TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators

Nandeeka Nayak
ndnayak2@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Toluwanimi O. Odemuyiwa
todemuyiwa@ucdavis.edu
University of California, Davis
Davis, California, USA

Shubham Ugare
sugare2@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Michael Pellauer
mpellauer@nvidia.com
NVIDIA
Westford, Massachusetts, USA

Joel S. Emer
emer@csail.mit.edu
MIT/NVIDIA
Cambridge, Massachusetts, USA

ABSTRACT

Over the past few years, the explosion in sparse tensor algebra workloads has led to a corresponding rise in domain-specific accelerators to service them. Due to the irregularity present in sparse tensors, these accelerators employ a wide variety of novel solutions to achieve good performance. At the same time, prior work on design-flexible sparse accelerator modeling does not express this full range of design features, making it difficult to understand the impact of each design choice and compare or extend the state-of-the-art.

To address this, we propose TeAAL: a language and simulator generator for the concise and precise specification and evaluation of sparse tensor algebra accelerators. We use TeAAL to represent and evaluate four disparate state-of-the-art accelerators—ExTensor, Gamma, OuterSPACE, and SIGMA—and verify that it reproduces their performance with high accuracy. Finally, we demonstrate the potential of TeAAL as a tool for designing new accelerators by showing how it can be used to speed up vertex-centric programming accelerators—achieving 1.9× on BFS and 1.2× on SSSP over GraphDynS.

ACM Reference Format:
Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher W. Fletcher, Michael Pellauer, and Joel S. Emer. 2023. TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3623791

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3623791

1 INTRODUCTION

Sparse tensor algebra workloads have exploded in popularity over the past few years, with applications ranging from deep learning [4, 27, 45] to graph algorithms [3, 5, 12, 28, 32, 41] to physical simulations [20, 46, 49]. This surge has been accompanied by a corresponding rise in proposals for custom hardware to service common sparse kernels, e.g., sparse matrix multiply [15, 16, 34, 36, 38, 55, 56].

While these accelerators have the potential to provide dramatic speedup over the best CPU and GPU algorithms, they take significant effort and space to describe, refine, and evaluate. Specifically, sparse accelerators are typically described either with RTL or a block diagram and an accompanying natural language description. The former is verbose and often difficult to comprehend, while the latter is imprecise and often incomplete. Neither makes it easy to model and evaluate the impact of proposed design changes.

The goal of our work is to ameliorate these issues. That is, to enable the precise and concise specification of sparse tensor algebra accelerators, thereby providing a basis for describing, modeling, evaluating, comparing, and extending proposed designs.

We draw inspiration from existing practice in *dense* tensor algebra accelerator design. Here, a number of tools support concise, precise specifications and the derivation of efficient models [25, 35, 54]. To simplify specification, these tools follow the model proposed by Halide [39] and support separately providing a target *algorithm* (i.e., a functional description of the problem, such as an equation in Einstein summation (Einsum) notation [13]) and a *mapping*, expressing when and where in the processor each action (e.g., compute or storage access) occurs [8]. These, together, correspond to a *mapped representation* (e.g., a loop nest), describing both the algorithm and how it is executed. A *model of the target platform* then evaluates the mapped representation to produce metrics like performance and energy.

Yet, dense tensor accelerator modeling techniques cannot support the sparse case. This is due to the novel complexity that arises when trying to efficiently orchestrate and compute on irregularly sparse data. For example, one can accurately model dense kernels using a few summary statistics (like tensor shapes). However, such summary statistics cannot capture the variety in sparsity distributions present in real-world tensors. Indeed, sparse accelerators regularly employ specialized algorithms and mechanisms to cope

with irregular (often low) data reuse, myriad compression formats, additional meta-computation (e.g., intersection), and more. For example, the OuterSPACE accelerator [34] splits sparse-sparse matrix multiply (SpMSpM) into several phases that respectively produce, sort, and consume an array of linked lists representing partial products. Gamma [55] executes the same kernel with two stages that are connected with a high-radix hardware merger to process the data efficiently. SIGMA [38] uses yet a third strategy (irregularly filling a PE array with only non-zero data). And so on. All of the above stems from irregular sparsity, which obviously does not manifest when data is dense.

Likewise, existing tools for modeling *sparse* tensor algebra accelerators do not fully overcome challenges arising from irregular sparsity. For example, STONNE [30, 31] supports only the SpMSpM kernel, and even then, only six pre-selected mappings for that kernel. Sparseloop [52] can model an accelerator describable in a single deep loop nest. As we will show, this is insufficient to express SIGMA, OuterSPACE, and Gamma. Additionally, Sparseloop uses abstract distribution functions to model sparsity, rather than precisely modeling the behavior of actual input sets. While better than using *just* shape-based information (like dense modeling), we show how this approach still results in degraded modeling accuracy (Section 7).

Our contribution. We provide a basis for specifying sparse tensor algebra accelerators by showing how recent designs can be expressed as *cascades (directed acyclic graphs or DAGs) of mapped Einsums* and *content-preserving transformations* on the constituent tensors in those Einsums. For example, both OuterSPACE and Gamma can be described by rewriting the Einsum for matrix multiply into several, dependent Einsums. In this abstraction, the sort/merge operations in those designs are described as *reordering the dimensions* of an intermediate tensor to improve locality, while the differences between the two operations are captured in how each Einsum is mapped. We show how the design choices of other accelerators can likewise be described in terms of a small set of categories of operations, e.g., splitting/combining dimensions of a tensor while preserving its contents.

Based on the above abstraction, we propose TeAAL (for *Tensor Algebra Accelerator Language*), a novel declarative language, simulator generator, and performance model that enables precise design specification and modeling of sparse tensor algebra accelerators. The TeAAL simulator generator takes accelerators described as mapped Einsum cascades and produces an imperative-style *intermediate representation (IR)* that describes tensor transformations as primitive operations on tensors represented as *fibertrees* [45]. It then uses implementation-level specifications (e.g., describing the tensor formats) to augment the IR to produce an accurate, validated performance model that processes real tensors.

Although attributes such as language expressivity and conciseness are difficult to quantify, as part of a study to validate TeAAL's fidelity, we write the TeAAL Einsum and mapping specifications of four recent (and disparate) sparse tensor algebra accelerator proposals (OuterSPACE [34], ExTensor [16], Gamma [55], and SIGMA [38]) in less than a page (see Figures 3 and 8), with each specification taking $\sim$ 30 lines. We verify that the models generated for each of these accelerators reproduce the designs' original published performance results (given the same input data sets) with high accuracy.

We view our primary contribution to be the accurate specification and modeling of sparse tensor algebra accelerators. That said, we also show how TeAAL can be applied in adjacent domains and be used to explore optimization opportunities for accelerators in those domains. Specifically, we use TeAAL to describe Graphicianado [14] and GraphDynS [53], which accelerate vertex-centric programming (a popular paradigm for graph algorithms), and demonstrate how to improve these designs by making point changes to their TeAAL specifications.

Taken as a whole, we feel TeAAL constitutes a significant advance over state-of-the-art practices in modeling and evaluating sparse tensor accelerators. Using TeAAL, one is able to describe a design using a precise, unified set of abstractions, supporting qualitative and quantitative comparison to other designs and enabling the exploration of the impact of a series of design changes. By contrast, standard practice today is to rely on English descriptions/figures and bespoke simulators, which suffer along all of these dimensions. Although prior works exist for exploring the space of sparse accelerator designs (e.g., Sparseloop [52]), they have more limited expressivity. For example, Sparseloop is only able to represent one of the six accelerators that we show results for.

To summarize, we make the following contributions:

- We show how modern sparse tensor algebra accelerator features can be represented using cascades of mapped Einsums and content-preserving transformations on those Einsums' constituent tensors. Based on this abstraction, we propose the TeAAL specification language for concisely and accurately specifying the design of sparse tensor algebra accelerators.
- We propose and design a simulator generator that transforms TeAAL specifications into an imperative-style IR that performs operations on fibertrees and lowers that IR to an accurate performance model of the specified design that processes real sparse tensor inputs.
- We validate TeAAL's accuracy in terms of modeling memory traffic, performance, and energy with respect to the reported results of four state-of-the-art accelerators.
- We demonstrate the potential of TeAAL as a tool for accelerator design by using it to speed up accelerators for vertex-centric programming—by 1.9× on BFS and 1.2× on SSSP over GraphDynS [53].

Beyond the artifact (Appendix A), we have made the source code for TeAAL available at https://github.com/FPSG-UIUC/teaal-compiler.

2 BACKGROUND AND MOTIVATION

We review key attributes of sparse tensor algebra and outline the design decisions typically made by sparse tensor accelerators. We highlight the difficulties with informal comparisons between accelerators and motivate the need for a precise, formal specification.

2.1 Tensors and Fibertrees

In this paper, an *N-tensor* is a multidimensional array with N dimensions. For example, a 0-tensor is a scalar, a 1-tensor is a vector,

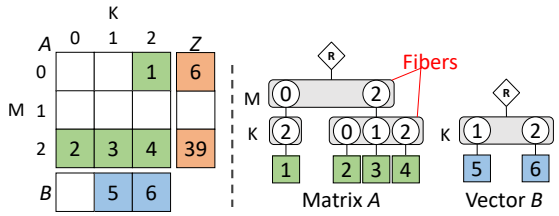


Figure 1: Sparse matrix-vector multiplication and corresponding fibertree representations.

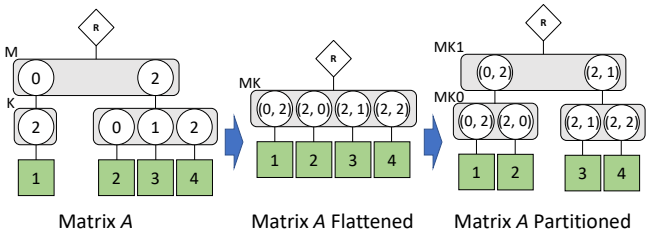


Figure 2: Flattening then partitioning ranks M, K of tensor A (Fig. 1).

and a 2-tensor is a matrix. Figure 1 shows a 2-tensor, A, with dimensions M and K. Using the terminology from Sze et al. [45], we describe a tensor's attributes:

- A *rank* refers to an axis/dimension in the tensor. A matrix has two ranks, often described as rows and columns.
- A *point* is a logical location within a tensor that contains a scalar *value*. A point is identified by an N-tuple of *coordinates* with one coordinate for each rank in the tensor. We denote the tensor A's element at point (m, k) as $A_{m,k}$.

