The Sparse Abstract Machine

Olivia Hsu
Stanford University
USA
owhsu@stanford.edu

Maxwell Strange
Stanford University
USA
mstrange@stanford.edu

Ritvik Sharma
Stanford University
USA
rsharma3@stanford.edu

Jaeyeon Won
MIT
USA
jaeyeon@mit.edu

Kunle Olukotun
Stanford University
USA
kunle@stanford.edu

Joel S. Emer
MIT and NVIDIA
USA
jsemer@mit.edu

Mark A. Horowitz
Stanford University
USA
horowitz@ee.stanford.edu

Fredrik Kjølstad
Stanford University
USA
kjolstad@stanford.edu

ABSTRACT

We propose the Sparse Abstract Machine (SAM), an abstract machine model for targeting sparse tensor algebra to reconfigurable and fixed-function spatial dataflow accelerators. SAM defines a streaming dataflow abstraction with sparse primitives that encompass a large space of scheduled tensor algebra expressions. SAM dataflow graphs naturally separate tensor formats from algorithms and are expressive enough to incorporate arbitrary iteration orderings and many hardware-specific optimizations. We also present Custard, a compiler from a high-level language to SAM that demonstrates SAM's usefulness as an intermediate representation. We automatically bind from SAM to a streaming dataflow simulator. We evaluate the generality and extensibility of SAM, explore the performance space of sparse tensor algebra optimizations using SAM, and show SAM's ability to represent dataflow hardware.

CCS CONCEPTS

• **Computer systems organization** → Data flow architectures; • **Software and its engineering** → Compilers.

KEYWORDS

sparse tensor algebra, domain-specific, streams, abstract machine

ACM Reference Format:
Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3582016.3582051

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582051

1 INTRODUCTION

Specialized streaming dataflow accelerators that leverage pipelining, locality, and parallelism are becoming increasingly popular as performance- and energy-efficient alternatives to CPUs and GPUs. But the efficiency comes at the cost of programmability: all have limits to their application domain, and most have limited and/or difficult programming interfaces. As a result, users often access these accelerators through library calls that are created by expert programmers [56, 57]. Recent research has proposed closing this flexibility and programmability gap by creating reconfigurable dataflow architectures or coarse-grained reconfigurable arrays [7, 13, 22, 36, 37, 39, 42], including compilation tools to map a class of user applications to these arrays [29, 33, 43, 59, 65].

Given these trends, it is not surprising that interest in general accelerators for sparse tensor algebra is increasing [13, 22, 47]. Sparse tensor algebra has applications across many fields including science, engineering, data and graph analytics, and machine learning [1, 6, 16, 25, 30, 40]. Tensor algebra generalizes linear algebra to higher-order tensors, and "sparse" indicates tensor algebra computations where one or more tensors are stored in compressed data structures that omit zeros. Sparse tensor algebra, expressed as a language using tensor index notation or Einstein summation (Einsum) notation, is an important language with a long history, starting as a mathematical notation [46]. It has recently gained traction as a computational language [3] that subsumes linear algebra. To accelerate these computations, many papers have also been published on point solutions for single-expression hardware, where the expression is often sparse matrix multiplication [38, 44, 52, 53, 64, 66].

Most sparse tensor algebra accelerators are fixed-function matrix multiply engines. Arbitrary sparse tensor contractions must therefore be reduced to sparse matrix multiplications through algebraic *factorization* [50, 51]. Factorization breaks up large expressions using transpositions, tensor-to-matrix conversions, and temporaries. However, compared to dense tensor algebra, factorization is significantly more expensive for sparse tensor algebra. In fact, a sequence of matrix multiplications is often more than an order of magnitude slower compared to bespoke generated tensor contractions. And, more importantly, the lack of fusion in sparse computations can, and often does, lead to inferior worst-case asymptotic complexity: the runtime of an unfused expression may grow with the number of tensor components while a fused expression grows with the number of nonzero components [28]. To address the limitations of fixed-function accelerators, the SPU [13], ExTensor [22], and Capstan [47] systems propose programmable sparse dataflow hardware. However, they lack full support for sparse tensor algebra.

We define an abstract machine model for sparse tensor algebra called the Sparse Abstract Machine (SAM) that accelerates general sparse tensor algebra. SAM consists of abstract dataflow blocks that lend themselves to VLSI implementations and compose to implement any sparse tensor algebra expression and to implement many algorithms for each expression, including fused algorithms, unfused algorithms with temporaries, tiled algorithms, and parallelized and vectorized algorithms. Thus, SAM can simultaneously be used to analyze point solutions, be the abstract architecture of a programmable sparse tensor algebra dataflow implementation, and be the intermediate representation of its compiler.

We built SAM to be for sparse tensor dataflow accelerators what LLVM [32] is for instruction-based conventional processors: It defines the machine functionality and provides an interface between the compiler and hardware, allowing the end-to-end system to continue to function while both sides are independently optimized. SAM also enumerates the primitives that are needed to support all features of sparse tensor algebra. Our contributions are:

(1) the first abstract machine model that expresses the whole of sparse tensor algebra computations as spatial dataflow graphs on multidimensional sparse and dense tensors,
(2) cleanly defined dataflow primitives for each of the fundamental features of sparse tensor algebra,
(3) a representation of multidimensional sparse/dense tensors as flattened streams with hierarchical control tokens, and
(4) a compilation strategy from a high-level tensor index notation to our abstract machine model.

To evaluate our contributions we implemented SAM as a cycle-approximate simulator that is generated by our compiler, Custard. Using the simulations, we search the space of sparse tensor algebra architectural designs. Finally, we show that SAM can represent prior sparse dataflow accelerators.

2 BACKGROUND

This section describes the necessary features of a general sparse tensor algebra computing system. We discuss how these features can be programmed with the input APIs of the TACO compiler [27]. We describe why TACO only targets von Neumann machines and propose a new compiler in Section 5 that uses SAM to target dataflow accelerators. Finally, we discuss limitations of prior work on fixed-function and programmable sparse tensor algebra hardware.

2.1 The Design Space of Sparse Tensor Algebra

Tensor algebra computations are typically expressed using tensor index notation (or Einsum notation), where tensors are indexed by index variables, are multiplied and added, and where results may be summed over index variables. For example, matrix multiplication can be written in tensor index notation as $X_{ij} = \sum_k B_{ik}C_{kj}$, where index variables i, j, and k range over the rows and columns that they index. Expressions may have more than two operands, such as sampled dense-dense matrix multiplication (SDDMM) $X_{ij} = \sum_k B_{ij}C_{ik}D_{jk}$. For such compound expressions, it is often beneficial to fuse the resulting computation (i.e., loop fusion or hardware pipelining to avoid materializing large temporary data structures). Tensor index notation only specifies the expression (or algorithm of computation) and does not does not include a description of the schedule (e.g. dataflow traversal order, tiling methodology, and parallelization). Prior work popularizes the separation of algorithm and schedule in both software [8, 45, 48, 55] and hardware [9].

Tensor index notation consists of five features that must be supported by any general tensor algebra computing system:

(1) a way to traverse multidimensional tensors;
(2) a way to combine traversal over multiple tensors;
(3) a way to repeat operands over other operands, e.g., in $x_i = \sum_j B_{ij}c_j$, c must be multiplied by each row of B;
(4) a way to compute scalar additions and multiplications, including summation reductions; and
(5) a way to assign results to a tensor.

Efficient sparse tensor algebra computing systems must also support

(6) compressed data structures for sparse tensors,
(7) index variable iteration in any order, and
(8) fusion of the computation

in order to avoid inferior worst-case asymptotic complexity [2, 28].

2.2 The TACO Compiler

The TACO compiler [27] and related systems [5, 35, 61–63] compile tensor index notation to von Neumann architectures, including CPUs, GPUs [48], and distributed machines [61, 62]. TACO supports all five features of tensor index notation, as well as compressed data structures [11, 27], iteration reordering [26], and fusion [27]. The TACO compiler has three separate languages that independently describe functionality, data, and optimization: tensor index notation, a data representation language, and a scheduling language.

Although TACO has the generality we described in Section 2.1, it only compiles to von Neumann architectures. This limitation is due to its lowering machinery fundamentally embedding the (co-)iteration over one or more tensors into general-purpose control flow found in von Neumann machines including: indirect index accesses, `while` loops, and `if` statements. The heavy reliance on these constructs during lowering and code generation for traversing irregular structures makes it inapplicable for reconfigurable dataflow accelerators since many of these architectures remove general control flow to achieve higher performance [7, 13, 22, 39, 42, 47]. Converting the complex control flow generated by TACO into a dataflow abstraction, such as the one we describe in this paper, would require significant assumptions about data accesses and/or a complex synthesis system. Instead, we describe an alternative compiler lowering algorithm that lowers from TACO's high-level concrete index notation [26] to the SAM dataflow abstraction.

2.3 Prior Work on Fixed-Function Hardware

There is a large body of recent work on architectures that explore point solutions in the space of hardened sparse tensor algebra expressions and algorithms [10, 19, 21, 38, 40, 44, 52, 53, 64], which we will call fixed-function accelerators. Many of these implement sparse matrix multiplication (SpM*SpM) [19, 21, 38, 44, 52, 64]. For example, SIGMA [44] implements the inner-product $i \rightarrow j \rightarrow k$ iteration order. Although this order is typically preferred for dense matrix multiplications, and although it avoids scattering into the result, it has poor asymptotic performance because it iterates over all combinations of i and j before coordinates are intersected at the contracted variable k. GAMMA [64] applies Gustavson's [18] $i \rightarrow k \rightarrow j$ iteration order, which improves the asymptotic complexity at the cost of merge hardware to rearrange the reduction step to allow in-order generation of elements of X. And OuterSPACE [38] implements the outer-product $k \rightarrow i \rightarrow j$ iteration order, which for doubly-compressed sparse row (DCSR) matrices has better asymptotic complexity than the $i \rightarrow k \rightarrow j$ order but requires an additional step for merging whole matrices into X.

Using fixed-function matrix multiplication hardware to compute any expression in tensor algebra relies on factorizing general expressions into a sequence of invocations of matrix multiplication operations. Factorizing unfuses the computation and fixes the dataflow ordering. SAM removes this limitation by having sufficient power to express fused expressions on accelerators, letting us explore both the benefits and the costs of these approaches.

2.4 Prior Work on Programmable Hardware

The limitations of fixed-function dataflow hardware motivate programmable sparse tensor algebra dataflow hardware, which are implementations of an abstract machine like the one in this paper. We describe three major lines of prior work on programmable sparse tensor algebra dataflow hardware: the SPU [13], ExTensor [22], and Capstan [47]. While these designs do not support the full generality described in Section 2.1, they suggest the essential hardware structures and techniques in dataflow accelerators for sparse tensor algebra and greatly influenced our work.

SPU. The Sparse Processing Unit (SPU) [13, 37] is a spatial streaming dataflow architecture where instructions on a CPU configure a Coarse-Grained Reconfigurable Architecture (CGRA) and then stream arrays to it. It has efficient hardware both for combining streams (e.g., an intersection) and for using one stream to index into an on-chip array [13]. The SPU can be used to implement binary vector operations, relational joins, and graph algorithms [12], thus unifying domains. The SPU also includes an idiom-directed compiler [59] from pragma-annotated C loops to CGRA configurations.

