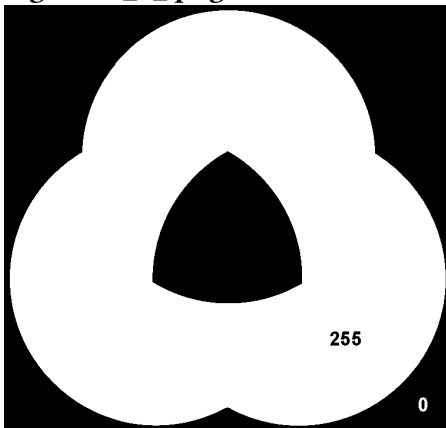
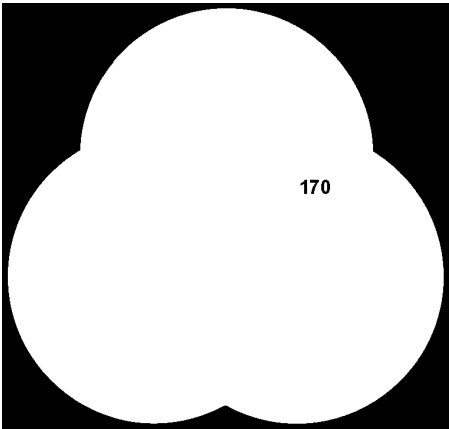


Fig0114_c_bottles.png



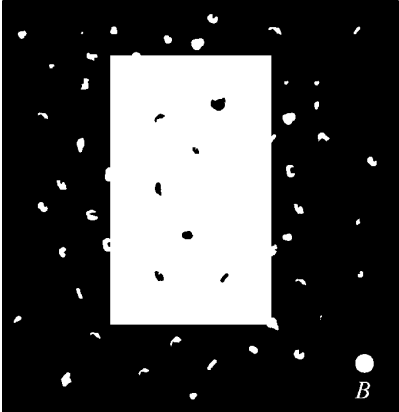
FigP0616_c.png



FigP0433_b_.png



Figp0917.png



FigP0905_b_.png

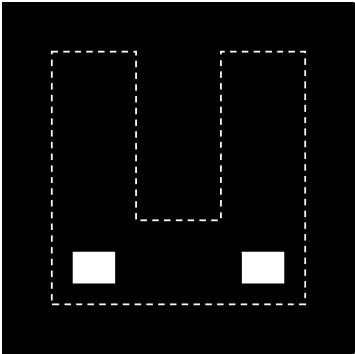


Fig1059_c_NegADI.png

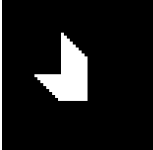
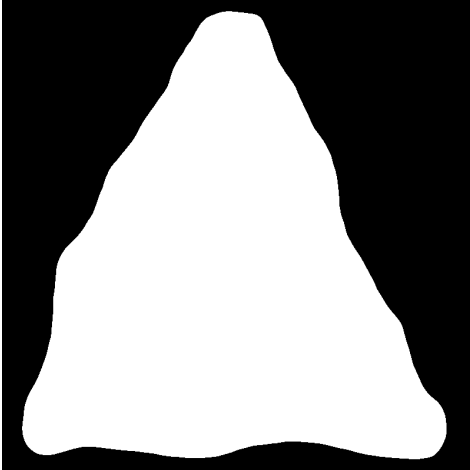
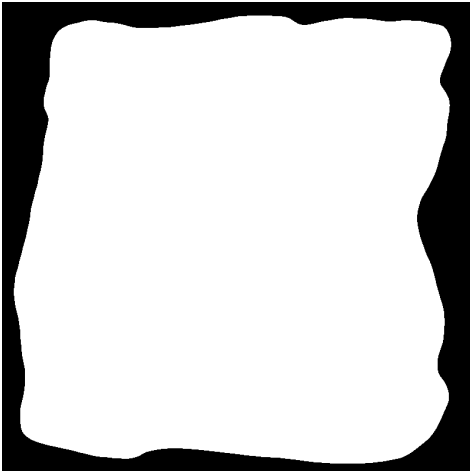
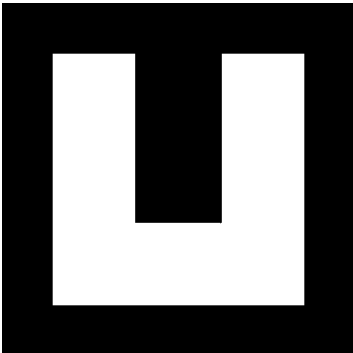