Mathematically, tensors have no notion of sparsity or compression format (e.g., CSR). To avoid getting bogged down in the numerous details of various formats, we leverage the following abstractions proposed in Sze et al. [45]:

- A *fibertree* represents a tensor as a tree, with each level corresponding to a labeled rank in the tensor. Tensor A in Figure 1 has ranks M and K.
- The order of levels in a fibertree reflects its *rank order*, denoted $[M, K]$ in our example. The rank order list read left-to-write corresponds to the fibertree's ranks read top-to-bottom in the tree.
- Every level contains one or more *fibers*. A *fiber* is the set of elements sharing all coordinates in all higher levels of the tree. Fibers are more precise than "rows" or "columns," because they naturally extend to N-tensors.
- Each *element* in the fiber is a coordinate/payload pair, where the *payload* is a scalar value when it is at a leaf or a reference to a fiber when it is an intermediate node.
- The *shape of a fiber* is the the set of legal values the coordinates in that fiber can take on, where an integer shape means the open interval from zero to that integer. The *shape of a rank* is the union of the shapes of all fibers in that rank. The *shape of a tensor* is the list of shapes of each of the ranks in rank order.

One advantage of fibertrees is that they naturally handle both dense and sparse tensors (i.e., tensors where a number of points are zero). A dense tensor's fibertree has every coordinate present in the entire shape (i.e., a complete tree). On the other hand, a sparse tensor's fibertree can omit all elements with empty payloads (either zero values or empty fibers). The semantics of operations on fibers and fibertrees remain the same in both cases. Note that fibertrees are just an abstraction we use to describe operations on tensors. To model a specific design, all fibertrees are lowered to concrete representations, like CSR or COO (see Section 4.1.1). In this work, we use fibertrees both to categorize the space of design choices (Section 3) and as an IR during performance modeling (Section 4).

The fibertree abstraction also supports a number of transformations that change the fibertree corresponding to a tensor:

- A *rank flattening*, demonstrated in the first transformation in Figure 2, combines two ranks together into a single rank. After flattening, the coordinates are tuples of the coordinates in the original fibers that reference a payload from the original lower rank.
- A *rank partitioning*, demonstrated in the second transformation in Figure 2, separates a rank into two ranks. The coordinates of the new upper rank denote the first legal coordinate in the fiber below.
- A *rank swizzle* changes the fibertree's rank order (i.e., reorders the levels of the fibertree).

An important insight in our work is that many sparse accelerator behaviors can be viewed as one or more of these transformations (Section 3).

2.2 Tensor Algebra with Extended Einsums

TeAAL expresses the individual computations performed by an accelerator using equations written in an extended Einstein summation notation [13, 16, 45]. For simplicity, we call equations written in this form Einsums. Einsums are general enough to describe all tensor algebra kernels and have been used as the tensor algorithm specification in prior work for tensor algebra compilation [23, 50] and accelerator modeling [35, 45, 52].

We now present an operational definition of the Einsums used by TeAAL. An Einsum specifies three things: (1) the input and output tensors involved and their ranks, (2) an *iteration space* containing a point for each computation to be performed, and (3) the specific computation to be performed at each point in the iteration space. For example, the Einsum for matrix-vector multiply is:

$$Z_m = A_{m,k} \times B_k. \tag{1}$$

Here, the equation defines the input tensors (A, B), the output tensor (Z), and the iteration space—the Cartesian product of all legal coordinates in the expression—($M \times K$). An implementation of this Einsum must traverse each point in this space. For each point, it computes the operation ($\times$) on the right-hand-side using the specified points in the input operands (A, B). It then takes the result and populates the location specified in the left-hand-side (Z). Since the K rank does not appear in the output tensor, the Einsum will attempt to repeatedly populate the same point (Z_m). Einsum semantics resolve this by sequentially reducing the multiple values (using, in this case, addition) into a single value for that point. Note that the Einsum does not specify iteration order; this is left to the mapping (Section 2.3).

Table 1: Comparison of selected sparse tensor accelerator hardware proposals. TeAAL specifications increase both the precision and formalism of such comparisons, and enable automatic generation of performance/energy models.

Accelerator	Year	Mapping Approach	Architectural Focus
OuterSPACE [34]	2018	Outer Product parallelized across rows of A	Sparse matrix multiply with serial multiply/add phases, custom merge unit
ExTensor [16]	2019	Inner Product tiled across all dimensions for locality	Arbitrary Einsums and TACO formats [9], skip-ahead intersection unit
MatRaptor [42]	2020	Row-wise Product with parallel summation	Sparse matrix multiply with co-design of micro-architecture and C^2SR format
SIGMA [38]	2020	Inner Product parallelized across multiple dimensions	Sparse matrix multiply with custom bitmap format, flexible hardware topology
SpArch [56]	2020	Outer Product with parallel merge	Sparse matrix multiply with optimized RAM interface in sum phase
Tensaurus [43]	2020	Inner Product with extended scalar-fiber product followed by fiber-fiber product (SF^3)	SF^3 demonstrated applicability to multiple Einsums beyond matrix-matrix multiply
Gamma [55]	2021	Row-wise Product, adoption of Gustavson's alg.	Sparse matrix multiply with custom FiberCache, transposed merge-and-sum

By adding a new rank N to B, we can extend the above Einsum to matrix multiplication:

$$Z_{m,n} = A_{m,k} \times B_{k,n}. \tag{2}$$

This expands the iteration space to $M \times K \times N$.

As another example, our operational definition of an Einsum allows us to represent kernels beyond standard tensor algebra. For example, we can write 1D direct convolution with the Einsum:

$$O_q = I_{q+s} \times F_s \tag{3}$$

Just like the other examples, this equation defines the input tensors (I, F), output tensors (O), the iteration space ($Q \times S$), and the computation to occur at each point in the space. A reduction occurs across points in the iteration space with the same q and different s coordinates, as one would expect from 1D convolution.

2.3 Mapping Hardware Accelerators

Mapping [8] is the task of scheduling the computation of an Einsum onto limited hardware resources to jointly optimize for the desired combination of throughput, latency (execution time), power, etc. We summarize the mapping attributes used for hardware modeling and design in prior work [7, 17, 19, 25, 29, 35, 54] that we use throughout.

1) Loop order. The Einsum's large iteration space must be serialized through finite datapath resources in some order. Two choices for Equation 1 are: (1) $[M, K]$ or (2) $[K, M]$. Loop order is read left-to-right, corresponding to the topmost-to-bottommost loop in a loop nest. For example, (1) above reads "for each value of m, iterate through all values of k." This choice affects data locality and, in turn, memory access costs. Depending on on-chip buffer sizes, loop order (1) for matrix-vector multiply keeps an element of Z *stationary* [8] in on-chip memory while B is streamed in multiple times. Meanwhile, loop order (2) keeps B stationary but repeatedly streams Z.

2) Splitting. To further improve data locality across all tensors, many algorithms employ splitting (or strip mining, blocking, etc.) to divide the iteration space into subspaces that refer to a small enough subset of each of the tensors that they fit fully in on-chip buffers. Fibertrees model these subsets by partitioning their fibers according to the split iteration space. How a fiber is partitioned is a function of the coordinates in the fiber. For example, suppose matrix A has a rank-order of $[M, K]$. Splitting K by shape $K0$ results in a new A tensor with rank-order $[M, K1, K0]$, where K is split into $K1$ partitions with $K0$ coordinates each.[1]

[1] In other words, each tile stores coordinates in the coordinate range $[i{*}K0, (i{+}1){*}K0)$ for some i.

3) Work scheduling. Finally, the mapping specifies how the iteration space is traversed, by placing each point at a specific location in both time and space. Mapping a computation at different locations in time implies that the computation is serial (i.e., computations happen one after another on the same component), while mapping at different locations in space implies parallelism (i.e., computation happens at the same time on different processing elements (PEs)).

2.4 Accelerating Sparse Tensor Algebra

Sparse tensor algebra introduces new opportunities and challenges to the mapping problem. Sparse tensors are typically *compressed* to remove the zero elements, resulting in fibertrees with missing coordinate-payload pairs. Compression can yield significant savings in storage and data transfers and avoid ineffectual compute—operations that have no impact on the result and can be safely skipped, e.g., multiplication or addition with zero [16].

However, realizing these benefits requires accelerators to "sparsify" the iteration space, or remove the ineffectual compute, increasing design complexity, sensitivity to memory latency/bandwidth, and load imbalance. For example, the $[M, K]$ loop order for Equation 1 may *skip*, for example, from $(m = 0, k = 2)$ to $(m = 2, k = 0)$ in one step [52]. Such skipping can remove ineffectual compute but may require *co-iteration* of the operands and additional operations (e.g., intersection for fibers multiplied together). Without careful engineering, this can lead to inefficiencies that do not occur when tensors are dense [33]. For example, the same-shape tiles produced by the scheme described in Section 2.3 may have different memory footprints, leading to data transfer and compute load imbalance when tiles are distributed to workers.

To deal with these challenges, sparse tensor accelerators have proposed a wide variety of custom hardware solutions, summarized in Table 1. We note that the complexity of this topic makes such a table an imprecise and ultimately unsatisfying comparison. Additionally, all of these works used custom hand-written simulators run on actual data sets to ensure all complexities are captured. In the remainder of this paper we present a formalism to resolve this imprecision and enable concise apples-to-apples comparison.

3 OVERVIEW AND INSIGHTS

We now propose TeAAL: a language and simulator generator that 1) enables the concise specification of a sparse tensor algebra accelerator and 2) generates efficiency statistics for that accelerator computing on actual sparse tensors.

Our key conceptual contribution, which guides the design of TeAAL, is to show that recent sparse tensor algebra accelerators can

be expressed as *cascades* (directed, acyclic graphs; DAGs) of mapped Einsums (Sections 2.2, 2.3) and *content-preserving transformations* on their tensors. This can be elaborated as two novel insights:

Insight 1: Einsum cascades can represent multi-phase accelerator designs (Section 3.1). A variety of accelerators and algorithms targeting seemingly monolithic kernels are more accurately and succinctly described as a sequence of distinct, interconnected phases. We show that cascades of Einsums are sufficiently expressive to represent these multi-phase computations (e.g., Toeplitz-based convolution, OuterSPACE's multiply-merge, SIGMA's pre-filtering, Graphicionado's process and apply).

Insight 2: Content-preserving transformations on tensors, representable as core operations on fibertrees, capture idioms for sparse tensor data orchestration (Section 3.2). A variety of sparse accelerator behaviors (e.g., work scheduling, splitting, sorting/merging) can be represented as content-preserving transformations on *specific* tensors in the Einsum cascade. We show how these transformations can be represented as a small set of core operations performed on fibertrees (Section 2.1). In specific, we use fibertree rank partitioning/flattening as a general pattern for representing both sparse tensor splitting and work scheduling strategies. Additionally, we use fibertree rank swizzling as a general pattern for sorting and merging, which is often used to improve tensor traversal efficiency.