Although the SPU literature describes operations that support a broad set of domains—tensor algebra, relational algebra, and graph operations—the SPU CGRA [13] only supports vector operations, while higher-order tensor algebra operations appear to be implemented as CPU loops that dispatch inner-loop vector operations to the CGRA (see Figure 6 from Nowatzki et al. [37], which the SPU extends). Thus, higher-order expressions must be broken into pieces with data flowing between the CPU and CGRA engine, reducing pipeline locality. The abstract machine we describe in this paper provides a complete dataflow model for tensor algebra. Using it, together with the compiler, to target the SPU and the TACO compiler to target the CPU, provides a fruitful path towards compiling to the SPU system from a higher-level tensor algebra language.

ExTensor. The ExTensor system [22] is a CGRA-style architecture designed to hierarchically evaluate Einsum operations on sparse tensors. In ExTensor, a compute element is made up of memory that stores pieces of tensors, i.e, fibers, and hardware to perform operations on those fibers (e.g., read/stream them in/out, perform intersections/MACCs). ExTensor's design primarily considered a topology of compute elements that each operated on two fibers (from two operand tensors) at a time, and was focused on computing SpM*SpM. To support SDDMM, which has three operands, an ExTensor instance with additional compute elements was proposed. However, it cannot perform all Einsum computations, such as those with union merges for addition. Extensor also did not provide a programmatic approach to map an arbitrary Einsum to a concrete ExTensor instance, which makes it hard to add new expressions that were not tested by the authors. Nor is there a discussion of how the architecture might change to better suit the needs of arbitrary Einsums. SAM and our compiler addresses these limitations, and should allow for the creation of ExTensor configurations.

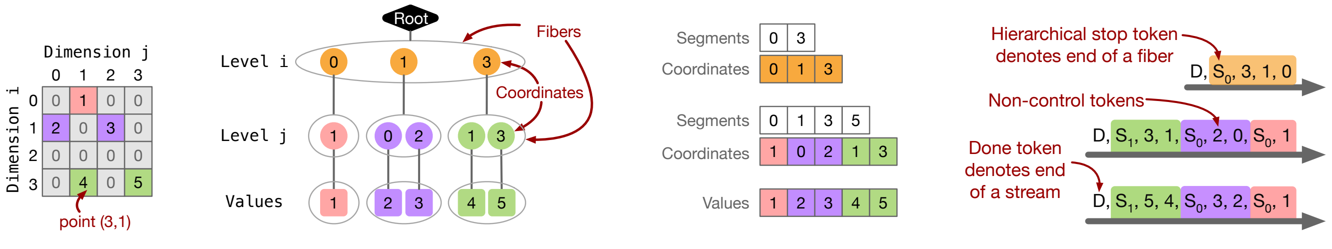
Capstan. Finally, the Capstan system [47] supports hierarchical iteration over dense, compressed, bitvector, and bit-tree data structures. For some data structure types, it supports any tensor algebra expression, including additions and multiplications. Its primary limitation is that it does not support combining two or more compressed data structures (e.g., by intersection) at one iteration level. Instead, it relies on bitvectors and bit-trees, which densify the iteration. The paper argues that this works well for common clustered sparse tensors, but not for arbitrary sparsity. Finally, the Capstan system does not support efficient broadcasting of a tensor over a sparse inner dimension of another tensor, as it relies on programmed counters to control broadcasting. While Capstan is programmed by Spatial [29, 65], a domain specific language for hardware accelerators based on parallel patterns [41], Spatial is at a lower-level of abstraction—more descriptive of the hardware accelerator—than the Custard input APIs. Again SAM provides an opportunity to provide a higher-level programming interface for this system.

3 THE CORE SPARSE ABSTRACT MACHINE

SAM provides a clean method to transport tensors on wires and to express all tensor algebra operations, serving as an LLVM-like interface. We define nine types of dataflow blocks that can be composed to execute arbitrary sparse tensor algebra expressions. Level scanners fetch a tensor's nonzero coordinates and send them as streams to intersecters, unioners, and repeaters that combine coordinates from different tensors. ALUs and reducers compute tensor operations. Coordinate droppers filter out unnecessary coordinates, and level writers write the resulting sparse tensor to arrays in memory.

SAM lets programs use as many blocks as needed. Of course, any physical implementation is constrained to a finite set of resources. Our compiler can be used to transform an unconstrained graph to a specific physical backend by breaking up the computation in time through data movement into temporary memories and block reuse.

SAM is sufficiently expressive to represent any dataflow for any tensor algebra expression, as it implements all the features in Section 2.1. Furthermore, SAM implements a streaming model of those features, and Kovach and Kjolstad have shown an equivalence proof of sparse tensor algebra and a formal streaming model [31].



(a) Matrix B_{ij} (b) B_{ij} as a fibertree (c) B_{ij} stored in memory (d) B_{ij} sent through a stream

Figure 1: The data model of the SAM models sparse tensors (Figure 1a) as a coordinate fibertree, (Figure 1b) that can be stored in memory (Figure 1c) as DCSR data structure, or sent through streams (Figure 1d) where time increases from right to left.

3.1 Tensor Data Model

In the SAM abstract data model, each tensor is a coordinate tree where each tree level represents the coordinates of a different tensor dimension. This coordinate tree abstraction was first introduced as part of the TACO system [11, 27] and further abstracted and formalized as *fibertrees* [54, 60]. Fibertrees are tries where each coordinate at one level is linked to a *fiber*—a list of child coordinates—at the next level. Crucially, only those children whose sub-trees have nonzeros are stored. Figure 1a shows a sparse matrix and Figure 1b its corresponding fibertree. The matrix is stored in row-major order, so the i coordinates (orange circles) are stored at the top fibertree level. The i coordinate 2 is not stored since its sub-tree (the third row) has only zeros. The middle level stores a j coordinate for every nonzero component and the last level stores nonzero tensor values. Fibertrees are useful for reasoning about tensors level by level without considering the exact storage representation.

Fibertrees are stored in memory and transmitted via streams. When in memory, each tree level is separately assigned a storage type that specifies its data representation. A level's data representation can be an uncompressed level that stores a single number encoding the fiber size or it may be a compressed data structure that stores only coordinates with nonempty sub-trees. Many other data representations are possible with this abstraction [11, 54, 60]. Figure 1c depicts one possible in-memory data structures for the fibertree in Figure 1b, where both levels are stored in compressed data structures. This storage format is called doubly-compressed sparse rows (DCSR), where a segment array denotes the start and stop reference positions of each segment in the coordinate array. A segment is one way to encode fiber data associated with an array representation. Concretely in Figure 1c, the level j segment [3, 5) refers to the green level j coordinates [1, 3] since the coordinates are located at indices [3, 4] in the level j coordinate array.

3.2 Tensor Streams

SAM streams are abstractions of physical wires that connect processing blocks and transmit data between these blocks. Each SAM stream is a sequence of tokens that transmits one level of fibertree data, along with stop tokens (S_n) denoting the hierarchical fiber boundaries within a level, and a done token (D) to mark the end of a stream. There are three types of SAM streams: coordinate streams (abbreviated as `crd`) that transmit coordinate levels, value streams (`vals`) that transmit last-level tensor values, and reference streams (`ref`) that transmit references to the location of each coordinate's child fiber in memory. Figure 1d shows the fibertree in Figure 1b as coordinate and value streams. Streams can be interpreted as variable-length nested lists where each stop token represents a parenthesis. Thus, the value stream in Figure 1d,

$$\overleftarrow{1,\ S_0,\ 2,\ 3,\ S_0,\ 4,\ \ 5,\ S_1,\ D}$$

represents the nested value level

$$((1), (2, 3), (4, 5)).$$

The data closest to the arrowhead is sent first, and the done (D) token terminates the stream.

3.3 Tensor Iteration

SAM sparse dataflow algorithms start with level scanners that load tensors from memory and turn them into streams.

Definition 3.1 (Level Scanner). A level scanner takes in a reference stream and outputs two streams: one of coordinates and one of references. It produces a single fibertree level on its output coordinate stream, fiber by fiber. Each non-control token on the input stream is a reference to a single fiber location for a given level in memory. The level scanner generates all coordinates in that fiber, along with their corresponding references, and then adds an additional stop token to denote the end of the fiber.

Each SAM level scanner generates fibers for only one dimension. Therefore, multiple scanners must be composed to iterate over a multidimensional tensor. The composition uses the references emitted from each successive level scanner to locate the fibers of the next level scanner. The key to this composition is that level scanners

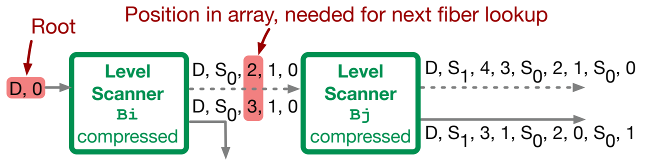

Root
Position in array, needed for next fiber lookup
D, 0
Level Scanner Bi compressed
D, S0, 2, 1, 0
D, S0, 3, 1, 0
Level Scanner Bj compressed
D, S1, 4, 3, S0, 2, 1, S0, 0
D, S1, 3, 1, S0, 2, 0, S0, 1

Figure 2: Composition of level scanner blocks.

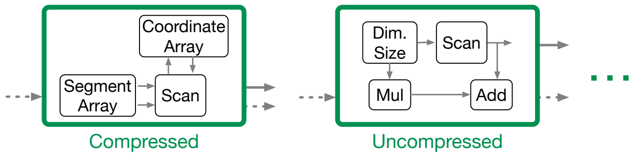


Figure 3: Implementations of the level scanner interface.

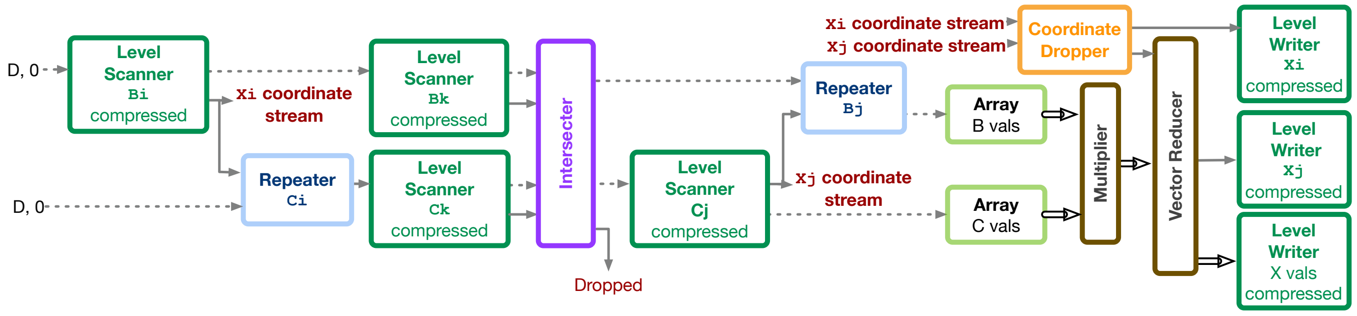


Figure 4: The SAM dataflow graph for sparse matrix multiplication $X_{ij} = \sum_k B_{ik} C_{kj}$, on DCSR matrices with linear combination of rows ($i \rightarrow k \rightarrow j$ order). Stipled, solid, and double arrows resemble reference, coordinate, and value streams respectively.

communicate information by embedding both fiber location and coordinate hierarchy—needed by downstream level scanners—into the reference streams. Each level scanner adds a level to the hierarchy by either adding an S_0 stop token at the end of each scan or by incrementing all input stop tokens by one. They thus chain together to load an entire tensor and to convert it to per-level streams. Figure 2 shows two level scanners that iterate over the compressed matrix in Figure 1c. The reference stream emitted by the final-level scanner is sent to blocks that load values from memory, as described in Section 3.6. Each level scanner also connects to a memory array (Definition 3.5) that stores the fiber and coordinate information for the level, but these are not shown in figures to reduce clutter.