Fig1111_a_triangle.png**Fig1111_b_square.png****FigP0905_top.png****FigP1110.png**

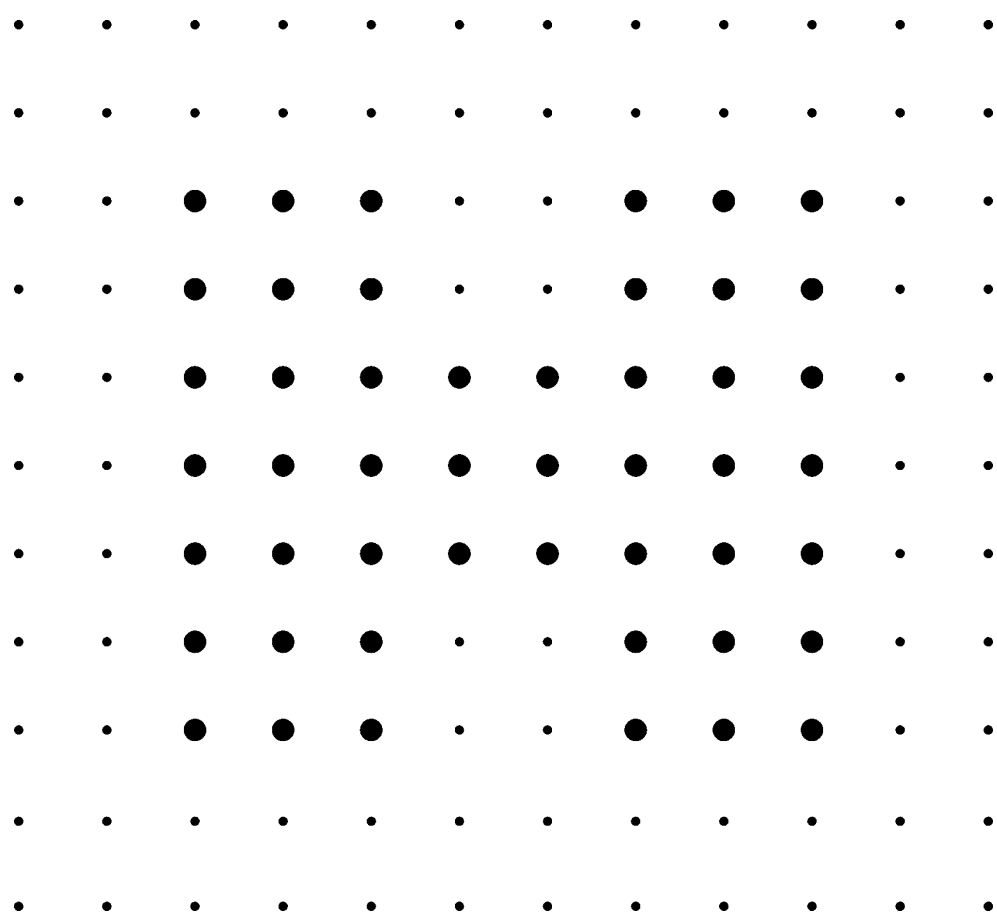
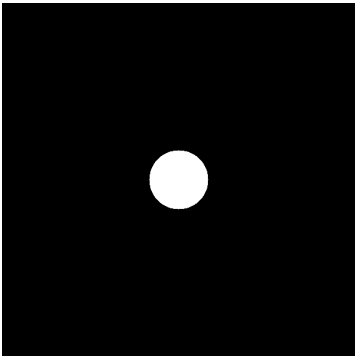