This section describes the above insights in more detail and how they enable the design of the TeAAL specification language. Section 4 describes how the TeAAL simulator generator converts TeAAL specifications into an imperative-style IR describing operations on fibertrees and how this IR (augmented with some additional information, e.g., describing the architecture and concrete formats) is subsequently converted into an accurate performance model.

TeAAL specifications. The TeAAL *specification language* is a declarative, domain-specific language (DSL) that defines the computation as a cascade of Einsums (*expressions*), attributes on each tensor (*declaration, rank-order, partitioning*), and a dataflow that describes when and where those tensors' data is moved (*loop-order, spacetime*). We refer to the tensor declarations and Einsums as the *einsum specification* and the tensor and dataflow attributes as the *mapping specification*. Rank swizzling is not expressed explicitly, but is inferred from other mapping attributes, such as the *rank-order* and *loop-order*.

OuterSPACE running example. Throughout this section and Section 4, we use the example TeAAL specification in Figure 3, which describes the OuterSPACE accelerator [34]. At a high level, OuterSPACE accelerates SpMSpM using the *multiply-merge* algorithm. It first performs all multiplications between input tensors A and B in an outer-product fashion, writes the resulting partial products to an array-of-linked-lists data structure, sorts the linked lists to facilitate reduction, and finally performs reductions over the now-sorted lists to derive final results. Throughout the rest of the section, we will discuss how TeAAL both implicitly and explicitly captures these behaviors.

```
einsum:
  declaration:   # Ranks are listed alphabetically in this section
    A: [K, M]    # Rank order is specified below in rank–order
    B: [K, N]
    T: [K, M, N]
    Z: [M, N]
  expressions:
    – T[k, m, n] = A[k, m] * B[k, n] # T_{k,m,n} = A_{k,m} × B_{k,n}
    – Z[m, n] = T[k, m, n]  # Z_{m,n} = T_{k,m,n}
mapping:
  rank–order:
    A: [K, M]
    B: [K, N]
    T: [M, K, N]
    Z: [M, N]
  partitioning:
    T:
      (K, M): [flatten()]
      KM: [uniform_occupancy(A.256), uniform_occupancy(A.16)]
    Z:
      M: [uniform_occupancy(T.128), uniform_occupancy(T.8)]
  loop–order:
    T: [KM2, KM1, KM0, N]
    Z: [M2, M1, M0, N, K]
  spacetime:
    T:
      space: [KM1, KM0]
      time: [KM2, N]
    Z:
      space: [M1, M0]
      time: [M2, N, K]
```

Figure 3: TeAAL specification for the Einsums and mappings of OuterSPACE [34], described in detail in Section 3.

Table 2: Cascades of Einsums for various accelerators and algorithms. Also see Figure 12 for the Einsum cascades describing Graphicionado [14] and GraphDynS [53].

Accelerators / Algorithms	Cascade
ExTensor [16]'s SpMSpM	$Z_{m,n} = A_{k,m} \times B_{k,n}$
Gamma [55]'s SpMSpM	$T_{k,m,n} = take(A_{k,m}, B_{k,n}, 1)$ $Z_{m,n} = A_{k,m} \times T_{k,m,n}$
OuterSPACE [34]'s SpMSpM	$T_{k,m,n} = A_{k,m} \times B_{k,n}$ $Z_{m,n} = T_{k,m,n}$
SIGMA [38]'s SpMSpM	$S_{k,m} = take(A_{k,m}, B_{k,n}, 0)$ $T_{k,m} = take(A_{k,m}, S_{k,m}, 0)$ $Z_{m,n} = T_{k,m} \times B_{k,n}$
Eyeriss [8]'s CONV	$O_{b,m,p,q} = I_{b,c,p+r,q+s} \times F_{c,m,r,s}$
Toeplitz expansion/im2col + CONV [45]	$T_{b,c,p,q,r,s} = I_{b,c,p+r,q+s}$ $O_{b,m,p,q} = T_{b,c,p,q,r,s} \times F_{c,m,r,s}$
Tensaurus [43]'s MTTKRP	$C_{i,r} = T_{i,j,k} \times B_{j,r} \times A_{k,r}$
Factorized MTTRKP [48]	$S_{i,j,r} = T_{i,j,k} \times A_{k,r}$ $C_{i,r} = S_{i,j,r} \times B_{j,r}$
Cooley-Tukey FFT Step [10]	$E_{0,k0} = P_{0,k0,n1,0} \times X_{n1,0}$ $O_{0,k0} = P_{0,k0,n1,0} \times X_{n1,1}$ $T_{k0} = P_{0,k0,0,1} \times O_{0,k0}$ $Y_{0,k0} = E_{0,k0} + T_{k0}$ $Y_{1,k0} = E_{0,k0} - T_{k0}$

3.1 Insight 1: Einsum cascades capture multi-phase accelerators

Our first insight is that seemingly monolithic tensor algebra kernels (e.g., matrix multiply) are often implemented as a DAG of operations, and that each of these operations can be expressed as an Einsum that produces and consumes intermediate tensors. We call this DAG a *cascade*. For example, consider a 1D convolution between input I and filter F. Convolution is performed using two predominant implementation styles. The first style is direct convolution, which is often employed by accelerators. As an Einsum, direct convolution is written as

$$O_q = I_{q+s} \times F_s \tag{4}$$

An alternative style is the Toeplitz expansion [45], which converts the convolution into a matrix-vector or matrix-matrix multiply and is common on systolic arrays and data-parallel processors like GPUs. First, the input is refactored into a matrix to enable, in this case, matrix-vector multiplication between the input (now stored in T) and the filter F in the second stage. An important observation is that this can be written as the following sequence of dependent Einsums:

$$T_{q,s} = I_{q+s}; \quad O_q = T_{q,s} \times F_s \tag{5}$$

Importantly, the RHS of the Einsum used to generate T mirrors how I is indexed in the Einsum for direct convolution. The Toeplitz expansion simply relaxes the requirement that the access into I and the corresponding access into F happen at the same time. Decomposing an Einsum into a cascade enables each resulting Einsum to be mapped independently, exposing new degrees of freedom for building and using intermediate tensors. Note that this sequence of Einsums says nothing about how (if at all) the two stages are overlapped. They can happen entirely sequentially, or the accelerator can implement pipeline parallelism. For example, the Q rank can be partitioned (Section 3.2.1) and once a partition of T is produced, it can be consumed by the multiply stage. Section 4.3 describes how TeAAL determines this parallelism.

Beyond convolution, cascades of Einsums can be used to represent other common implementation styles in sparse tensor algebra accelerators. For example, they capture the multiply-merge algorithm in the OuterSPACE example (Figure 3). During the multiply phase (Line 8), columns of the A matrix are multiplied with rows of the B matrix to form partial products, which we call T. Then, during the merge phase, specified by the second Einsum (Line 9), T is reduced along the K rank, yielding the final result Z.

Sparsity also motivates new operations on tensors, and our Einsum notation can be extended to include them if needed. We currently support one—the take(.) operator—which decouples intersection from computation with the following semantics: if at least one of the inputs is zero at a point, the output is zero, otherwise, copy one of the inputs into the output. Take the example:

$$T_{k,m,n} = take(A_{k,m}, B_{k,n}, 1) \tag{6}$$

The final parameter denotes which input is copied into the output. This example copies B into T, but if the last parameter were 0, A would be copied.

Beyond the above examples, Table 2 shows a variety of accelerators and algorithms represented as cascades of Einsums.

3.2 Insight 2: Content-preserving transformations on fibertrees capture accelerator data-orchestration strategies

Our second insight is that a variety of accelerator behaviors (e.g., work scheduling) are describable as what we call *content-preserving transformations* applied to *specific* tensors in the Einsum cascade. We say a tensor transformation is content-preserving if it does not change the content of the tensor, i.e., the set of values at the leaves of the fibertree, but does change the coordinate system used to access each value. Such transformations may also impact the tensor's data layout when lowered to a concrete representation.

We make an important observation that the core operations performed on fibertrees (Section 2.1) represent a set of content-preserving transformations that are useful for describing a variety of prevalent sparse data orchestration strategies. Specifically, fibertree rank partitioning (and its inverse: flattening) can be used as a single abstraction for specifying both sparse tensor splitting and work scheduling strategies. Similarly, fibertree rank swizzling can be used as a single abstraction for specifying transposing data in memory, sorting, and merging.

3.2.1 Sparse Tensor Splitting and Work Scheduling. Recall from Section 2.3 that splitting for dense problems is shape-based. This can be expressed by partitioning a fibertree rank at coordinate-based boundaries given by the tile dimension. Unfortunately, when data is sparse, this strategy can lead to low reuse and under-utilization of tiles (and therefore buffers) [33], i.e., if different partitions have different occupancies.

We make an important observation that partitioning naturally generalizes to other types of splitting that *can* adapt to irregular sparsity, simply by changing the *partitioning criteria*, i.e., where the partition boundaries occur. From studying existing accelerators, we define a simple sparsity-aware strategy we call *uniform occupancy-based partitioning*. In this scheme, each fiber at a level in the fibertree is split so that each new fiber has an equal number of elements (modulo remainders). Importantly, each fiber's coordinate range after an occupancy-based partitioning is irregular. Thus, to ensure that partitions of multiple tensors have matching coordinate ranges for co-iteration (Section 2.4), occupancy-based partitioning uses a *leader-follower* paradigm: the partitions' coordinate ranges are chosen so that the leader tensor's partitions are equal occupancy and all follower tensors adopt those ranges [52].

Unfortunately, uniform occupancy-based partitioning may still result in partitions with varying occupancies because a partition must end where its parent fiber ends. Flattening (Section 2.1), when combined with occupancy-based partitioning, mitigates this imbalance by first combining the flattened ranks, then redistributing the elements so that, globally, each partition has the same number of values. For example, Figure 2 shows how a fibertree whose fibers start with an unequal number of coordinates can be flattened and re-partitioned to equalize the number of coordinates per partition. Note that, though all partitioning directives modify the fibertree abstract representation, the concrete representation may remain unchanged.