The SAM level scanners support iterating over tensors stored in various in-memory level formats presented in [11], which decouples an algorithm from the tensor formats. Thus, the interfaces of the level scanner are format agnostic and Figure 3 demonstrates how they remain unchanged as the level format implementation varies.

3.4 Illustrative Example

We will use the linear combination of rows algorithm (sometimes referred to as Gustavson's algorithm [18]) for sparse-matrix sparse-matrix multiplication (SpM*SpM) to illustrate the operation of SAM blocks, and to demonstrate how their composition defines different algorithms. The Einstein summation notation for this algorithm is $X_{ij} = \sum_k B_{ik} * C_{kj}$, where the matrix multiplication is accomplished by using an index order of $i \rightarrow k \rightarrow j$ [64]. The advantage of this iteration order is that k coordinates are first intersected and only those ks that survive result in further computation.

Figure 4 shows the algorithm as a SAM dataflow graph. From the left, the coordinates of the two matrices are loaded from DCSR data structures in memory by level scanners. The coordinates are then transformed into a three-dimensional iteration space by chaining together the $i \rightarrow k$ coordinates of the B matrix with the $k \rightarrow j$ coordinates of the C matrix. This space requires duplicating data to fill in missing dimensions. In this example each matrix is broadcast over an index variable of the other matrix (B over j and C over i).

3.5 Stream Merging

Once the operand coordinate streams have been generated, the next task is to merge them. The index variables of a tensor index notation expression create an iteration space that we must cover, taking advantage of both the sparsity of the tensors and the mathematical properties of the operations to avoid unnecessary computation. Our design covers this sparse iteration space hierarchically by merging the coordinates of one dimension at a time, with the surviving coordinates from one dimension dictating what fibertree fibers need to be merged in the next dimension. The hierarchical merging is implemented with per-level merging blocks (intersection and union) and repetition machinery to handle the case where a tensor is broadcast [20, 24] across the dimension of another tensor, as required by our illustrative example in Figure 4.

The merging operations combine m streams, representing the same coordinate level of all operand tensors, fiber by fiber. Coordinate merging is inherently a set operation: specifically, intersection (since $a \cdot 0 = 0$) and union (since $a + 0 = a$) suffice for tensor algebra.

Definition 3.2 (Intersecter). An intersecter has m pairs of coordinate and reference streams go in and one coordinate stream and m reference streams come out. It outputs coordinates and corresponding input references when all input coordinates are equivalent.

Definition 3.3 (Unioner). A unioner has the same input/output interface as the intersecter. However, it outputs coordinates and their associated input references whenever there exists at least one coordinate from any input. If the coordinate exists only on p inputs where $p < m$, the union block outputs an empty (N) token on the the other $m - p$ output reference streams.

Figure 5 shows an example of a binary unioner that produces a coordinate stream that is the union of two input streams, along with the references from each input reference stream whose coordinates survived the union. Both emitted reference streams are augmented with empty tokens (N) to have the same shape as the emitted coordinate stream.

As we saw in Figure 4, it is common for expressions to replicate one tensor across a dimension of another, often called array broadcasting. Figure 6 shows a simple vector scaling example. It demonstrates how the repeater block replicates a reference stream over

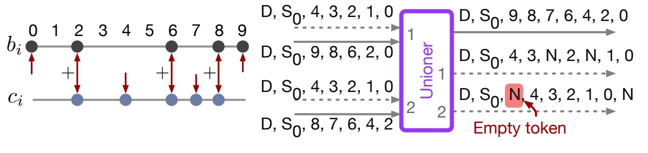


Figure 5: Example of union coiteration for $b_i + c_i$

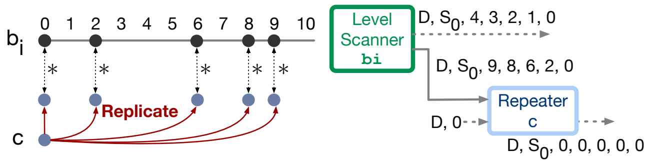


Figure 6: Repeating a scalar with a repeater block.

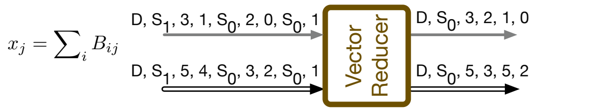


Figure 7: Example using the row reducer, where $n = 1$, to accumulate the columns of the matrix from Figure 1a.

every coordinate of the provided coordinate stream. The repeater is a new primitive that solves limitations on prior work architectures needing to pre-configure higher-order iteration counts [13, 22, 47].

Definition 3.4 (Repeater). Repeaters have one input coordinate stream, one input reference stream, and one output reference stream. Each non-control token in the input reference stream is repeated m number of times, where m is the number of non-control tokens from the input coordinate stream before a stop token is seen.

Hierarchical repeating and stream merging compose to express algorithms for multidimensional tensor contractions. Reconsider the linear combination of rows SpM*SpM algorithm from Figure 4. The i coordinates loaded from B are not only passed to the i level writer of X by way of the coordinate dropper, but also fed to a repeater that broadcasts all of C's k coordinates over each i.

3.6 Computation

After stream merging, the remaining coordinates are coordinate space points that contribute to the result. Their corresponding reference streams are passed to array blocks that load their values.

Definition 3.5 (Array). An array block is a proxy for a memory interface and can be treated as a contiguous section of memory. It has two interface modes—load, which given one input reference stream fetches data to produce one output stream of any type, and store, which given one input reference stream and one input data stream of any type has a side effect that stores the data to its corresponding reference location in memory.

In SAM, arrays store values, coordinates, and references. In the computation pipeline, value streams are read from the array of each operand, with the same coordinates, and combined using streaming arithmetic-logic units (ALUs). The Figure 4 ALU is a multiply unit.

Definition 3.6 (ALU). An ALU block consumes two value streams and produces one value stream. It applies an arithmetic operator (add, subtract, or multiply) to inputs, treating empty tokens as zeros.

In addition to combining values at the same coordinate, often the algorithm needs to accumulate a tensor. In our illustrative example in Figure 4 this occurs at the end, where we sum over the k dimension (multiple dimensions can be summed by chaining reduction blocks). Reductions in tensor algebra may occur over any tensor dimension, independent of the order in which we choose to merge coordinates. Thus, summation reductions may occur over the coordinate level that is merged last (requiring a scalar to accumulate the result), over the coordinate-level merged second to last (requiring a vector to accumulate the results), or over coordinates merged earlier (requiring a higher-dimensional tensor to accumulate the results). SAM provides one block for reductions that must be configured for any specific dimension of accumulation.

Definition 3.7 (Reducer). A reducer is configured by n, the dimension of the memory needed in the reduction. It inputs and outputs n coordinate streams and one value stream. The block is sent an entire n-dimensional (sub-)tensor with repeated points/values and outputs streams that represent that tensor with unique coordinates and summed values. Specific reducers include: scalar where $n = 0$, vector where $n = 1$, and matrix where $n = 2$.

The reducer internally adds values corresponding to equivalent coordinate points and stores the results in an internal storage, which may be a dense or a sparse data structure. Finally, when an n-level reduction is completed, for example when a whole row has been processed for the Gustavson's algorithm in Figure 4, the reducer emits the resulting tensor as streams with deduplicated coordinates. When accumulating coordinates with empty fibers, resulting from ineffectual intersections, the reducer may be configured to either accumulate empty fibers into an explicit zero (the identity for addition) or to remove the empty fibers by removing their extra stop tokens. The choice is an implementation decision, but empty reduction behavior may affect SAM graph construction for other blocks (see Definition 3.9 and Table 2).

Like with level scanners, various implementations of the reducer are possible underneath the abstraction, including k-way merging, dense arrays, compressed data structures, and bitmaps [34, 42, 47]. Figure 7 shows an example of a row ($n = 1$) reducer.

3.7 Tensor Construction

The final step of a SAM graph is to store the resulting tensor streams back to memory. Specifically, the surviving coordinate streams for the index variables used to index the left-hand side of the Einsum expression, as well as the computed values, need to be stored back into per-level tensor memory representations.

Definition 3.8 (Level writer). Level writers take in either one value stream or one coordinate stream and store its contents to memory, internally generating reference information and auxiliary level data structures. As a result, the block is a wrapper around the store mode of a coordinate Array (and its metadata) or a value array. The level writer's internally generated references store the data tokens from the input stream in order.

In cases with at least one index-variable level above an intersection level, the result coordinate streams must be cleaned before the level writer stores it back to memory. The coordinate cleanup removes any outer-level result coordinates that have ineffectual inner-level intersections (either empty intersections or zero values) as shown in Figure 8. We introduce the coordinate dropper block to handle these cases. Coordinate droppers with value stream inputs are optional if explicit zeros need not be removed. In this case, other coordinate dropper blocks are also optional if the reducer

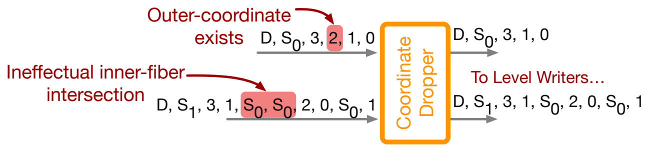

Outer-coordinate exists
D, S0, 3, 2, 1, 0
Ineffectual inner-fiber intersection
D, S1, 3, 1, S0, S0, 2, 0, S0, 1
Coordinate Dropper
D, S0, 3, 1, 0
To Level Writers...
D, S1, 3, 1, S0, 2, 0, S0, 1

Figure 8: Dropping coordinate 2 from the matrix in Figure 1a.

blocks is configured to accumulate empty fibers into 0-values, since ineffectual (empty) intersections will produce explicit values.

Definition 3.9 (Coordinate Dropper). The coordinate dropper takes in one outer-level coordinate stream and one inner-level coordinate or value stream. It removes both the outer-level and inner-level tokens that came from ineffectual merging or computation (empty fibers or zeros) at the inner level.

3.8 Alternatives and Tradeoffs Discussion

We discuss alternatives and tradeoffs of core SAM design decisions, which we hope will provide insights into our design rationale.

Stream Control Tokens. Section 3.1 presents a solution for streaming dataflow where the control tokens (stop, empty, and done) are directly passed through the data plane, but alternative solutions to control flow for dataflow exist. Other valid dataflow control-flow solutions include control signaling on dedicated control-only streams, embedding control (via counters and control tokens) directly into each primitive, and having a completely separate control plane. We give examples, discuss the limitations of these approaches, and justify putting control tokens on the data plane.