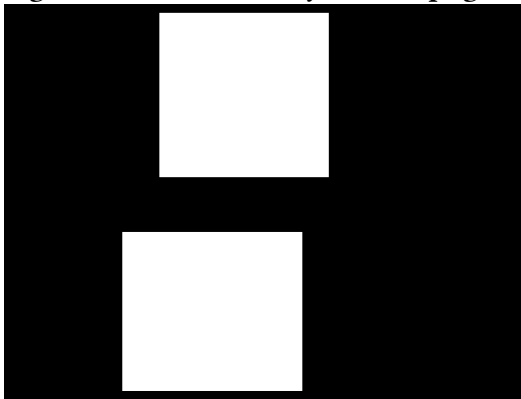
Fig0533_a__circle_.png



FigP0917_noisy_rectangle_.png



Fig0230_b_dental_xray_mask.png



FigP0528_b_two_dots.png

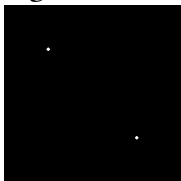


Fig1059_a_AbsADI.png

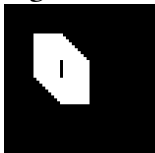


Fig1059_b_PosADI.png

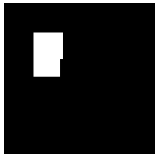


Fig0539_c__shepp-logan_phantom_.png



FigP0905_c_.png

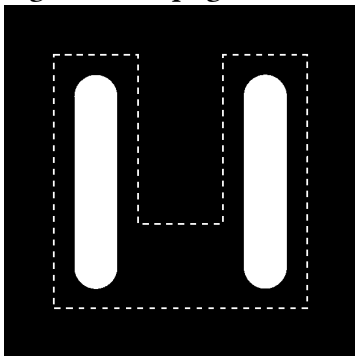
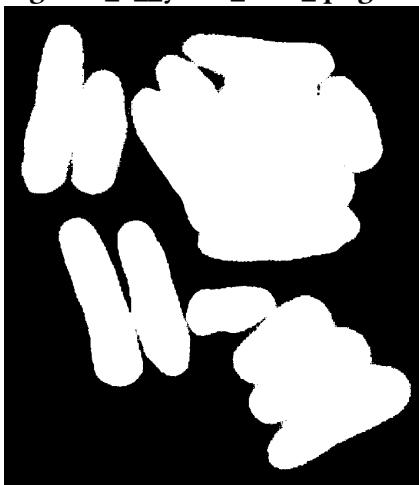


Fig1043_a__yeast_USC_.png



FigP0905_U_.png

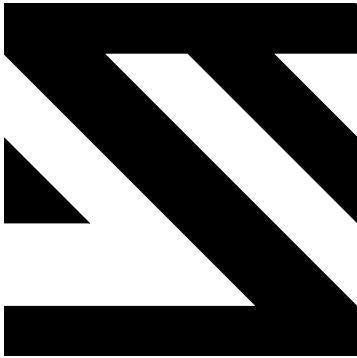


Fig0524_b__blurred-impulse_.png

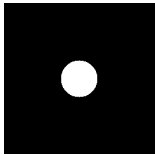


Fig0424_a__rectangle_.png

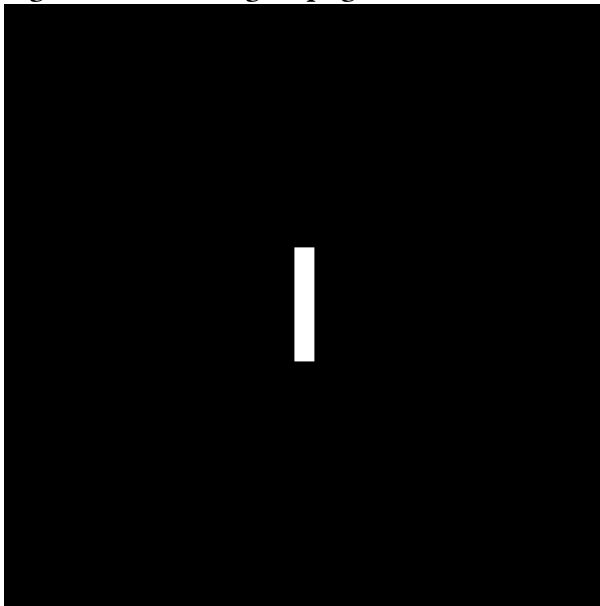
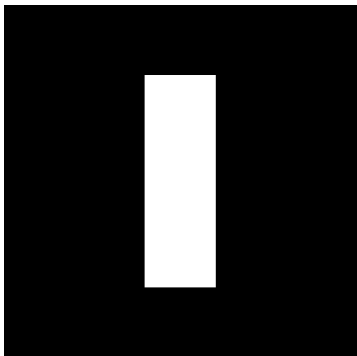