The above describes how tensor data can be efficiently split in the presence of sparsity. More subtly, we observe that partitioning

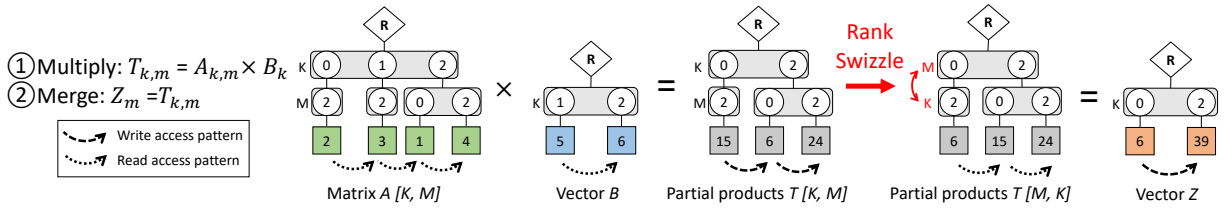


Figure 4: Rank swizzling in sparse tensor algebra computations, using outer-product multiply-merge matrix-vector multiplication. Matrix A and vector B use values from Figure 1 for consistency. An offline rank swap ensures that A has rank order $[K, M]$ prior to the multiply phase, and an online rank swap ensures that T has rank order $[M, K]$ prior to the merge phase, ensuring concordant traversal in both phases.

and flattening are also useful abstractions through which to specify *work scheduling* when work is parallelized. Consider OuterSPACE, which works on 256 non-empty elements of matrix A at a time during the multiply phase, further subdividing these into 16 groups of 16 elements to each be processed by a "Processing Tile" (a group of PEs [34]). The TeAAL specification for OuterSPACE (Figure 3) represents this as a flattening (Line 18) and then an occupancy-based partitioning applied twice hierarchically (Line 19).[2] The TeAAL specification describes the *parallelism* this partitioning enables on Line 27 by scheduling ranks $KM1$ and $KM0$ in space (Section 2.3).

3.2.2 Transposition, Sorting, and Merging. We observe that sparse tensor algebra accelerators employ a number of techniques that, when expressing their tensors in the fibertree abstraction, are tantamount to fibertree rank swizzles. These operations enable the more efficient *concordant* (as opposed to *discordant*) traversal [45]. Concordant traversal occurs when a loop nest traverses a fibertree in the order in which its ranks appear, i.e., traverses each fiber sequentially and in a depth-first manner. For example, in Figure 4, A is traversed concordantly, since it has a $[K, M]$ rank order, and the multiply phase has a $[K, M]$ loop order. Thus, we never have to search for the next k or m coordinate; it is always the first or next coordinate in the current fiber.[3] Conversely, iterating over K in the bottom-most loop would be a discordant traversal (for this rank order). In OuterSPACE (Figure 3), despite all of the partitioning, during the multiply phase both A and B are traversed concordantly. K is traversed sequentially and most slowly, then M and then N.

It is common practice to swizzle ranks to enable concordant traversal on input tensors. For example, the transposition of a matrix from the CSR format into the CSC format can be viewed as a rank swizzle and is used by OuterSPACE to achieve a $[K, M]$ rank order on A (Line 12) in preparation for the outer-product-style multiply phase. Input tensor swizzles are usually performed offline.

More subtly, we observe that sparse tensor accelerators also perform rank swizzles on *intermediate tensors* formed during kernels expressed as cascades of Einsums (Section 3.1). Depending on whether coordinates in the intermediate tensors are stored in sorted or unsorted order and on the extent to which the intermediate tensors are built before being consumed, this either requires a merge or a (more expensive) sort operation.

[2]Note: OuterSPACE only enables half its PEs during the merge step, so the occupancy-based partitioning applied to the second Einsum (Line 21) only involves 128 PEs (8 per "Processing Tile").

[3]Though many concrete representations enable efficient sequential iteration through fibers, some do not. The true cost of iteration is accounted for during modeling (Section 4).

Table 3: Supported hardware components and their attributes.

Component	Attributes
DRAM	bandwidth
Buffer	type (buffet [37] or cache), width, depth, bandwidth
Intersection	type (two-finger, leader-follower, or skip-ahead), leader
Merger	inputs, comparator_radix, outputs, order (fifo, opt), reduce
Sequencer	num_ranks
Compute	type (mul or add)

Figure 4 shows an example of a multiply-merge for outer product, matrix-vector multiplication. To support concordant traversal on both input and output tensors, the multiply phase uses a $[K, M]$ loop order, while the merge phase uses an $[M, K]$ loop order. Thus, at the end of the multiply phase, T has rank order $[K, M]$. Then, during the reduction, a rank swizzle changes the rank order of T to $[M, K]$ to match the $[M, K]$ loop order. The dashed arrows in Figure 4 show that *both* the tensor read and write access patterns through A, B, T, and Z are all concordant through both phases. Importantly, such online rank swizzles may degrade performance and, therefore, warrant dedicated hardware support. Yet, they can significantly improve spatial/temporal locality, and thus, appear in multiple prior designs [34, 55, 56].

By default, TeAAL infers rank swizzling automatically to maintain concordant traversal. For example, for OuterSPACE (Figure 3), T has rank order $[M, K, N]$, but TeAAL produces T during the multiply phase in $[K, M, N]$ order, swizzles it to $[M, K, N]$ order to be stored in memory, and then swizzles it again to $[M, N, K]$ order to prepare for the merge.

4 GENERATING THE MODEL

In Section 3, we showed that the fibertree abstraction is general enough to describe many of the design decisions used in sparse tensor algebra accelerators. However, to manifest a specific design, the fibertrees must be lowered onto concrete representations and their operations bound to specific hardware components. In this section, we define three additional specifications—*format*, *architecture*, and *binding*—used by TeAAL to perform this lowering and describe how these, plus the *einsum* and *mapping* specifications from Section 3, are combined to produce an executable model for evaluating accelerator workload performance.

4.1 Lowering Mapped Einsums to Hardware

This section describes the three additional specifications (*format*, *architecture*, and *binding*) that are used to lower mapped Einsums to concrete representations and hardware resources.

4.1.1 Format. Prior works on modeling sparse tensor algebra computations [31, 52] extend TACO's level formats concept [9], which

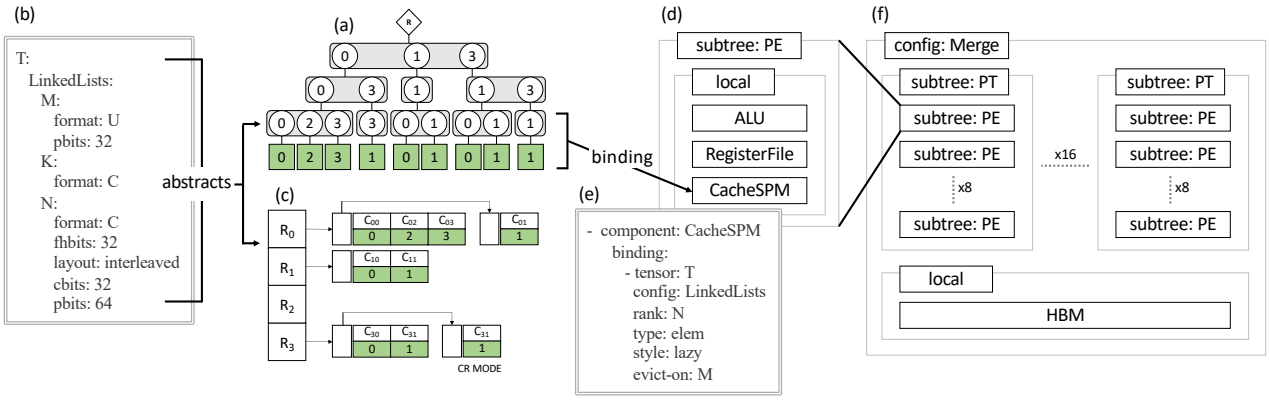


Figure 5: TeAAL concrete/hardware-level model of the OuterSPACE accelerator [34]. The fibertree (a) combined with the format specification (b) describe the concrete representation, a custom array-of-linked-lists format (c). TeAAL specifies the architecture hierarchically (f), where each level has a set of local components (d) that have tensor operations bound to them (e). More details are given in Section 4.2.

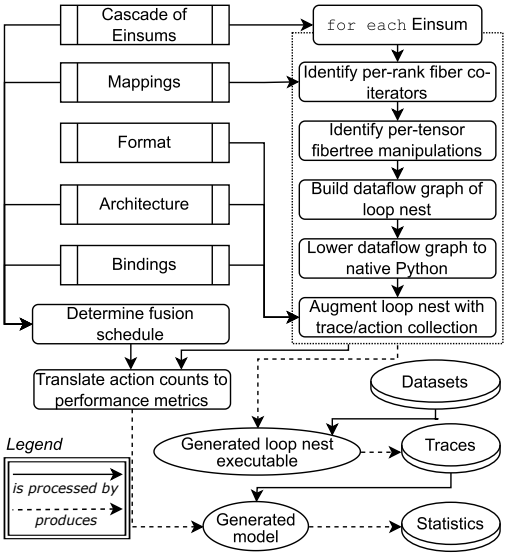


Figure 6: TeAAL tool flow diagram, described in detail in Section 4.3.

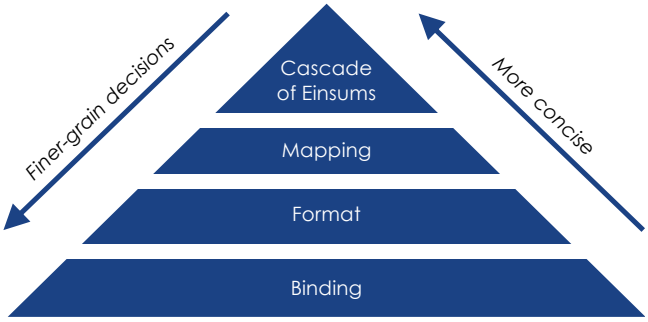


Figure 7: Separation of concerns enabled by TeAAL. We consider the architecture to be fixed during map space exploration and, therefore, outside the separation of concerns.

facilitates a convenient abstraction for translating a fibertree to its concrete representation by specifying a per-fiber format as described in [45]. However, the formats used by existing sparse accelerator modeling frameworks restrict themselves to a fixed number of common configurations (e.g., bitmap [31] or uncompressed offset pointers [52]).