In initial iterations of SAM, we considered representations with primitives that had dedicated control signaling using control-only streams. For example, we considered passing repeat information by connecting two level scanners together to exchange signals that denote when to stop repeating coordinates. Having streams that only communicate control information risks underutilization, where no information is passed through most cycles, and complicates the composition of multiple blocks, as they would need to be hardened together. However, the benefit of this design would be that control information (when produced) and data can be processed in parallel.

We also considered a SAM design that embedded control directly into each block, which included embedding repeat counters into level scanners and done signaling into each primitive. The direct embedding of counters and other control logic into the primitives increases the primitive area and hardware complexity. Additionally, pre-configuration of counters is usually necessary, meaning that metadata information (like number of nonempty elements) must be obtained by the compiler statically during compile time by iterating over the sparse data at least once on the CPU.

Finally, a design with a separate control plane (not just separate control wires) would be more similar to a von Neumann architecture than prior work on dataflow architectures. Dataflow architectures attempt to remove performance overheads of traditional CPUs by eliminating most general-purpose control, which is done by removing the control unit and restricting control. Having a separate control plane on SAM would fundamentally push our design closer to a von Neumann machine abstraction and would thus not be a good fit for representing streaming dataflow accelerator backends.

Although having control tokens directly processed on the data plane may decrease performance—primitives must now process these tokens—they increase interconnect and logic utilization, decrease primitive area, minimize primitive logic complexity, and still allow for streaming dataflow processing (massive pipelining/parallelism with low control overhead).

Level-Based Stream Representation. Another approach to our level-based streaming tensor representation would be a less efficient point-based streaming representation. One implementation could stream flattened tensor point tuples with no control tokens. The tensor from Figure 1 could thus be represented as

$$\overleftarrow{(0,1,1),\ (1,0,2),\ (1,2,3),\ (3,1,4),\ (3,3,5),\ D}$$

In this representation, the number of processed stream tokens for identity matrices is $3 \cdot \text{nnz}_B$, where nnz_B is the number of nonzeros in B. We can compare the two representations to find when the point-based representation has more tokens using the equation $3 \cdot \text{nnz}_B > (1+c) \cdot \text{nnr}_B + 2 \cdot (1+c) \cdot \text{nnz}_B$ where c is the fraction of control tokens and nnr_B is the number of nonempty rows in B. Using worst-case numbers from our analysis in Figure 14, we rewrite the equation to $3 \cdot \text{nnz}_B > 1.3326 \cdot \text{dim}_{B_i} + 2 \cdot 1.3326 \cdot \text{nnz}_B \implies \text{nnz}_B > 3.98 \cdot \text{dim}_{B_i}$ where dim_{B_i} is the number of rows in B. The result demonstrates that our level-based representation, in the worst-case, processes less tokens than the point-based approach when there are on average more than 4 elements per row. Of the matrices we selected in Figure 14, all 5 middle 50 and 5 large 50 matrices satisfy the 4× inequality and are more efficient in our level-based representation. Our approach becomes even more efficient for higher-order tensors. The coordinates at every level are expanded to the last level—proportional to roughly $O(n^N)$ instead of $O(n^2)$ for matrices, where n is a single tensor dimension and N is the tensor order—to produce the tensor point tuples.

4 OPTIMIZATION DISCUSSION

The core SAM blocks introduced in Section 3 are complete in the sense that they compose to express every tensor algebra expression. Moreover, they suffice to express all coordinate processing (dataflow) orders and fusion—the primary tools to construct algorithms with good asymptotic complexity [2]. To express SAM graphs that further optimize performance and deal with finite hardware, we have added additional capabilities. These capabilities let the graphs express parallelism, tiling, and more ways to represent tensor information either in memory or as streams. In this section, we discuss how SAM extends to include these additional optimizations and how they compose with the core SAM from Section 3.

4.1 Tiling

In our data model, tiling a tensor splits a single fibertree level into multiple levels and then reorders those levels to produce smaller subtensors (tiles). Figure 9 shows how SAM can sequence tiled tensors between host and accelerator devices for computation with fixed-size memories. SAM graphs are used in outer levels to sequence the tile coordinates (tile IDs) for reuse and in the inner levels to perform the computation. The tile sequencing is equivalent to tensor iteration (Section 3.3) and stream merging (Section 3.5), where tile IDs are coordinates and the values are references to the next level of

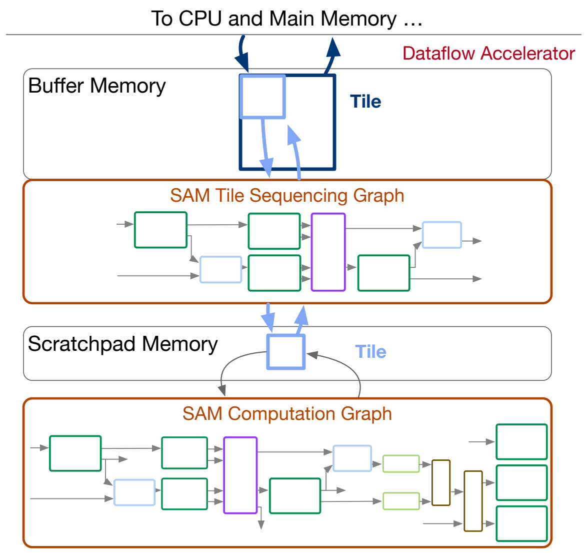


Figure 9: How to sequence tiled tensors (shown in blue) to fit in finite memory for SpM*SpM. The SAM computation graph is the same as in Figure 4, and the SAM tile sequencing graph performs coiteration and merging of tile coordinates.

tiles. As in ExTensor [22] and Capstan [47], we assume that tensors are tiled beforehand so that each tile fits in the dataflow accelerator's memory hierarchy. We demonstrate in Section 6.4 SAM's ability to fit tensors into finite memories and the tradeoff space of different memory configurations (dictated by architectural and implementation-specific memory configurations, like the maximal tile size and bandwidth at each level of the memory hierarchy).

4.2 Tensor Locating

In Section 3, all intersections are performed using coiteration, where the coordinates are intersected using a two-finger merge strategy. This is sufficient for computational correctness, however, it can be asymptotically inefficient. (The core SAM has to coiterate between an uncompressed dense counter level and a compressed level even though it is sufficient to iterate through just the sparse level.) We can often improve intersection efficiency if one tensor has far fewer elements than the other. Rather than waiting for the larger tensor to stream all its level coordinates, it can be more efficient to ask the larger tensor if it contains any of the coordinates from the smaller tensor. This operation, known as iterate-locate or leader-follower intersection, is possible with another SAM block that uses a coordinate instead of a reference to index an array.

Definition 4.1 (Locator). A locator takes in one coordinate and reference stream and outputs one coordinate and two reference streams. For each coordinate, the block finds the associated reference within an array block, if it exists, and outputs that reference and the input coordinate and reference. Otherwise, it emits an empty fiber on all streams.

With locators, we can reorganize SAM graphs to remove intersecters. A prominent example that benefits from this optimization is the inner product sparse matrix-vector multiplication, where the vector is dense. By streaming through the coordinates of each matrix row and locating into the vector, we avoid loading the values of the vector whose corresponding matrix value is zero. Locate blocks can also be used to scatter into a result that supports random insert, such as a dense left-hand-side tensor. Thus, the linear combination of rows matrix-vector multiplication can avoid a vector reducer.

Locators can also speed up intersection when used in conjunction with intersecters that communicate information back to level scanners about coordinate ranges that are no longer needed. This optimization, called coordinate skipping or galloping, is common in software and has also been proposed in hardware [22]. In coordinate skipping, the intersecter sends a signal back to the trailing level scanner (extending the interface of both blocks from the definitions in Section 3), informing it of the coordinate that is needed next. The level scanner, in conjunction with a locator, then skips ahead to this coordinate and avoids sending useless coordinates between its current coordinate and the coordinate sent by the intersecter.

4.3 Bitvectors

Bitvectors are a natural way to compress coordinate information since bits are easy to implement in hardware. Bitvectors have a 1 in positions where explicit coordinates exist and a 0 for empty (or zero) coordinates. Bitvectors have a pseudo-dense iteration space—one that iterates proportional to some constant factor of the dense dimension of each tensor level. This iteration is usually asymptotically worse in performance when compared to compressed iteration, especially with increasing sparsity. However, bitvectors may also increase efficiency since an n-bit bitvector, encoding n coordinate elements, can be processed in one cycle. In some prior-work hardware like Capstan [47] and SIGMA [44], bitvectors are the only compression protocol. Bitvectors may also be offered in addition to compressed coordinates, with blocks that convert between their stream protocols. We introduce a bitvector converter that transforms a coordinate stream into a new bitvector stream protocol and describe a level scanner for the bitvector level format.

Definition 4.2 (Bitvector Converter). Bitvector converters transform b coordinates from the input coordinate stream into a single bitvector token of b bits on the bitvector stream output. Each bit indicates whether it has children or whether its sub-tree is empty.

The SAM bitvector level scanner is similar to Definition 3.1, but it outputs a bitvector stream instead of a coordinate stream. The bitvector level scanner also changes the reference stream behavior presented in Section 3.2. Consider the b vector from Figure 6 that produces the coordinate stream $\overrightarrow{D, S_0, 9, 8, 6, 2, 0}$ compressed as the 4-bit bitvector stream $\overrightarrow{D, S_0, 0011, 0100, 0101}$. As a reminder, the compressed level scanner would output a reference stream of $\overrightarrow{D, S_0, 4, 3, 2, 1, 0}$ to indicate contiguous references (positions) in memory. However, the bitvector level scanner instead produces the reference stream $\overrightarrow{D, S_0, 3, 2, 0}$ that sums bitcounts (popcounts) to find the positions in memory for the next level. The bitvector format thus demonstrates how SAM handles various stream types as different compression protocols on the wires (coordinates versus bitvectors), along with various reference stream protocols, while maintaining composability.

4.4 Parallelization

Given the spatial streaming abstraction in SAM, parallelism is easily representable via vectorization and graph duplication. Conceptually the simplest extension is to vectorize streams as wire buses and to update the blocks to handle the increased data rates.

To enable coarse-grained parallelism, SAM dataflow graphs can fork streams with a parallelizer and join streams with a serializer. The parallelizer block takes in a sequential tensor stream and parcels out different elements to multiple output streams concurrently. The serializer block works inversely and joins parallel streams into a sequential stream by interleaving their coordinates.

5 THE CUSTARD COMPILER

The Compiler for Unified Sparse Tensor Algebra Reconfigurable Dataflows (Custard) is our compiler to SAM which acts as an intermediate representation. Custard compiles tensor algebra expressions with associated data structure specifications [11] and schedules [26, 48] to SAM dataflow graphs (see Figure 10). Custard is an open-source C++ project that utilizes the TACO front-end [27] but supplies a new lowerer and code generator. Custard uses TACO's three input APIs (tensor index notation, a format language, and a scheduling language) and the code that transforms these into a high-level IR called concrete index notation—an abstract loop nest with scheduling information shown in Figure 10. Although Custard generates SAM dataflow graphs, automatic binding to prior-work hardware backends described in Section 6.5 is left as future work.