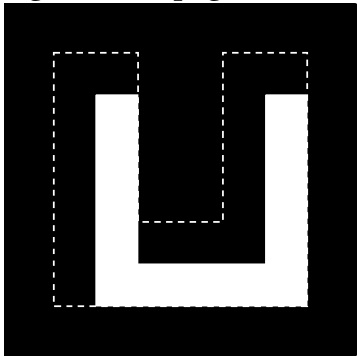
Fig1008_c__roof_edge_.png



Fig0539_a__vertical_rectangle_.png



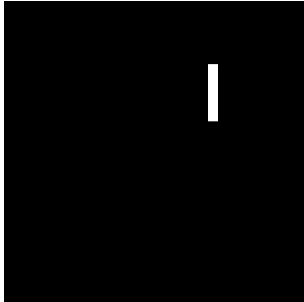
FigP0905_a.png



FigP0433_a.png



Fig0.15_a_translated_rectangle.png



FigP0918_c.png

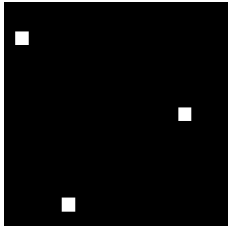


Fig0524_a_impulse.png

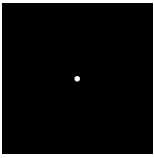


Fig0236_a__letter_T_.png

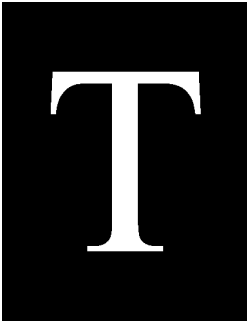
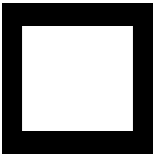