TeAAL extends these concepts with a more modular specification, capturing a larger class of formats than existing tools, by separating the attributes of a fiber's format into three categories: a format type, a layout, and data widths for the coordinates (*cbits*), payloads (*pbits*), and fiber headers ($fhbits$). Fibers are concretized as an array of coordinates and a array of payloads (struct-of-arrays) or as a single array of elements (array-of-structs). TeAAL supports three format types: uncompressed (U)—where the sizes of the data arrays correspond to the shape of the fiber, compressed (C)—where the sizes of the data arrays correspond to the occupancy of the fiber, or a combination (B)—where the coordinates are uncompressed and the payloads are compressed. For simplicity, currently, all fibers in a rank have the same format. Note that not all information in the fibertree is stored explicitly, e.g., an uncompressed fiber does not need to store coordinates because they can be inferred from the position of the payloads. TeAAL supports this by allowing the corresponding data width—in this case, *cbits*—to be unspecified or set to 0. This specification is flexible enough to support a variety of common formats (e.g., uncompressed arrays, CSR, run-length encoding) and custom formats even beyond those supported by TACO (e.g., from OuterSPACE [34] and SIGMA [38]). The specification also allows each tensor to be associated with multiple formats (differentiated by the configuration name), since manipulating the fibertree may cause the representation to change dynamically. Finally, while there are formats that require functionality outside the above framework (e.g., TACO's hashed format, which requires a hash function be specified), we feel these could be added with minimal changes to the both other specifications and simulator generator.

4.1.2 Architecture. The TeAAL architecture specification (inspired by Timeloop [35]) describes the accelerator topology as a tree of compute and storage units. At each level of the hierarchy, one can

define a list of hardware components local to that level and a list of subtrees below that level. In addition to the component classes supported by Timeloop, we define new classes of components that are involved in performance-limiting operations on sparse accelerators, including caches, intersection units, and hardware mergers. Table 3 gives the full list of supported classes and their attributes. Since an accelerator (e.g., OuterSPACE [34]) may reorganize itself during the execution of a single computation, TeAAL also supports specifying multiple topologies for the same design.

4.1.3 Binding. Finally, TeAAL's binding specification matches the Einsum- and mapping-induced fibertree operations to specific concrete representations and hardware components in the architecture. First, each Einsum must be bound to a single accelerator topology. Then, for each storage component, its bindings describe what data resides there. Each binding contains the data's tensor, configuration, rank, type (e.g., payload), whether elements are accessed lazily (loading/storing only the element on access) or eagerly (loading/storing the entire subtree below an element on access), and sometimes (e.g., for buffets [37]), how long the data is buffered. A storage component can have multiple such bindings. Similarly, for each compute component, the binding describes which compute operations are performed on that component.

4.1.4 TeAAL's Expressibility and Extensibility. Putting everything together: TeAAL decomposes the design of an accelerator into a set of categories of abstractions, where a specific design choice is an instance of the category. As shown in Figure 7, these abstractions are hierarchical; the accelerator's cascade of Einsums is the most concise representation of that accelerator's design, while its binding encodes the finest-grain design decisions enabling the highest-fidelity modeling.

This separation of concerns enables TeAAL to express a large number of accelerators, and facilitates the process of adding new features to represent yet other accelerators. For example, TeAAL can express accelerators that differ only in specific details (such as tensor format [23] or cache replacement policy [55]) simply by changing that part of the specification, leaving all else equal. New features can likewise be added by augmenting the relevant abstraction category, again leaving other categories unmodified.

4.2 Specifying OuterSPACE [34]

We continue to use OuterSPACE as a running example to motivate the features provided by TeAAL. Figure 5 shows a simplified version of OuterSPACE's custom tensor format, its architecture during the merge phase, and the correspondence between the fibertree representation of T and its concrete representation and binding. In Figure 5a, we see the fibertree for a concrete example tensor T. The format specification (Figure 5b) for this tensor lowers it onto OuterSPACE's custom array-of-linked-lists format (Figure 5c). To differentiate it from other representations of the same tensor, we give it the configuration name *LinkedLists*. On the M rank, the array of pointers is given by an uncompressed (U) array of payloads. On the N rank, the fiber header data width ($fhbits$) describes the linked list pointers, the *layout* describes that corresponding coordinates and payloads are adjacent (array-of-structs), and the $cbits$ and $pbits$ describe the data widths of the coordinates and payloads, respectively. Figure 5d shows an OuterSPACE PE. During the merge phase, this level has three components: the ALU, the register file, and the L0 scratchpad. OuterSPACE loads the entire subtree under a given M coordinate into the L0 scratchpad to perform its sort. TeAAL expresses this binding with the specification given in Figure 5e. The $tensor$, $config$, $rank$, and $type$ denote exactly what data is buffered, and the $evict-on$ keyword is required for binding to explicitly managed buffers, whose fill and drain policy must be set by the user. Because the elements bound to this buffer evict on M, old data is drained when the M coordinate changes. Finally, Figure 5f shows an overview of the entire accelerator topology.

4.3 Simulator Generation

Figure 6 demonstrates how TeAAL puts everything together. For each Einsum in the cascade, TeAAL combines the Einsum equation with its mapping information to produce an executable loop nest. To do so, it identifies the necessary per-tensor fibertree manipulations (e.g., rank swizzling) and per-rank fiber co-iterators (e.g., intersection). TeAAL then uses this information to build a dataflow graph of a loop nest, which it then lowers to an embedded DSL within Python for executing computations as fibertree operations [1]. The resulting code is an imperative-style representation of the Einsum cascade (i.e., a series of loop nests, one per Einsum), which can directly evaluate real tensors represented as fibertrees. TeAAL then breaks the modeling of an accelerator into three stages: generate *traces* describing when each coordinate and each payload is accessed, calculate the action counts for each component from the traces, and combine the action counts from all components to produce summary statistics like execution time and energy.

Trace generation. TeAAL combines information from the format, architecture, and binding to identify which traces need to be collected in preparation for performance modeling. It then instruments the mapped loop nests to collect the desired traces. When executed, the mapped loop nests perform the computation on fibertrees (storing real tensor data) and generate a trace of when each coordinate and each payload is accessed. Therefore, unlike an analytical model, TeAAL is able to fully capture the impact of each real tensor's specific sparsity patterns on the kernel's performance, significantly improving TeAAL's fidelity over that of analytical models. We quantitatively explore this phenomenon in Section 7 and Figure 10a.

Trace consumption. TeAAL provides a library of per-component action count models (see Table 3 for a full list). It inserts calls to these component models after the loop nest, passing information about their specific attributes (e.g., buffer width and depth) and a list of traces to be read. During the evaluation of the model, each component uses the traces generated to produce the action counts it performed.

Action count consumption. TeAAL uses Accelergy [51] to translate action counts to energy use and a custom analytical modeling/bottleneck analysis to translate the action counts to execution time. To compute execution time, TeAAL must first determine the Einsum *blocks*, or sets of Einsums that are fused together. Fusion occurs when Einsums communicate by sharing sub-tensors with each other (instead of entire tensors). The full cascade of Einsums,

```
einsum:
 declaration:
  A: [K, M]
  B: [K, N]
  T: [K, M, N]
  Z: [M, N]
 expressions:
  - T[k,m,n] = take(A[k,m], B[k,n], 1)
  - Z[m,n] = T[k,m,n] * A[k,m]
mapping:
 rank-order:
  A: [M, K]
  B: [K, N]
  T: [M, K, N]
  Z: [M, N]
 partitioning:
  T:
   M: [uniform_occupancy(A.32)]
   K: [uniform_occupancy(A.64)]
  Z:
   M: [uniform_occupancy(A.32)]
   K: [uniform_occupancy(A.64)]
 loop-order:
  T: [M1, M0, K1, K0, N]
  Z: [M1, M0, K1, N, K0]
 spacetime:
  T:
   space: [M0, K1]
   time: [M1, K0, N]
  Z:
   space: [M0, K1]
   time: [M1, N, K0]
```

(a) Gamma accelerator [55].

```
einsum:
 declaration:
  A: [K, M]
  B: [K, N]
  Z: [M, N]
 expressions:
  - Z[m,n] = A[k,m] * B[k,n]
mapping:
 rank-order:
  A: [K, M]
  B: [K, N]
  Z: [M, N]
 partitioning:
  Z:
   K:
    - uniform_shape(K1)
    - uniform_shape(K0)
   M:
    - uniform_shape(M1)
    - uniform_shape(M0)
   N:
    - uniform_shape(N1)
    - uniform_shape(N0)
 loop-order:
  Z: [N2, K2, M2, M1, N1, K1, M0, N0, K0]
 spacetime:
  Z:
   space: [K1]
   time: [N2, K2, M2, M1, N1, M0, N0, K0]
```

(b) ExTensor accelerator [16].

```
einsum:
 declaration:
  A: [K, M]
  B: [K, N]
  S: [K, M]
  T: [K, M]
  Z: [M, N]
 expressions:
  - S[k, m] = take(A[k, m], B[k, n], 0)
  - T[k, m] = take(A[k, m], S[k, m], 0)
  - Z[m, n] = T[k, m] * B[k, n]
mapping:
 rank-order:
  A: [K, M]
  B: [K, N]
  S: [K, M]
  T: [K, M]
  Z: [M, N]
 partitioning:
  Z:
   K: [uniform_shape(128)]
   (M, K0): [flatten()]
   MK0: [uniform_occupancy(T.16384)]
 loop-order:
  S: [K, M, N]
  T: [K, M]
  Z: [K1, MK01, MK00, N]
 spacetime:
  S:
   space: []
   time: [K, M, N]
  T:
   space: []
   time: [K, M]
  Z:
   space: [MK00]
   time: [K1, MK01, N.coord]
```

(c) SIGMA accelerator [38].

Figure 8: State-of-the-art sparse tensor accelerators. uniform_shape()/flatten() **are syntax for shape-based partitioning/flattening (Section 3.2.1).**

mappings, architectures, and bindings are used to determine the Einsum blocks. Specifically, TeAAL infers that Einsums can be fused together when three conditions are met:[4]

- The Einsums use the same accelerator configuration.
- The temporal ranks in all loop orders before the first spatial rank are the same.
- Disjoint subsets of the non-storage components are each exclusively used by only one Einsum.

As a simple heuristic, TeAAL starts at the first Einsum and greedily fuses the successive Einsums together into a single block, until it cannot do so any more. At which point, it starts a new block. TeAAL sums together the action counts for each component performed by each block and then computes per-block, per-component execution times. It then applies a bottleneck analysis: the execution time of the block is the execution time of the longest component, and the execution time of the cascade is the sum of the execution times of all of the blocks.

[4]These conditions for inferring fusion are not fundamental and can be changed if needed.

5 ACCELERATOR SPECIFICATION

Sections 3-4 used OuterSPACE [34] as a running example. We now describe the Einsums and mapping specifications for three other, state-of-the-art accelerators relevant to our evaluation, shown in Figure 8. We have modeled other accelerators that we omit for space, including Graphicionado [14] and GraphDynS [53] (Section 8), Eyeriss [8], Tensaurus [43], Flexagon [30], and DSTC [47]. We also omit the format, architecture, and binding specifications for brevity.