Figure 10 illustrates a partial compilation to the SAM dataflow graph for the ikj-order SpM*SpM example from Section 3.4. Custard converts the concrete index notation to a graph that represents each tensor's path through the index variables (shown as colored arrows with tensor labels in Figure 10). Custard then builds the following three sections in order: tensor iteration and merging, computation, and tensor construction. It builds the tensor iteration and merging by iterating over the Cartesian product of index variables and input tensors, which in our example is $\{i, k, j\}$ by $\{B, C\}$. For every index variable in a tensor's path, Custard places and connects a level scanner, which we color corresponding to its associated tensor path. For every index variable absent from a tensor's path that does not have an outer index variable reduction, Custard inserts a repeat block. Finally, if multiple tensor paths exist for an index variable, then Custard inserts an intersecter (for multiplication) or unioner (for addition). Next, the output reference streams from the first part are connected to the compute tree, which consists of scalar operations and reductions, (extracted from the concrete index notation). Finally, the output values from the computation section and each index variable's final coordinate stream are connected to the output construction blocks (denoted by the orange in Figure 10) with coordinate drop blocks inserted as necessary.

6 EVALUATION

We use Custard to compile disparate sparse tensor algebra algorithms into SAM graphs that are automatically lowered to a cycle-approximate functional simulator. The Custard code, used to automatically compile SAM graphs, is compiled using GCC 9.4.0. The SAM lowering and SAM simulator are written in Python 3.8. Our SAM simulator tracks each cycle iteration and models SAM graphs

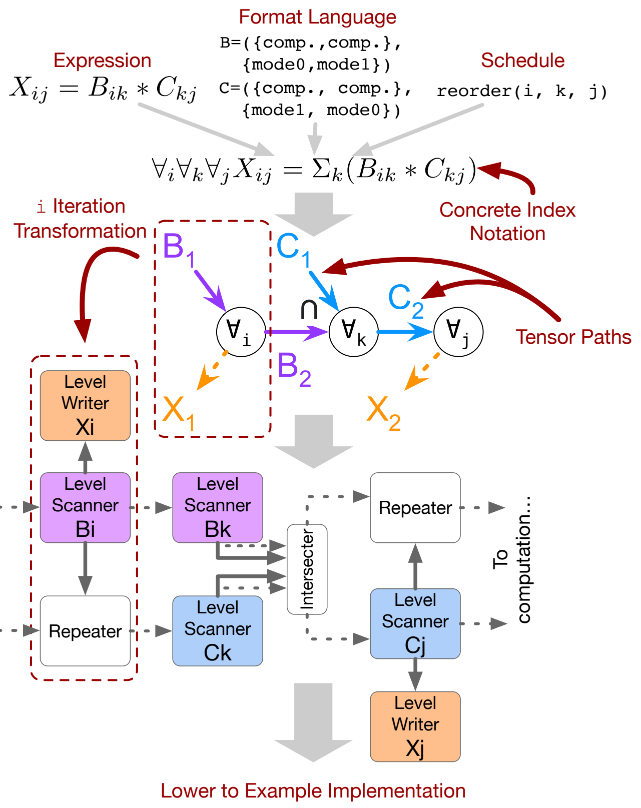


Figure 10: Custard's steps for compiling SAM tensor iteration, merging, and construction for the SpM*SpM example in Section 3.4. We abbreviate compressed as comp. From top to bottom, Custard uses the TACO input APIs to generate concrete index notation, creates index-variable paths for each tensor, and constructs the partial SAM graph (where the color of each block corresponds to a tensor path—purple for B, blue for C, and orange for X). Custard lowers the dotted red region of the tensor paths to the dotted red region of SAM blocks.

as fully pipelined (i.e. every primitive produces one token each cycle). It is cycle approximate since we assume for this section—except *Modeling Hardware with Finite Constraints* in Section 6.4—that: input queues are infinite, data fetched from arrays (memory) take only one cycle, memories are pre-initialized, and primitives are not time-shared. These assumptions do not affect our evaluation conclusions since this section only contains comparisons against simulator cycles. All benchmarks are run on a 2.2 GHz Intel Xeon Silver 4214 24-core CPU with a 16 MB LLC running Ubuntu 18.04.

6.1 Empirical Study of the Generality of SAM

We demonstrate the generality of SAM by generating dataflow graphs for a wide range of useful sparse tensor algebra expressions, shown in Table 1, including all expressions used in the TACO paper [27]. These real-world applications comprise of algorithms such as factor analysis (e.g. alternating least squares), graph convolutional networks, and tensor factorization and decomposition

Table 1: SAM primitive counts for a wide range of real-world expressions, demonstrating the expressiveness and generality of SAM. Each expression also contains a breakdown of the sparse tensor algebra features it contains. All features and primitive counts are obtained assuming the expression uses an alphabetical dataflow index-variable ordering except in SpM*SpM.[b]

Name	Expression	Sparse Tensor Algebra Feature						SAM Primitive Composition (count)								
		Out order	Input order	Num. inputs	Reduce order	Broad-cast	Op	Lvl Scan	Rep-eat	Inter-sect	Uni-on	ALU	Red-uce	Crd Drop [a]	Lvl Wr	Arr-ay
SpMV	$x_i = \sum_j B_{ij} c_j$	1	1,2	2	0	✓	*	3	1	1		1	1	1	2	2
SpM*SpM	$X_{ij} = \sum_k B_{ik} C_{kj}$	2	2	2	0–2 [b]	✓	*	4	2	1		1	1	0–2 [b]	3	2
SDDMM	$X_{ij} = \sum_k B_{ij} C_{ik} D_{jk}$	2	2	3	0	✓	*	6	3	3		2	1	2	3	3
InnerProd	$\chi = \sum_{ijk} B_{ijk} C_{ijk}$	0	3	2	0	✗	*	6		3		1	3		1	2
TTV	$X_{ij} = \sum_k B_{ijk} c_k$	2	1,3	2	0	✓	*	4	2	1		1	1	2	3	2
TTM	$X_{ijk} = \sum_l B_{ijl} C_{kl}$	3	2,3	2	0	✓	*	5	3	1		1	1	3	4	2
MTTKRP	$X_{ij} = \sum_{kl} B_{ikl} C_{jk} D_{jl}$	2	2,3	3	0	✓	*	7	5	3		2	2	3	3	3
Residual	$x_i = b_i - \sum_j C_{ij} d_j$	1	1,2	3	0	✓	*,-	4	1	1	1	2	1	1	2	3
MatTransMul	$x_i = \sum_j \alpha B^T_{ij} c_j + \beta d_i$	1	0-2	5	1	✓	*,+	4	4	1	1	4	1	1	2	5
MMAdd	$X_{ij} = B_{ij} + C_{ij}$	2	2	2	✗	✗	+	4			2	1			3	2
Plus3	$X_{ij} = B_{ij} + C_{ij} + D_{ij}$	2	2	3	✗	✗	+	6			2	2			3	3
Plus2	$X_{ijk} = B_{ijk} + C_{ijk}$	3	3	2	✗	✗	+	6			3	1			4	2

[a] Coordinate dropper primitive counts in the SAM graphs assume that the reducer is configured to filter out and remove reductions of empty fibers.
[b] In SpM*SpM we show the features and primitive counts for all dataflow orderings: inner product, linear combination of rows, and outer product.

Table 2: The number of sparse tensor algebra algorithms from the TACO website that are not expressible if a SAM primitive is removed. The All and Unique columns analyze all input algorithms and distinct algorithms, respectively.

		Expressions Lost		Percentage (%)	
	SAM Primitive Removed	Unique	All	Unique	All
1	Comp. Level Scanner	2773	19363	72.23	81.38
2	Comp. + Uncomp. Level Scanners	3814	23713	99.35	99.66
3	Repeater	3162	19924	82.37	83.74
4	Unioner	600	2229	15.63	9.37
5	Intersecter keep Locator	720	2715	18.75	11.41
6	Intersecter w/ Locator Removed	1878	15777	48.92	66.31
7	Adder	1023	3118	26.65	13.1
8	Multiplier	3220	20986	83.88	88.2
9	Reducer	3008	20036	78.35	84.21
10	Coordinate Dropper	617	2292	16.07	9.63
11	Comp. Level Writer	1075	5525	28	23.22
12	Comp. + Uncomp. Level Writers	3698	23260	96.33	97.76

for domains like machine learning, data analytics, and scientific computing. Table 1 lists the sparse tensor algebra features used by each expression and the number of primitives it uses (empty cells denote a primitive is not used). We see the primitive counts for level scanners and writers are higher, since they are used for tensor iteration and correspond to the tensor orders of all inputs and outputs respectively. The primitive composition counts also show that most blocks are uniformly used. It is interesting to note that two expressions use all primitive types. In addition, we automatically lowered all graphs to our simulator and checked for functional correctness on the set of all real and integer SuiteSparse matrices [14] and FROSTT tensors [49] that fit into memory and the Facebook tensor [58].

6.2 Ablation Study on the Utility of SAM Blocks

Each SAM block in Section 3 is essential for expressing the domain of sparse tensor algebra for dataflow. In order to demonstrate the usefulness of each primitive, we analyze the entire set of algorithms

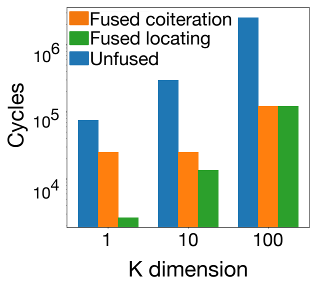


Figure 11: Performance of fused and unfused SDDMM algorithms.

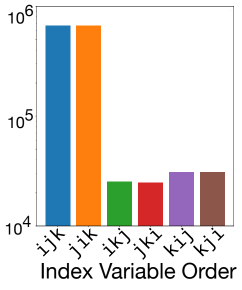


Figure 12: Performance of SpM*SpM dataflows.

input by users into the TACO website, provided by the TACO authors, and show which algorithms are not expressible if a given SAM primitive does not exist. We use the TACO website as our dataset since it is representative of real-world sparse tensor algebra computations. From the website, users have successfully compiled 23,794 sparse tensor algebra algorithms to date—of which 3,839 were distinct algorithms (unique combinations of expression and format) and 1,745 were unique solely in expression. Table 2 shows that removing any SAM primitive limits the expressible algorithms in the domain of sparse tensor algebra. Most blocks are used for most applications, and, moreover, full algorithmic generality requires all the primitives presented in Section 3.

6.3 Asymptotic Tradeoff Analysis

We next explore the performance attributed to: dataflow ordering, fusion, and various acceleration techniques. While the former fundamentally change the dataflow of the computation, the other optimizations presented are orthogonal and, only affecting a single tensor level, can be used in conjunction with any dataflow.

Dataflow Ordering. The index-variable order avoids different data-dependent asymptotic behaviors [23, 61] and allows for generality in the execution of a particular dataflow algorithm. We simulate all six permutation orders of ijk for the SpM*SpM expression using two distinct 95% sparse uniformly random matrices with different dimensions of sizes $I = J = 250$ and $K = 100$. Figure 12 shows the inner-product algorithms (ijk, jik) perform the worst for matrix multiply. The linear combination of rows (ikj, jki) and outer product (kij, kji) algorithms perform a least an order of magnitude better. The performance is dictated by the order of k since coordinates are filtered out (intersected) at k earlier in the dataflow before repeating along the other dimensions i, j. These algorithms differ in their asymptotic complexity [2, 28], so performance differences will increase with increases in sparsity. However, the inner-product algorithm may be more efficient with other data and uses asymptotically less memory for the reduction (a scalar instead of a row). Since the efficiency choice is a tradeoff, sparse hardware should support many processing orders.