Fig0503__original_pattern_.png



FigP0501.png

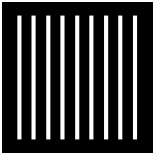


Fig1218_airplanes_.png

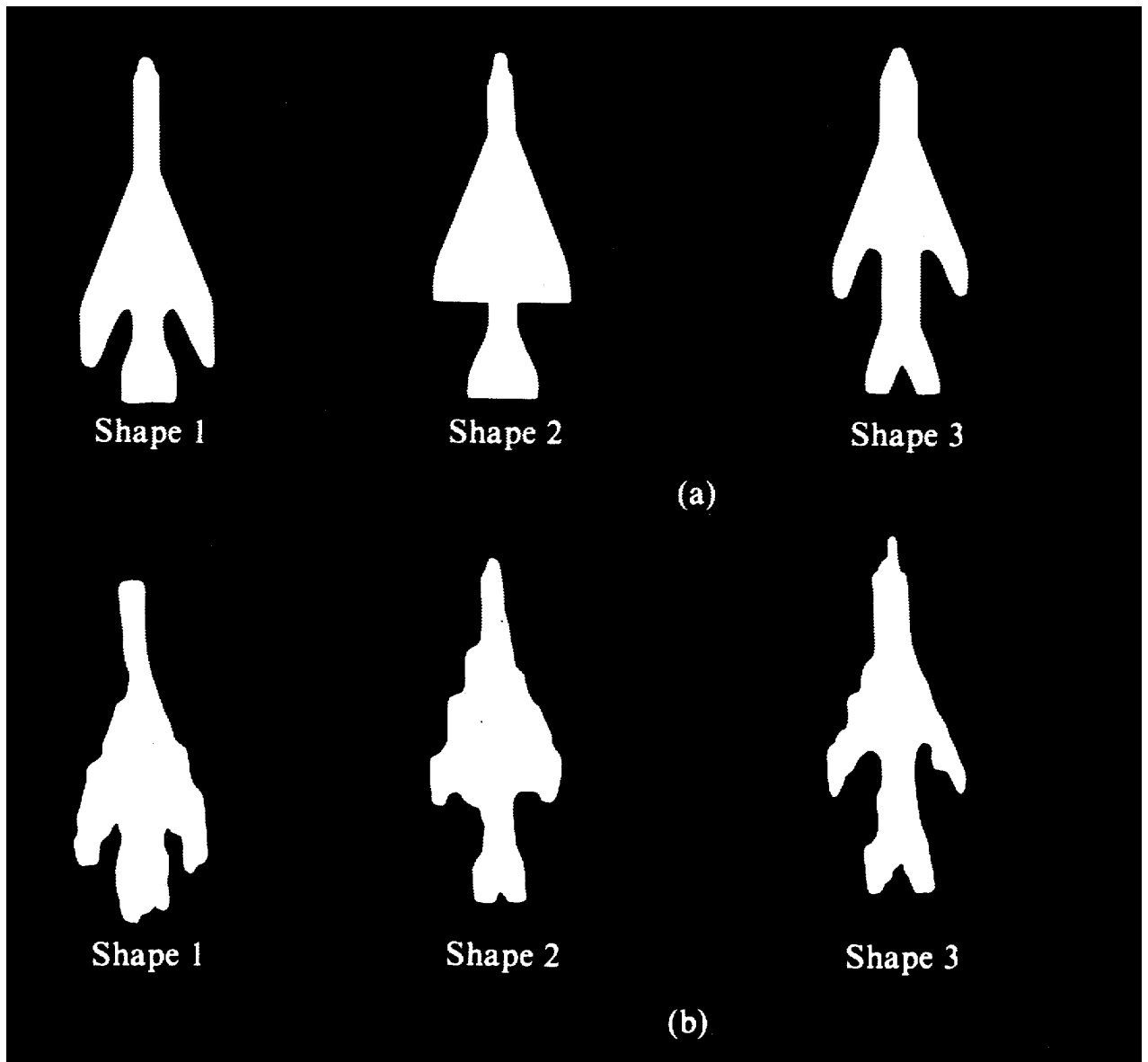
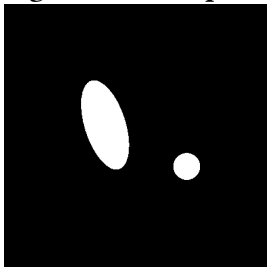
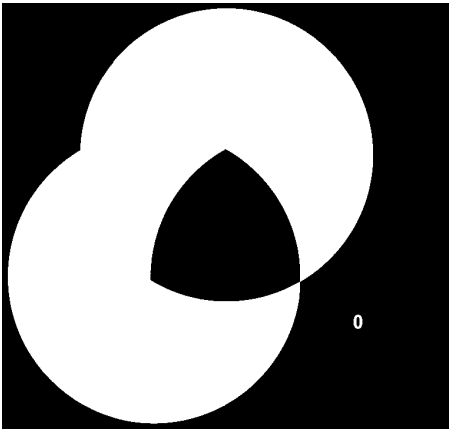
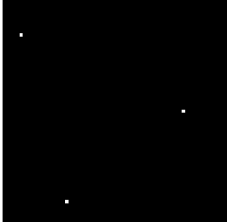


Fig0534_a__ellipse_and_circle_.png

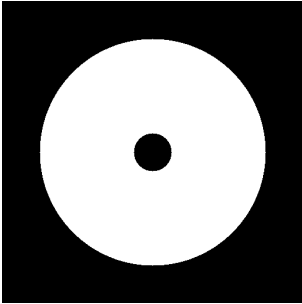




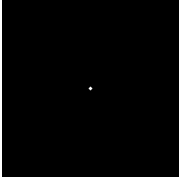
FigP0918_b_.png



FigP0528_c_.png



FigP0528_a__single_dot_.png



REFERENCES

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