Gamma [55]. Gamma (Figure 8a) is a row-wise-style accelerator that uses a tightly-pipelined multiply-merge-style architecture to reduce partial output traffic and enable concordant traversal across both input and output tensors. In Gamma's dataflow, a row of A is combined and reduced with rows of B. Gamma distributes rows of A

Table 4: Tensor data set characteristics. The top 5 tensors are used in our validation study (Section 7); the bottom 3 in our new design study (Section 8).

Matrix	Shape	NNZ	Domain
wiki-Vote (wi)	$8.3K \times 8.3K$	104K	elections
p2p-Gnutella31 (p2)	$63K \times 63K$	148K	file-sharing
ca-CondMat (ca)	$23K \times 23K$	187K	collab. net.
poisson3Da (po)	$14K \times 23K$	353K	fluid dynamics
email-Enron (em)	$37K \times 37K$	368K	email comms.
flickr (fl)	$0.82M \times 0.82M$	9.8M	site crawl graph
wikipedia-20070206 (wk)	$3.6M \times 3.6M$	42M	site link graph
soc-LiveJournal1 (lj)	$4.8M \times 4.8M$	69M	follower graph

Table 5: Hardware configs, chosen to match original publications.

ExTensor [16]	1 GHz clock speed, 128 PEs, 64 kB PE buffer per PE, 30 MB LLC, 68.256 GB/s memory bandwidth
Gamma [55]	1 GHz clock speed, 64-way merger per PE, 32 PEs, 3 MB FiberCache, 16 64-bit HBM channels, 8 GB/s/channel
OuterSPACE [34]	1.5 GHz clock speed, 16 PEs per PT, 16 PTs, 16 kB L0 cache per PT, 4 kB L1 cache per 4 PTs, 16 64-bit HBM channels, 8000 MB/s/channel
SIGMA [38]	500 MHz clock speed, 128 PEs per FlexDPE, 128 FlexDPEs, 32 MB Data SRAM, 4 MB Bitmap SRAM, 960 GB/s SRAM bandwidth, 1024 GB/s HBM bandwidth
Graphicionado [14]	1 GHz clock speed, 8 streams, 64MB eDRAM, 68 GB/s memory bandwidth

to each PE and, based on which values in each row are non-zero, the PE fetches a subset of the rows of B. This filtering is implemented using the take(.) operator (Section 3.1). After being fetched to each PE, the rows of B (which initially have rank order $[K, N]$) are sorted with hardware mergers to facilitate reduction over K. Similar to OuterSPACE, this is expressed as a rank swizzle: T has rank order $[M, K, N]$ and the rightmost (bottommost) rank in the loop order for the Einsum computing Z is K. Hence, TeAAL inserts a rank swizzle on T, making its rank order $[M, N, K]$ in the context of the second Einsum. Unlike OuterSPACE, the two Einsums in the cascade are fused together, per the criteria described in Section 4.3.

ExTensor [16]. ExTensor (Figure 8b) employs a hybrid dataflow that is inner product-style at the innermost level. ExTensor's two salient characteristics are the use of uniform shape-based partitioning (Section 2.3) and hierarchical intersection. Lines 14-23 describe this partitioning, while hierarchical intersection is accounted for implicitly due to fibertree semantics (Section 2.4). Note that our ExTensor specification includes details beyond the original paper from private correspondence with the authors about the actual design of the simulator used for evaluation.

SIGMA [38]. SIGMA (Figure 8c) is a deep-learning accelerator that uses occupancy-based partitioning to only distribute non-zero elements of the stationary matrix to PEs, reducing ineffectual compute. While SIGMA can be configured to support A and B-stationary dataflows, we only describe/evaluate the A-stationary dataflow here. SIGMA utilizes an Einsum cascade (Section 3.1), first identifying empty K-fibers (rows) of B (Line 9), removing them from A (Line 10), and then performing the multiplication (Line 11). We express the partitioning on Lines 21-23 using a combination of shape-based partitioning, flattening, and occupancy-based partitioning (Section 3.2.1). Finally, because all PEs work in parallel, the spatial dimension is $MK00$ (Line 36).

6 EXPERIMENTAL SETUP

This section describes the details of the experimental set-up used in Sections 7-8 to evaluate the performance characteristics of concrete accelerators.

Tensors. To evaluate the TeAAL models, we execute the models for the accelerators on a combination of randomly generated matrices with uniform sparsity and a set of matrices from SuiteSparse [11] and SNAP [26], described in Table 4.

Simulation Framework. We implement the accelerators by writing TeAAL specifications for their Einsums, mappings, formats, architectures, and bindings. For each accelerator, we use the hardware parameters given in Table 5. TeAAL uses Accelergy [51] as a power model to convert the per-component action counts to an energy characterization.

Baselines. To validate our results, we normalize our performance estimates using the same baseline as the original papers that published the relevant accelerators. All accelerators' "reported" statistics come either from published results or from private communication with the original authors. When possible, we also report Sparseloop [52] performance estimates using the *hypergeometric* sparsity distribution on both the inputs and outputs, estimated using the values in Table 4, and the hardware parameters in Table 5.

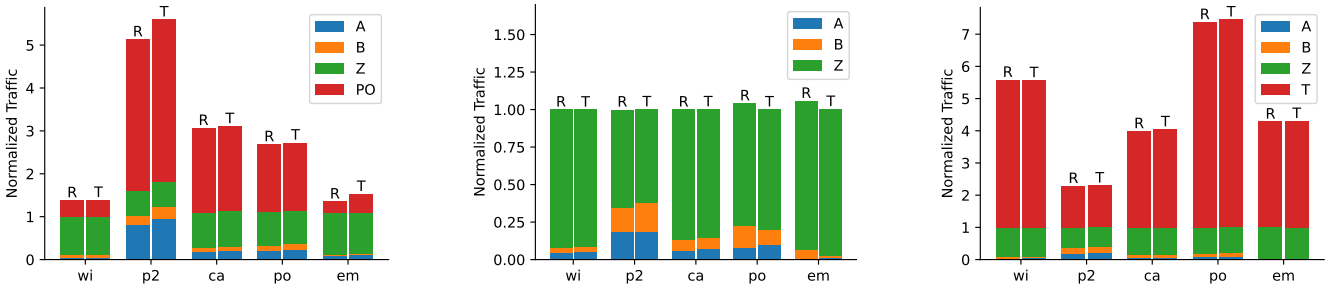
7 SIMULATOR VALIDATION

In this section, we describe a set of experiments used to validate TeAAL as an accurate cost model. Specifically, we compare memory traffic, performance, and power as reported by TeAAL to the numbers reported in the papers originally proposing each accelerator. We report all averages as arithmetic means, following the methodology presented in [21].

Memory Traffic. Figure 9 presents a comparison of the memory traffic of the TeAAL models of each of the accelerators to the corresponding baseline. We use the first five tensors in Table 4 because the prior work evaluates these tensors. The takeaway is that we can reproduce each accelerator's memory traffic with low error (on average, 3.8%). The single outlier, ExTensor on p2, is caused by slightly different policies for eager loading between ExTensor and TeAAL (Section 4.1.3). This policy difference is not fundamental, and can be remedied with additional effort. We were unable to validate TeAAL's memory traffic model of SIGMA because there were no baseline numbers available.

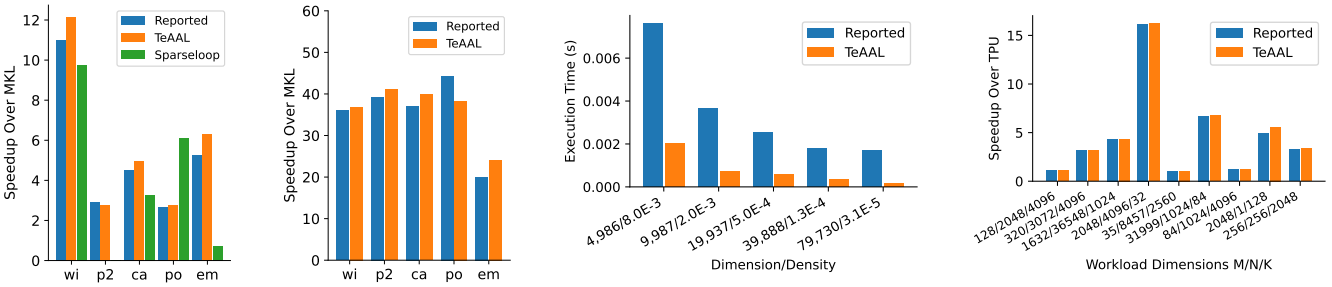
Performance. Figure 10 presents a performance comparison of each TeAAL model against the reported numbers and Sparseloop [52]'s estimate, when possible. We evaluate on the same five tensors as were used in the memory-traffic study or uniformly random sparse tensors.

Figures 10a and 10b show the speedup of ExTensor and Gamma, respectively, over Intel MKL. TeAAL shows consistently low error rates for each (on average, 10% and 6.4%, respectively). We perform an analogous evaluation for SIGMA in Figure 10d, relative to a Google Cloud TPU and using synthetic matrices with uniform-random sparsity (where all matrices A and B had 80% and 10% sparsity, respectively). Here, we show an average error of only 2.5%.



(a) ExTensor Memory Traffic (b) Gamma Memory Traffic (c) OuterSPACE Memory Traffic

Figure 9: Comparison of the memory traffic reported by the original publication (left, "R") and the TeAAL model (right "T") for ExTensor, Gamma, and OuterSPACE on 5 real-world tensors, discussed in Section 7. Traffic is normalized to the algorithmic minimum, and benchmark acronyms are defined in Table 4. Tensor names A, B, T, and Z correspond to the names defined in the Einsums in Figures 3 and 8. PO denotes the partial outputs (i.e. Z traffic that is not the final write of Z).



(a) ExTensor (b) Gamma (c) OuterSPACE (d) SIGMA

Figure 10: Validation of TeAAL-generated timing models against numbers/simulators reported/used in the original publications, also discussed in Section 7. Benchmark acronyms are given in Table 4.

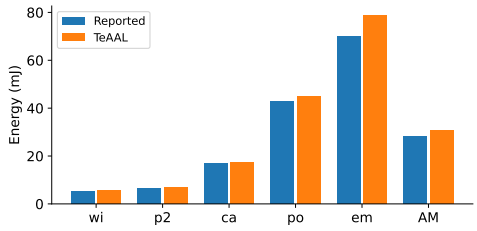


Figure 11: Validation of the ExTensor energy model, also discussed in Section 7. Benchmark acronyms can be found in Table 4.

We compare Sparseloop's results to ours on ExTensor in Figure 10a.[5] Sparseloop has an average error of 187%, which we attribute to its analytical sparsity distribution. We could not model p2 on Sparseloop because the tensor shape caused an integer overflow error.