Fusion. We demonstrate the algorithmic performance advantage of fusion using a common expression from machine learning, the ijk-ordered SDDMM $X_{ij} = \sum_k B_{ij} C_{ik} D_{jk}$ [4, 17]. We generate a 95% sparse uniformly random matrix along with two dense matrices of dimensions $I = J = 250$ with a sweep of $K = \{1, 10, 100\}$. Figure 11 shows that the unfused implementation performs far worse, since calculating the entire dense matrix multiplication is costly with mostly wasted work. Given the number of nonzeros in B as nnz_B, the unfused computation complexity is proportional to $\max(\text{nnz}_B * K, \text{locate}(\text{nnz}_B))$, while the cost of factorization becomes $I * J * K + \text{locate}(\text{nnz}_B)$. The only case where we would want to factorize this expression is when the matrix B is almost fully dense and we have very efficient dense matrix multiplication hardware. But for a sufficiently sparse matrix, a fused expression will perform far better. Efficient sparse hardware must therefore support fused expressions.

We further enhance performance by using locator blocks (Section 4.2) to find the sampled i, j values, which is trivial in a dense array. Interestingly, Figure 11 shows that this advantage becomes negligible as K increases: iteration costs of the dense inner-product dimension k will dominate the computation time, hiding the benefits of locating during intersection. But locating provides significant performance gains when the amount of computation is modest, which is often true in sparse computations.

Accelerator Structures. We next explore different iteration acceleration techniques by comparing various configurations of coordinate-skipping (Section 4.2), bitvector iteration (Section 4.3), and iteration-splitting (Section 4.1). Figure 13 compares the performance when both vectors are in the following formats: one uncompressed level (Dense), one compressed coordinate level (Crd), one compressed coordinate level with coordinate-skipping (Crd w/ skip), two compressed coordinate levels (Crd w/ split), one pseudo-dense bitvector level (BV), and two bitvector levels (BV w/ split), also known as a bit-tree. For this set of experiments, we assume the coordinates were already split before this operation[1] and use the vector-vector element-wise multiply expression $x_i = b_i * c_i$ with both b and c as single dimensional vectors of size 2000. We use three types of synthetic vectors, namely *urandom*, *runs*, and *blocks*; runs and blocks are shown in Figure 17. Vectors with *runs* are pairs of vectors where one vector will have longer stretches of nonzeros between the nonzeros of the other vector. Similarly, *blocks* are vectors which have dense blocks of nonzeros placed throughout the vector. For both these vectors, the number of nonzeros is 400 (20%) with the index indicating the size of the runs/blocks in each vector.

[1]The splitting operation requires a full scan through the data structure, which for this example is as expensive as the operation itself.

Figure 13a shows the performance as a function of sparsity for *urandom* data with bitvector bitwidth $b = 64$ and split factor (how many chunks the vector is divided up into) $s = 64$, where applicable, and shows the limitations of a single-level bitvector. As the sparsity increases, the compressed coordinate format becomes better than the bitvectors, since bitvectors are still a dense representation. The coordinate-skipping behaves exactly the same as the compressed coordinate format since *urandom* tensors on average have small (around 1.5) run lengths.

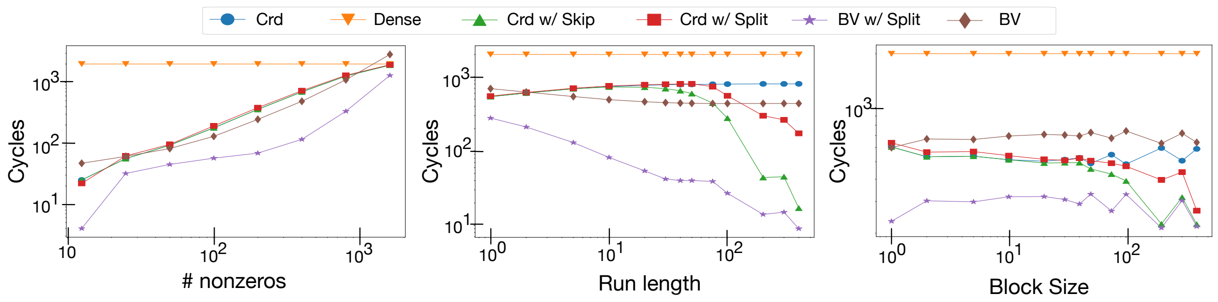
Figure 13b shows the utility of coordinate skipping and splitting. As run lengths increase, there are more opportunities to skip invalid input coordinates or avoid computation at the outer-level intersection. The bitvector remains flat since the number of nonzeros remains about the same for various run lengths. This advantage of skipping and splitting remains in the *blocks* case, without the dependence on block size, since intersections can also be dense. Overall, these results show the advantage of the implicit parallelism of bitvectors, but show that they need to be organized hierarchically for robust performance.

6.4 Modeling Exploration

Stream Analysis. We analyze the token breakdown of the SAM flattened stream representation and identify that the stream control overhead is modest. We use Custard to compile the SAM graph for the matrix identity expression $X_{ij} = B_{ij}$, where B is a sparse DCSR matrix, and count the token types for each coordinate stream at the output of each level scanner. In our simulator, we model streams as Python lists and all control tokens as strings.[2] We run the expression on 15 matrices of various sizes from the SuiteSparse matrix collection [14] (see Table 3 in the Appendix for matrix characteristics and selection criteria).

The control token overhead of our representation is reasonable, with an average non-idle control overhead reaching 0.95% for outer levels and 16.20% for inner levels as shown in Figure 14. (Section 3.8 shows the control overhead of the alternative of using non-flattened point streams would be higher.) The average inner-level percentage means that rows have an average of 5 nonzeros, an appropriate number of coordinates for this set of matrices. The outer-level B_i stream and inner-level B_j stream refer to the coordinate stream outputs of the first B_i level scanner and the second B_j level scanner, respectively. We do not show the B_{vals} breakdown since it is the same as B_j. Most tokens, on average 83.32%, on the B_i stream are idle since the B_i level scanner is in the done state while the inner-level iterates through its coordinates. This behavior occurs in compressed arrays, as in Figure 1c, because there are exponentially more coordinates for each lower level of a tensor. The done state of the primitive is efficient as it is idle and avoids computation

[2]In hardware implementations, however, one possible way to implement the control tokens would be as a tagged-union on the wire. There are alternative implementations, like hardcoding the control token level for each primitive, which removes the need for a stop level in the stream but complicates and hardens the state-machine logic of each primitive.



(a) Performance vs. sparsity of uniformly random synthetic vectors on a log-log scale **(b) Performance vs. run length of synthetic vectors with runs on a log-log scale** **(c) Performance vs. block size of blocked synthetic vectors on a log-log scale**

Figure 13: Simulated performance of various optimization techniques (compression, splitting, skipping, and bitvectors) for sparse vector sparse vector element-wise multiplication where the vectors have a dense dimension size of 2000.

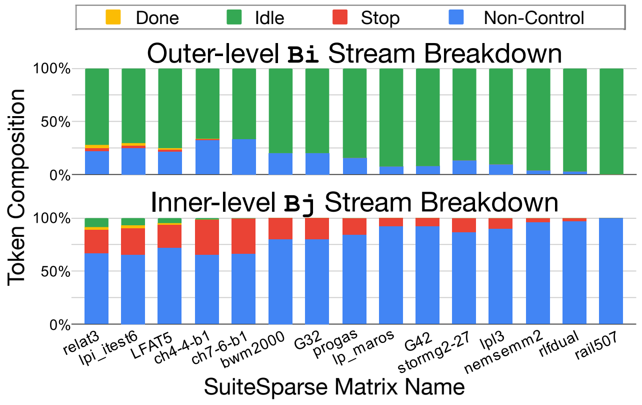


Figure 14: Breakdown of the outer B_i level stream and inner B_j level stream by token type for the matrix identity expression $X_{ij} = B_{ij}$ where B is a sparse DCSR matrix

activity. Improving efficiency and utilization of idle primitives could include switching the outer level scanner to other tasks through time multiplexing, which we leave as future work. At the lower level of the matrix, control overhead is dominated by stop tokens. The stop token overhead ranges from 0.12% (for `rail507`) to 33.26% (for `ch7-6-b1`). Again, these breakdowns are reasonable since higher percentages of stop tokens occur only in small matrices.

Modeling Hardware with Finite Constraints. Although SAM is an abstract machine with infinite resources, it can also represent finite hardware with finite memory. ExTensor [22] is one design point in the space of sparse tensor algebra accelerators, and SAM is sufficiently expressive to model it. We find that SAM can recreate the performance characteristics of ExTensor's evaluation. We recreate the synthetic data study, Figure 19 Section 8.4, in the ExTensor paper that measures “SpM*SpM performance across varying dimension sizes with a constant number of nonzeros per matrix” as shown in Figure 15. Our SAM model contains the hierarchical coordinate skipping, fixed-memory tiling, sparse tile skipping, and n-buffering techniques of ExTensor. Our performance matches the three performance regions of the ExTensor study: increasing runtime due to

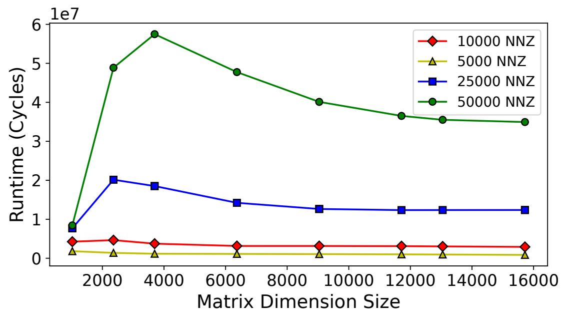


Figure 15: Recreation of ExTensor's “SpM*SpM performance across varying dimension sizes with a constant number of nonzeros per matrix” study using our SAM simulator.

more non-empty tiles at small dimensions, decreasing runtime due to sparse tile skipping, and decreasing runtime with saturating performance. Concretely, we model two levels of memory hierarchy, a last-level buffer (LLB) and a processing element buffer (PEB). SAM is used to sequence the sparse tile coordinates including the CPU loop-nests and using a DRAM bandwidth of 68.256 GB/s, an LLB size of 17MB, and a processing element (PE) tile size of 128×128 (configured using implementation-specific information).

6.5 Backend Case Studies

By construction, we designed SAM to easily represent dataflow hardware. To evaluate its likeness and ability to bind to hardware, we qualitatively analyze how SAM is able to represent fixed-function and reconfigurable dataflow backends including Gamma [64], OuterSPACE [38], ExTensor [22], and Capstan [47]. For example, Gamma's dataflow is similar to Figure 4. The main difference is that Gamma adds a parallelizer after the intersection unit and then uses a multi-input vector reducer to rejoin the parallel threads.