Because we could not obtain the raw baseline numbers used in the OuterSPACE paper, Figure 10c shows the performance of the original simulator and the TeAAL model on a number of synthetic sparse matrices generated from a uniform-random distribution. On average, our cost model is consistently ~ 80% faster than the original simulator, though the overall trend is consistent. In Figure 9c, we showed that TeAAL models OuterSPACE's memory traffic with a < 1.8% error, so we suspect that the discrepancy in execution time comes from an undocumented (and, therefore, unmodeled) feature of the OuterSPACE PE microarchitecture.

Energy. Figure 11 compares TeAAL's energy estimates to the reported baseline, showing consistently low error rates—on average, 7.8%. TeAAL over estimates energy on em because it overestimates ExTensor's memory traffic on that benchmark (see Figure 9a). Since none of the other accelerators reported their per-component, per-action energy consumption characteristics, we were unable to validate the power model on those designs.

8 IMPROVING GRAPHICIONADO

In this section, we demonstrate both the generality of TeAAL as a tool for modeling a broader class of accelerators and its value when proposing and evaluating new designs. As an example, we model Grapicionado [14]—an accelerator for graph algorithms written in the vertex-centric programming paradigm—and GraphDynS [53]—an accelerator that optimizes this design. Finally, we propose further optimizations to GraphDynS and demonstrate their value by evaluating all three designs using TeAAL.

Vertex-centric programming describes graph algorithms from the point of view of a single vertex. During each iteration, a vertex, if active, sends its property to all of its destination vertices. Then, the vertex, whether or not it is active, processes its incoming properties, reduces them to a single new value, and applies this value to its

[5] We were unable to use Sparseloop to model Gamma/OuterSPACE (because it does not support cascades of Einsums) or SIGMA (because its occupancy-based partitioning is not sufficiently general).

```
1 # Processing Phase
2 – SO[d, s] = take(G[d, s], A0[s], 0)
3 – R[d] = SO[d, s] * A0[s]
4
5 # Apply Phase
6 – P1[v] = R[v] + P0[v]
7 – M[v] = P1[v] – P0[v]
8 – A1[v] = take(M[v], P1[v], 1)
```

(a) Graphicionado [14]

```
1 # Processing Phase
2 – SO[d, s] = take(G[d, s], A0[s], 0)
3 – R[d] = SO[d, s] * A0[s]
4
5 # Apply Phase
6 – MP[v] = take(R[v], P0[v], 1)
7 – NP[v] = R[v] + MP[v]
8 – M[v] = NP[v] – MP[v]
9 – P0[v] = take(M[v], NP[v], 1)
10 – A1[v] = take(M[v], NP[v], 1)
11 – P1 = P0
```

(b) GraphDynS [53]

Figure 12: Einsum cascades for two vertex-centric programming accelerators. A specific algorithm manifests by redefining the × and + operators (e.g., for SSSP, to addition and minimum, respectively).

property [44]. In this evaluation, we will focus on algorithms where a subset of the vertices are active each iteration.

Figure 12a shows a cascade of Einsums representing Graphicionado [14]. We omit the rest of the TeAAL specification for space. Graphicionado divides its evaluation into two stages. During the processing phase, the active vertices $A0$ are used to select the edges that need to be processed SO (Line 2), the weights of those edges are combined with the source vertex properties (Line 3), and reduced into R (implicit in Line 3). Then, during the apply phase, the vertex property $P0/P1$ is updated (Line 6) and the new set of active vertices $A1$ is created using a mask M of updated vertices (Lines 7-8). By redefining the multiplication and addition operators (e.g., for single source shortest path (SSSP), to addition and minimum, respectively), this represents a functionally correct implementation of a graph kernel written in the vertex-centric programming model. Through private correspondence with the Graphicionado authors, we found that all data, simulators, etc. used in this paper are proprietary, making it impossible for us to perform a similar analysis to Section 7. However, using TeAAL, we were able to profile Graphicionado ourselves and compare it with other designs.

GraphDynS [53] optimizes Graphicionado by adding new Einsums to the cascade to take advantage of the sparsity of R. Figure 12b shows the updated cascade. Building an additional intermediate MP (Line 6), containing the values of $P0$ that can be modified, decreases the memory traffic incurred by $P0$ and the number of apply operations the accelerator needs to perform. Filtering the writes to $P0$ with M (Line 9) also decreases the memory traffic. GraphDynS implements this optimization by keeping a 256-element bitmap, where each bit corresponds to 1/256th of the vertices. In TeAAL, this manifests as an additional uniform_shape partitioning. If the bit is 1, the accelerator eagerly loads the entire partition of vertex properties. GraphDynS further improves upon Graphicionado by changing the format of the graph from an edge-list representation to CSR. This format change eliminates unnecessary reloading of the source vertex ID and removes the loading of the edge weight for algorithms that do not use it (e.g., BFS).

We optimize this design by removing the partitioning, allowing us to only load the property and perform the apply on vertices that are actually modified during the processing phase. Our proposed accelerator also implements the format optimization.

Table 6: Sparse tensor modeling frameworks.

	STONNE [30, 31]	Sparseloop [52]	SAM [18]	CIN-P [2]	TeAAL (*this work*)
Models Hardware	✓	✓	✓		✓
Generic Kernels		✓	✓	✓	✓
Cascaded Einsums			✓	✓	✓
Index Expressions		✓			✓
Shape-Based Part.		✓	✓		✓
Occ.-Based Part.			✓		✓
Generic Flattening			✓		✓
Rank Swizzling				✓	✓
Format Expressivity		✓	✓	✓	✓
Caches	✓				✓
Precise Data Set	✓		✓		✓
High Model Fidelity	✓				✓

We evaluate Graphicionado, GraphDynS, and our proposal on a subset of the graphs and all sparse active vertex set algorithms evaluated in the original Graphiciondo paper. To enable an apples-to-apples comparison, we use the hardware parameters chosen for Graphicionado (see Table 5). Figure 13 shows the speedup achieved by each of the designs over Graphicionado. Figure 13a shows that our proposal enables an average of 1.9× improvement over GraphDynS on BFS, while Figure 13b shows that our proposal enables an average of 1.2× improvement over GraphDynS on SSSP. Figure 13c explains this improvement. While GraphDynS's bitmap approach reduces the number of apply operations required when the set of active vertices is small, our proposal is also able to skip apply operations when the set of active vertices is large.

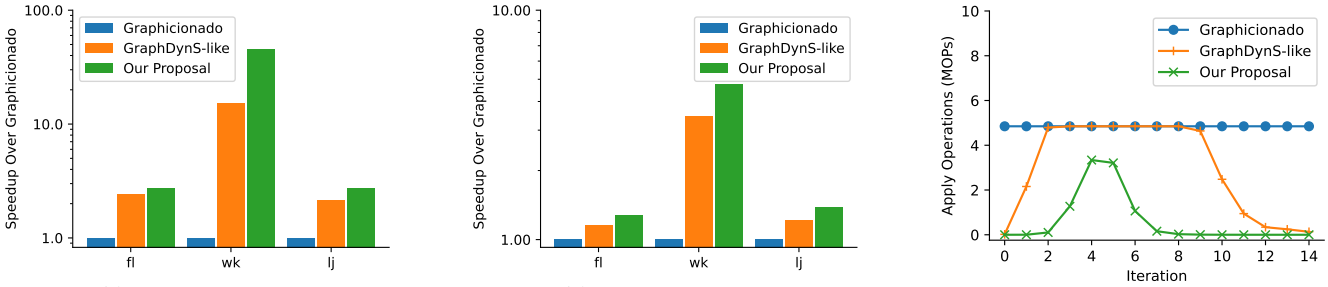
This study shows that TeAAL can express designs in domains beyond sparse tensor algebra. Furthermore, it also demonstrates TeAAL's value as a tool for qualitatively and quantitatively comparing designs, improving the iterative design refinement process. Notably, our proposed optimization only required meaningful changes to the mapping specification. By decomposing the design of an accelerator into a hierarchy of specifications, TeAAL enables us to efficiently express existing designs and propose new optimizations.

9 RELATED WORK

The rise in machine learning and tensor algebra accelerators has been followed by an increase in tools that explore the accelerator design space and model various efficiency characteristics [2, 19, 25, 29, 31, 35, 40, 52, 54]. Most frameworks solely support dense computations and target DNN applications [19, 25, 29, 35].

Table 6 compares frameworks that model architectures computing on sparse tensors. STONNE [31] is a cycle-level modeling framework for DNN accelerators. Like TeAAL, STONNE's analysis is data-driven; however, the only sparse workload it supports is SpMSpM.

Two other works—Sparseloop [52] and the Sparse Abstract Machine (SAM) [18]—model sparse workloads expressed in the Einsum language, but with lower fidelity than TeAAL. Sparseloop [52] has a flexible hardware backend and takes as input a specification of the architecture, a statistical model of the data, sparse optimizations such as intersection [16], and a user-specified mapping. It returns estimates of performance and energy consumption. Unlike TeAAL,



(a) Speed-Up on BFS

(b) Speed-Up on SSSP

(c) Apply Ops Performed for lj on BFS

Figure 13: Comparison between Graphicionado and improved accelerators. All designs use Graphicionado's parameterization (Table 5).

it does not support many of the features important to sparse computations, such as cascaded Einsums, caches, rank swizzling, and others. Of note, Sparseloop models sparsity analytically using probability distributions. By contrast, TeAAL evaluates on real tensors directly. This enables TeAAL to achieve higher-fidelity modeling, at the cost of increased simulator runtime. SAM [18] targets an architecture consisting of hardware modules similar to those supported by TeAAL. However, rather than generating high-fidelity models of specific accelerator designs, it models the accelerator dataflow on the SAM hardware.

Beyond accelerators, prior work on CIN-P [2] considers sparse workload modeling for programmable devices such as CPUs. Specifically, CIN-P is a mapper language that, when combined with an asymptotic cost model and autoscheduler, generates mappings that can then be compiled using TACO [23]. Since it uses asymptotic analysis, the mapper only considers a small subset of the space of mapping decisions, including loop order and fusion.

Beyond frameworks for modeling/evaluating sparse tensor algebra kernels, there is also a closely related line of work targeted at compiling these kernels for existing programmable devices [6, 9, 22, 23, 50]. These works provide a similar set of algorithms and mappings, with feature sets that overlap TeAAL. For example, Mosaic [6] supports modeling generic kernels, but inherits the limitations of the existing TACO [23] language. It does not support several idioms that TeAAL does, e.g., allowing affine expressions as tensor indices (only adding constraints to the shapes of ranks) and only supports 1D intermediates [22] (as opposed to generic cascades). However, unlike TeAAL, it allows users to mix existing library calls with user-defined kernels.

10 CONCLUSION AND FUTURE WORK

This paper presented TeAAL: a language and simulator generator for describing and evaluating sparse tensor accelerators. The key contribution is to demonstrate how state-of-the-art sparse accelerators can be represented as cascades of mapped Einsums and content-preserving transformations on the Einsums' constituent tensors. From this observation, we propose the TeAAL language which enables designers to utilize and combine these concepts (for both the modeling of current and the designing of new accelerators) and a generator from this language to executable simulators that emulate fibertree operations (lowered onto concrete data representations and hardware units). Beyond enabling a more efficient accelerator design process, TeAAL also provides the architecture community with a common language for comparing and discussing designs.

Using cascades of Einsums as the problem specification enables TeAAL to model accelerators in or near the space of tensor algebra, including for deep learning, tensor decomposition, and graph algorithms. We think that this abstraction can be extended to support a wider range of workloads and accelerators, e.g., by adding support for iterative algorithms or non-linear functions. Following the discussion in Section 4.1.4, other missing features can be added by augmenting other (specific) levels in TeAAL's pyramidal abstraction hierarchy. Our thesis is that many unsupported features will manifest as changes to one level (and for that matter, a lower level) in the hierarchy—which suggests they will be relatively easy to incorporate. For example, design changes to various types of hardware blocks (e.g., mergers, arithmetic units, memories) are representable as point changes to the architecture specification language (as opposed to changing the cascade of Einsums abstraction). We leave a full investigation to future work.

Another important direction for future work is to incorporate TeAAL into a flow for performing design space exploration. Such flows are typically hierarchical, starting with an exploration of the full design space using low fidelity models and only then performing a more in-depth, accurate analysis of the remaining promising designs [24]. We view TeAAL, as it is described in this work, as a middle level in this hierarchy. The simulators generated by TeAAL are higher fidelity than those produced by an analytical model like Sparseloop [52] and much more efficiently described by an architect than raw RTL. Then, the main technical challenge to solve is how to use TeAAL to *automatically* explore a space of designs. Specifically, allowing accelerators to be described as a cascade of Einsums (instead of a single Einsum [35, 52]) means that we would like to explore multiple cascades for the same problem, which requires a way to efficiently list the various promising cascades for that problem. This task, as far as we know, is open.

Finally, while the current TeAAL backend generates performance/energy models, we think future work could support other backends (e.g., for generating hardware directly).

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This research was partially funded by NSF grants 8191902 and 1942888, DARPA grant HR0011-18-3-0007, and by a UIUC SURGE Fellowship and Microsoft Research PhD Fellowship. We would like to thank a number of people for their help in enabling us to better validate/extend ExTensor, Gamma, OuterSPACE,

SIGMA, and Graphicionado: Tae Jun Ham, Kartik Hegde, Tushar Krishna, Francisco Muñoz-Martinez, Eric Qin, Daniel Sanchez, Hanrui Wang, Lisa Wu Wills, Guowei Zhang, and Zhekai Zhang. We would also like to thank Yannan (Nellie) Wu for help with Sparseloop, and Timor Averbuch, Alex Dicheva, John D. Owens, Yasmin Sarita, and Xinrui (Alice) Wu for many helpful discussions. Finally, we thank Willow Ahrens and Jaeyeon Won for feedback on early versions of the manuscript.

REFERENCES

[1] 2023. Fibertree Project. https://github.com/Fibertree-Project/fibertree.
[2] Peter Ahrens, Fredrik Kjolstad, and Saman P. Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *PLDI'22*.
[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *IPDPS'14*.
[4] Jorge Albericio, Patrick Judd, Tayler H. Hetherington, Tor M. Aamodt, Natalie D. Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *ISCA'16*.
[5] Ariful Azad, Aydın Buluc, and John Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *IPDPSW'15*.
[6] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. In *PLDI'23*.
[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. In *OSDI'18*.
[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *ISCA'16*.
[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. In *OOPSLA'18*.
[10] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* (1965).
[11] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. (2011).
[12] Stijn Dongen. 2000. Graph Clustering by Flow Simulation. *PhD thesis, Center for Math and Computer Science (CWI)* (2000).
[13] A. Einstein. 1916. The Foundation of the General Theory of Relativity. *Annalen der Physik* (1916).
[14] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO'16*.
[15] 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. In *ISCA'16*.
[16] 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 *MICRO'19*.
[17] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search. In *ASPLOS'21*.
[18] 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 *ASPLOS'23*.
[19] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *ISCA'21*.
[20] Jürg Hutter, Marcella Iannuzzi, Florian Schiffmann, and Joost VandeVondele. 2014. CP2K: Atomistic simulations of condensed matter systems. *WIREs Computational Molecular Science* (2014).
[21] Bruce Jacob and Trevor N. Mudge. 1995. Notes on Calculating Computer Performance.
[22] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *CGO'19*.
[23] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. In *OOPSLA'17*.
[24] B. Kumar and E. S. Davidson. 1980. Computer System Design Using a Hierarchical Approach to Performance Evaluation. *CACM'80* (1980).
[25] Hyoukjun Kwon, Michael Pellauer, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflows: A Data-Centric Approach Using MAESTRO. In *MICRO'19*.
[26] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[27] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. 2020. TensorDash: Exploiting Sparsity to Accelerate Deep Neural Network Training. In *MICRO'20*.
[28] Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. 2013. Standards for graph algorithm primitives. In *HPEC'13*.
[29] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. 2020. ZigZag: A Memory-Centric Rapid DNN Accelerator Design Space Exploration Framework. In *Arxiv'20*.
[30] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *ASPLOS'23*.
[31] Francisco Muñoz-Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2021. STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators. In *IISWC'21*.
[32] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2019. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Comput.* (2019).
[33] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel Emer, and Christopher Fletcher. 2023. Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling. In *ASPLOS'23*.
[34] 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 *HPCA'18*.
[35] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *ISPASS'19*.
[36] 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 *ISCA'17*.
[37] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen Keckler, Christopher W. Fletcher, and Joel Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *ASPLOS'19*.
[38] 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 *HPCA'20*.
[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI'13*.
[40] 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. In *OOPSLA'20*.
[41] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *SC'17*.
[42] 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 *MICRO'20*.
[43] 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 *HPCA'20*.
[44] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. In *VLDB'15*.
[45] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. *Efficient Processing of Deep Neural Networks*. Springer.
[46] Joost VandeVondele, Urban Borštnik, and Jürg Hutter. 2012. Linear Scaling Self-Consistent Field Calculations with Millions of Atoms in the Condensed Phase. *Journal of Chemical Theory and Computation* (2012).
[47] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side Sparse Tensor Core. In *ISCA'21*.
[48] Sasindu Wijeratne, Rajgopal Kannan, and Viktor Prasanna. 2021. Reconfigurable Low-latency Memory System for Sparse Matricized Tensor Times Khatri-Rao Product on FPGA. In *HPEC'21*.
[49] Jan Wilhelm, Patrick Seewald, Mauro Del Ben, and Jürg Hutter. 2016. Large-Scale Cubic-Scaling Random Phase Approximation Correlation Energy Calculations

Using a Gaussian Basis. *Journal of Chemical Theory and Computation* (2016).
[50] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. 2023. Unified Convolution Framework: A compiler-based approach to support sparse convolutions. In *MLSys'23*.
[51] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *ICCAD'19*.
[52] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2022. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. In *MICRO'22*.
[53] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *MICRO'19*.
[54] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *ASPLOS'20*.
[55] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *ASPLOS'21*.
[56] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *HPCA'20*.

A ARTIFACT APPENDIX

A.1 Abstract

In this artifact, we provide the source code for TeAAL, a simulator generator for sparse tensor algebra accelerators, as well as the scripts required to display the results of the simulation. For ease-of-use, we provide a Docker container and a set of Jupyter notebooks through which to run the experiments. This artifact can be evaluated on an x86-84 machine with 256 GB of memory and 75 GB of disk space.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Automatic generation of sparse tensor algebra accelerator simulators
- **Program:** Python, Sparseloop
- **Run-time environment:** Docker, Jupyter
- **Hardware:** An x86-64 machine with 256 GB of memory and 125 GB of disk space
- **Output:** Plots generated from scripts
- **Experiments:** Automatic generation of sparse tensor algebra accelerator simulators and execution of those simulators on specific benchmark data
- **How much disk space required (approximately)?:** 125 GB
- **How much time is needed to prepare workflow (approximately)?:** < 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 70 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT

A.3 Description - How to Access

Manually: The artifact is hosted on Github at https://github.com/FPSG-UIUC/micro23-teaal-artifact. Following the instructions in this repository will allow you to run specific experiments and nicely display the graphs.

Via the MLCommons CM Interface: It is also accessible through the MLCommons CM interface at https://github.com/ctuning/cm-reproduce-research-projects/tree/main/script/reproduce-ieee-acm-micro2023-paper-8. This method provides less control over what experiments are executed.

A.4 Installation

Manually: Since we provide a Docker container with all dependencies pre-installed, the artifact relies on Docker and access to a web browser. Specific installation instructions can be found at https://github.com/FPSG-UIUC/micro23-teaal-artifact/blob/main/README.md.

Via the MLCommons CM Interface: The instructions for installation can be found at https://github.com/ctuning/cm-reproduce-research-projects/blob/main/script/reproduce-ieee-acm-micro2023-paper-8/README.md.

A.5 Evaluation

Manually: We provide two notebooks to guide you through the evaluation: `notebooks/figs9and10.ipynb` and `notebooks/fig11.ipynb`. Please launch the docker container, open the Jupyter Lab in a web browser, and navigate to this notebook. Each cell either runs a simulation or displays a graph. The output of each display cell corresponds to a figure in the paper.

Via the MLCommons CM Interface: The instructions for evaluation can be found at https://github.com/ctuning/cm-reproduce-research-projects/blob/main/script/reproduce-ieee-acm-micro2023-paper-8/README.md

A.6 Expected Results

This artifact reproduces Figures 9a-11. The easiest way to check validity is to visually compare the figures, but raw results will be written to `data/generated/` and can be compared with the expected results found in `data/pregenerated/`. We note that certain experiments use randomly generated sparse tensors whose performance characteristics will exhibit some variety. Such datasets are noted in the notebook, and simulations can be rerun to obtain new seeds.

A.7 Experiment Customization

Input specifications in `yamls/teaal/` can be updated to work on new kernels, execute new mappings, represent tensors with new formats, and evaluate new architectures.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html