For space reasons, we only provide a concrete SAM graph for OuterSPACE (see Figure 16), which leverages an outer-product dataflow ($k \rightarrow i \rightarrow j$). We chose OuterSPACE because it factorizes SpM*SpM into two stages: a multiply phase ($Y_{ikj} = B_{ik}C_{kj}$) and a merge phase ($X_{ij} = Y_{ikj}$), thus showing how SAM supports factorization. For efficiency, B_{ik} and C_{kj} are respectively stored

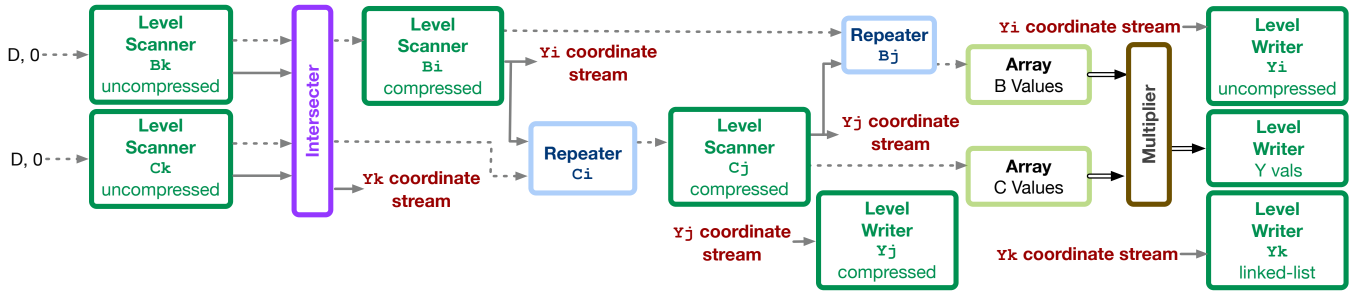


Figure 16: The SAM dataflow graph for SpM*SpM that represents the OuterSPACE multiply phase, which is followed by the OuterSPACE merge phase. Compare this outer product graph to the linear combination graph in Figure 4.

in column-major and row-major order. The first phase computes outer products between all columns of B and all rows of C and stores the partial result into a 3-dimensional tensor Y_{ikj}, as shown in Figure 16. To efficiently merge in the next phase, the intermediate result Y is stored in ikj-order, which is discordant with the dataflow kij. To efficiently support a discordant write of the tensor streams, OuterSPACE utilizes a linked-list representation as the level-format for k. Because our level writer is not restricted to a specific representation, SAM supports this dataflow. The merge phase (not shown to conserve space) then accumulates the partial product Y_{ikj} from the previous phase into a final result X_{ij}. This dataflow consists of three cascaded level scanners to generate the values Y_{ikj} that need to be summed, a vector reducer to sum the k dimension, and three level writers to store the X_{ij} results.

ExTensor's Stream Coordinator [22] is naturally representable with SAM. It is a hardware unit consisting of, in order, their Stream Sequencer, two of their Scanners, their Intersection unit, and two Data Storage units. The Stream Sequencer (and its `Configure()` command) is representable with two SAM repeater primitives, with the added benefit that SAM's repeaters do not have to be pre-configured. We represent their Scanner as a composition of n SAM level scanners with coordinate-skipping since their Scanner can scan through n levels of a fibertree. Finally, their Intersect and Data Storage units are equivalent to our intersecter and array primitives. We note that each ExTensor Coordinator is fixed for two input tensors (hence two Scanners with Data Storage) with the intersection always occurring at the last level of each unit. These implementation choices limit Extensor-like SAM graphs to a subset of the entire space producible by SAM.

Finally, we analyze the Capstan [47] specialized loop-header hardware, since it is one of the main contributions of that work. Capstan can represent this specialized hardware as a two-operand bitvector scanner that can be configured with `or` or `and`. Using SAM, the hardware is equivalent to two bitvector level scanners (one for each tensor) followed by an intersecter for `and` or a unioner for `or`. The output values produced by the Capstan hardware: addresses for both tensors, a store address, and a dense index correspond to SAM's post-intersect/union reference streams, result level writer address generation, and post-intersect/union coordinate stream, respectively. The vectorized loop bodies in Capstan are representable using n-lane stream buses (bundles), SAM arrays to get the data, a single SAM ALU, and level writers. Again, the class of SAM graphs that represent a single Capstan loop-header is fixed to two sparse operands with only a subset of SAM's expressibility. Additionally, the Capstan-like SAM only iterates through bitvectors, which is great for vectorization but is fixed to a pseudo-dense iteration space.

7 CONCLUSION

We introduced the Sparse Abstract Machine, an abstract machine model for both reconfigurable and fixed-function spatial dataflow accelerators for sparse tensor algebra. Our design led to a stream representation and a small set of physical blocks with well-defined interfaces. Our Custard compiler demonstrates SAM's utility as a compiler target. In addition, the flexibility and generality of SAM let us fairly evaluate optimization and dataflow alternatives for accelerating sparse tensor algebra algorithms. We hope that the Sparse Abstract Machine model will enable the microarchitectural design of future accelerators and inform the design decisions of architects. We also hope that compiler designs like Custard, targeting an abstract machine for portability, will improve the programmability and usability of this space.

ACKNOWLEDGMENTS

We thank Manya Bansal, Zimren Dixon, Scott Kovach, Zachary Myers, Aviral Pandey, Alexander Rucker, Matthew Sotoudeh, Shiv Sundram, Joseph Tan, and Rohan Yadav for their helpful feedback. We would also like to thank Ajay Brahmakshatriya, Jake Ke, Kalhan Koul, Qiaoyi Li, Keyi Zhang, Saman Amarasinghe, Riyadh Baghdadi, and Priyanka Raina for discussion and help with evaluation. Olivia Hsu was supported by an NSF GRFP Fellowship, Maxwell Strange was supported by the Apple Stanford Electrical Engineering PhD Fellowship in Integrated Systems, and Ritvik Sharma was supported by the Stanford Graduate Fellowship. This work was supported in part by the NSF under grant numbers 1937301, 2028602, CCF-1563078, and 1563113; by DARPA under the Domain-Specific System on Chip (DSSoC) Program; and by the DoE National Nuclear Security Administration (NNSA) under grant number DE-NA0003965. This research was also supported in part by the Stanford Agile Hardware (AHA) Center, Google Research Scholar program, and Stanford Data Analytics for What's Next (DAWN) Affiliate Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

A ARTIFACT APPENDIX

A.1 Abstract

This appendix describes how to set up and run our Sparse Abstract Machine (SAM) Python simulator and the C++ CUSTARD compiler, which compiles from concrete index notation (CIN) to SAM graphs (represented and stored in the DOT file format). The appendix also describes how to reproduce the quantitative experimental results in this paper. The artifact can be executed with any X86-64 or M-series Apple machine with Docker, Python 3, Git, and Bash support, at least 32 GB of RAM, and more than 20 GB of disk space.

A.2 Artifact Check-List (Meta-Information)

- **Compilation:** C++ compiler (either `gcc` or `clang`). The `gcc` 9.4.0 compiler is included with the Docker image. A fork of the TACO compiler (found here) is included as a submodule in our artifact (fork `weiya711/taco`, commit hash `cf8f007`).
- **Data set:** Suitesparse Matrix Market matrices (a script to download the dataset is included and the full dataset can be found at https://sparse.tamu.edu/), Frostt Tensor Dataset tensors (a script to download the dataset is included and the full dataset can be found at http://frostt.io/), and synthetically generated matrices/higher-order tensors (included).
- **Run-time environment:** Docker, git, Python 3, and bash need to be installed on the local machine. Docker is available for many operating systems. Proficiency in bash and git is recommended.
- **Hardware:** Any conventional x86 CPU with at least 32 GB of RAM should work.
- **Metrics:** Cycles (modeled as iteration counts in our simulator and source code), expression counts, and primitive counts
- **Output:** Terminal outputs, files, tables, and graphs (PDF figures, PNG figures, and DOT file format [15] graphs). Expected results are included in the submitted paper.
- **Experiments:** All steps are detailed in the `README.md` in https://github.com/weiya711/sam-artifact. The steps include pulling a Docker image and running/attaching a container, running scripts within the docker, running one Python 3 script locally outside of the Docker to copy results, and verifying result images/files. The experiments should have less than 5% variation since the simulator is deterministic. The 5% variation is caused by different data patterns in synthetic data generation (even with sparsity held constant due to random statistics). However, these variations do not affect the paper's conclusions.
- **How much disk space required (approximately)?:** Approximately 20GB of space should be sufficient.
- **How much time is needed to prepare the workflow (approximately)?:** About 10-15 minutes.
- **How much time is needed to complete experiments (approximately)?:** To complete all experiments it takes approximately 65 hours. We also include scripts to complete a subset of the experiments, which include Table 1, Table 2, Figure 11, Figure 12, Figure 13, Figure 14, and only 8 points in Figure 15, that takes about 10 hours to run on a standard machine.
- **Publicly available?:** Yes, on Github at the *sam* repository (https://github.com/weiya711/sam) for active development of source code and at the *sam-artifact* repository (https://github.com/weiya711/sam-artifact) for the artifact evaluation of this paper. The specific commits for this artifact are tagged with `asplos23-ae` in both repositories.
- **Code licenses (if publicly available)?:** MIT License
- **Workflow framework used?:** Docker
- **Archived (provide DOI)?:** Yes, the DOI is https://doi.org/10.5281/zenodo.7591742.

A.3 Description

A.3.1 How to Access. The code repository for this submission can be downloaded from https://github.com/weiya711/sam-artifact. The repository includes a Dockerfile from which a Docker image can be built for full evaluation of the artifact.

A.3.2 Hardware dependencies. We recommend a machine with a conventional x86 CPU and at least 32GB of memory. We found that some of the experiments will be OOM killed on a machine with only 16GB of memory.

A.3.3 Software Dependencies. Evaluation of the artifact requires a machine with Docker and Python 3 installed. We tested the artifact evaluation on the following configurations and found them to work: Ubuntu 20.04/Docker 20.10.12/Python 3.8 (AMD-based machine), and MacOS 13.1/Docker 20.10.22/Python 3.9 (Intel-based machine). We expect other versions of MacOS, Ubuntu, Docker, and Python 3 configurations to work as well.

Table 3: Matrices from the SuiteSparse matrix collection [14] used to analyze the overhead of our stream representation in matrix identity (Section 6.4). We randomly selected each set of 5 matrices (delineated in the table above) from the smallest, median, and largest 50 SuiteSparse matrices—based on dense dimension size—that would fit in memory.

Name	Domain	Dimensions	Nonzeros	Density (%)
relat3	Combinatorics	8×5	24	60.0
lpi_itest6	Linear Programming	11×17	29	15.5
LFAT5	Model Reduction	14×14	46	23.5
ch4-4-b1	Combinatorics	72×16	144	12.5
ch7-6-b1	Combinatorics	630×42	1260	4.8
bwm2000	Chemical Process Simulation	2000×2000	7996	0.2
G32	Undirected Weighted Random Graph	2000×2000	8000	0.2
progas	Linear Programming	1650×1900	8897	0.3
lp_maros	Linear Programming	846×1966	10137	0.6
G42	Undirected Weighted Random Graph	2000×2000	23558	0.6
stormg2-27	Linear Programming	$14,439 \times 37,485$	94274	0.02
lpl3	Linear Programming	$10,828 \times 33,686$	100525	0.03
nemsemm2	Linear Programming	$6943 \times 48,878$	182012	0.05
rlfdual	Linear Programming	$8052 \times 74,970$	282031	0.05
rail507	Linear Programming	$507 \times 63,516$	409856	1.3

A.3.4 Data sets. The evaluation requires matrices from the Suitesparse Matrix Market dataset (script to download the dataset is included, full dataset can be found at https://sparse.tamu.edu/), the Frostt Tensor Dataset (script to download the dataset is included, full dataset can be found http://frostt.io/), and synthetically generated matrices/higher-order tensors (included in the artifact evaluation). The synthetic data generation pattern for Section 6.3 is shown in Figure 17.

A.4 Installation

To install, first clone the *sam-artifact* repository to the local machine and initialize all submodules, then build the docker image:

```
$ git clone https://github.com/weiya711/sam-artifact
$ cd sam-artifact
```

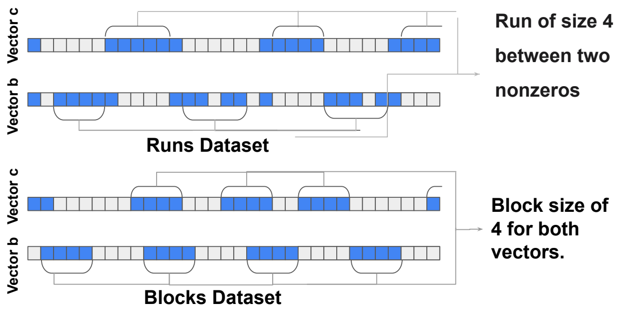

Vector c
Vector b
Run of size 4 between two nonzeros
Runs Dataset
Vector c
Vector b
Block size of 4 for both vectors.
Blocks Dataset

Figure 17: Sample *runs* and *blocks* vector data patterns used for our synthetic data generation in Section 6.3.

```
$ git submodule update --init --recursive
$ docker build -t sam-artifact .
```

The docker container can be started with the following command:

```
$ docker run -d -it --rm sam-artifact bash
```

A docker container ID will be printed upon completion of this command, and the container can be attached to with:

```
$ docker attach <CONTAINER_ID>
```

Once inside the container, a fire test can be conducted via the commands:

```
$ cd /sam-artifact/sam/
$ python scripts/collect_node_counts.py
```

A.5 Experimental Workflow

The experimental workflow for this artifact includes running a set of scripts within the Docker environment to generate tables/figures from the paper. The complete instructions can be found in the README.md included within the *sam-artifact* repository.

A.6 Evaluation and Expected Results

The following subsection includes information on how to reproduce all the automatically generated result figures/tables in the paper. We note that the left-hand side of Table 1 (Sparse Tensor Algebra Features) and Figure 16 were manually derived by the authors and have no associated source code.

Tables 1, Table 2 and Figures 11–14 can be generated with the following commands:

```
# In Docker Container
$ cd /sam-artifact
$ source scripts/generate_all_results.sh
ctrl-p ctrl-q   # Detach from Docker container
# In local machine
$ python sam/scripts/artifact_docker_copy.py \
    --output_dir <OUTPUT_DIRECTORY> \
    --docker_id <DOCKER_ID>
```

The expected results for the above commands are:

- **Table 1**: The standard output from the fire test, which is also saved at /sam-artifact/sam/tab1.log in the Docker container, should match the right hand side of Table 1. The left-hand side (Sparse Tensor Algebra Features) contains manually derived summaries of the expressions.
- **Table 2**: The file /sam-artifact/taco-website/tab2.log in the Docker container should match Table 2.
- **Figure 11**: The file fig11.pdf on the local machine should match Figure 11.
- **Figure 12**: The file fig12.pdf on the local machine should match Figure 12.
- **Figure 13**: The files fig13a.pdf, fig13b.pdf, and fig13c.pdf on the local machine should match Figure 13a, Figure 13b, and Figure 13c respectively.
- **Figure 14**: The file fig14.pdf on the local machine should match Figure 14.

Figure 15 generation is time-consuming to compute so we have given three options—one data point, a few data points, or all data points—each with size configurations, leading to 5 options total.

```
# In Docker container
$ cd /sam-artifact/sam
# [Option 1] Choose and run one point from Figure 15
$ ./scripts/single_point_memory_model_runner.sh \
    extensor_<NNZ>_<DIMSIZE>.mtx
# [Option 2] Run eight points from Figure 15
$ ./scripts/few_points_memory_model_runner.sh <GOLD>
# [Option 3] Run all points from Figure 15
$ ./scripts/full_memory_model_runner.sh <GOLD>
ctrl-p ctrl-q   # Detach from Docker container
# In local machine
$ python sam/scripts/artifact_docker_copy.py \
    --output_dir <OUTPUT_DIRECTORY> \
    --docker_id <DOCKER_ID>
```

where NNZ ∈ {5000, 10000, 25000, 50000}, DIMSIZE ∈ range(1024,15721,1336), and GOLD ∈ {0,1}, where 0 means no gold checking and 1 includes gold checking. A single point run has gold enabled by default and runs for between 20 minutes to 17 hours, depending on which NNZ and DIMSIZE combination is chosen. The few points script takes approximately 8 hours to run with no gold and 19 hours to run with gold enabled. The full script takes approximately 64 hours to run with no gold and 92 hours to run with gold enabled.

- **Figure 15**: The file fig15.pdf on the local machine should contain a subset of scatter points that match Figure 15.
- **Figure 16:** This figure is manually derived.

A.7 Experiment Customization

Detailed experiment customization can be found in the *sam-artifact* README.md section titled **How to Reuse Artifact Beyond the Paper**.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Peter 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)* (San Diego, CA, USA) *(PLDI 2022)*. Association for

Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

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

[4] Vivek Bharadwaj, Aydin Buluç, and James Demmel. 2022. Distributed-Memory Sparse Kernels for Machine Learning. https://doi.org/10.48550/ARXIV.2203.07673

[5] 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 (TACO)* 19, 4 (2022), 1–25.

[6] A. Canning, G. Galli, F. Mauri, A. De Vita, and R. Car. 1996. O(N) tight-binding molecular dynamics on massively parallel computers: an orbital decomposition approach. *Computer Physics Communications* 94, 2 (April 1996), 89–102. https://doi.org/10.1016/0010-4655(96)00009-4

[7] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. IEEE Symposium on VLSI Technology & Circuits.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[9] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[10] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

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

[12] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the value of flexibility for graph processing accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 595–608.

[13] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 924–939.

[14] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[15] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. 2002. Graphviz— Open Source Graph Drawing Tools. In *Graph Drawing*, Petra Mutzel, Michael Jünger, and Sebastian Leipert (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 483–484.

[16] Richard Feynman, Robert B. Leighton, and Matthew L. Sands. 1963. *The Feynman Lectures on Physics. Vol. 3.* Addison-Wesley.

[17] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. *Sparse GPU Kernels for Deep Learning*. IEEE Press, Chapter 17, 1–14.

[18] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978).

[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[20] 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 Bret, Allan Haldane, Jaime Fernández del Rio, 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 (2020), 357–362.

[21] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. *Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices.* Association for Computing Machinery, New York, NY, USA, Chapter 19, 1–12. https://doi.org/10.1145/3392717.3392751

[22] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 319–333.

[23] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of Sparse Array Programming Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (October 2021), 29 pages. https://doi.org/10.1145/3485505

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

[25] Jeremy Kepner and John R. Gilbert (Eds.). 2011. *Graph Algorithms in the Language of Linear Algebra.* Software, environments, tools, Vol. 22. SIAM. http://dblp.uni-trier.de/db/books/collections/KG2011.html

[26] Fredrik Kjolstad, Peter 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

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

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

[29] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018).* Association for Computing Machinery, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[30] Tamara G. Kolda and Jimeng Sun. 2008. Scalable Tensor Decompositions for Multi-aspect Data Mining. In *2008 Eighth IEEE International Conference on Data Mining.* 363–372. https://doi.org/10.1109/ICDM.2008.89

[31] Scott Kovach and Fredrik Kjolstad. 2022. Correct Compilation of Semiring Contractions. https://doi.org/10.48550/ARXIV.2207.13291

[32] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO.* San Jose, CA, USA, 75–88.

[33] Qiaoyi Liu, Dillon Huff, Jeff Setter, Maxwell Strange, Kathleen Feng, Kavya Sreedhar, Ziheng Wang, Keyi Zhang, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2021. Compiling Halide Programs to Push-Memory Accelerators. *CoRR* abs/2105.12858 (2021). arXiv:2105.12858 https://arxiv.org/abs/2105.12858

[34] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, New York, NY, USA, 1009–1022. https://doi.org/10.1145/3352460.3358254

[35] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. In *Languages and Compilers for Parallel Computing: 33rd International Workshop, LCPC 2020, Virtual Event, October 14-16, 2020, Revised Selected Papers.* Springer-Verlag, Berlin, Heidelberg, 87–103. https://doi.org/10.1007/978-3-030-95953-1_7

[36] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21).* Association for Computing Machinery, New York, NY, USA, 1064–1077. https://doi.org/10.1145/3466752.3480048

[37] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).* 416–429. https://doi.org/10.1145/3079856.3080255

[38] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 724–736.

[39] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13).* Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[40] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium*

on Computer Architecture (ISCA). 27–40. https://doi.org/10.1145/3079856.3080254

[41] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 651–665. https://doi.org/10.1145/2872362.2872415

[42] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 389–402. https://doi.org/10.1145/3140659.3080256

[43] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article 26 (aug 2017), 25 pages. https://doi.org/10.1145/3107953

[44] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[45] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

[46] M.M.G. Ricci and T. Levi-Civita. 1901. Méthodes de calcul différentiel absolu et leurs applications. *Math. Ann.* 54 (1901), 125–201. http://eudml.org/doc/157997

[47] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1022–1035. https://doi.org/10.1145/3466752.3480047

[48] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

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

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

[51] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190. https://doi.org/10.1016/j.jpdc.2014.06.002 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[52] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[53] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702. https://doi.org/10.1109/HPCA47549.2020.00062

[54] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. *Efficient Processing of Deep Neural Networks.* Morgan & Claypool Publishers.

[55] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. *SIGPLAN Not.* 50, 6 (June 2015), 521–532. https://doi.org/10.1145/2813885.2738003

[56] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. 2021. Aurochs: An Architecture for Dataflow Threads. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 402–415. https://doi.org/10.1109/ISCA52012.2021.00039

[57] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: Accelerating Machine Learning from Relational Data. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 309–321. https://doi.org/10.1109/ISCA45697.2020.00035

[58] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. 2009. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*. 37–42.

[59] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. 2022. Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro* 42, 5 (2022), 59–69.

[60] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2022. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. https://doi.org/10.48550/ARXIV.2205.05826

[61] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3519939.3523437

[62] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. *arXiv preprint arXiv:2207.13901* (2022).

[63] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2022. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. https://doi.org/10.48550/ARXIV.2207.04606

[64] 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*. 687–701.

[65] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1041–1054. https://doi.org/10.1109/ISCA52012.2021.00085

[66] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

Received 2022-10-20; accepted 2023-01-19