Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling

Yannan Nellie Wu	Po-An Tsai	Angshuman Parashar	Vivienne Sze	Joel S. Emer
MIT	*NVIDIA*	*NVIDIA*	*MIT*	*MIT / NVIDIA*
Cambridge, US	*Westford, US*	*Westford, US*	*Cambridge, US*	*Cambridge, US*
nelliewu@mit.edu	*poant@nvidia.com*	*aparashar@nvidia.com*	*sze@mit.edu*	*jsemer@mit.edu*

2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00096

***Abstract*—In recent years, many accelerators have been proposed to efficiently process sparse tensor algebra applications (*e.g.*, sparse neural networks). However, these proposals are single points in a large and diverse design space. The lack of systematic description and modeling support for these sparse tensor accelerators impedes hardware designers from efficient and effective design space exploration.**

This paper first presents a unified taxonomy to systematically describe the diverse sparse tensor accelerator design space. Based on the proposed taxonomy, it then introduces Sparseloop, the first fast, accurate, and flexible analytical modeling framework to enable early-stage evaluation and exploration of sparse tensor accelerators. Sparseloop comprehends a large set of architecture specifications, including various dataflows and sparse acceleration features (*e.g.*, elimination of zero-based compute). Using these specifications, Sparseloop evaluates a design's processing speed and energy efficiency while accounting for data movement and compute incurred by the employed dataflow, including the savings and overhead introduced by the sparse acceleration features using stochastic density models.

Across representative accelerator designs and workloads, Sparseloop achieves over 2000× faster modeling speed than cycle-level simulations, maintains relative performance trends, and achieves 0.1% to 8% average error. The paper also presents example use cases of Sparseloop in different accelerator design flows to reveal important design insights.

***Keywords*-Tensor computation; Hardware Accelerator; Analytical modeling**

I. Introduction

Sparse tensor algebra is widely used in many important applications, such as scientific simulations [1], computer graphics [2], graph algorithms [3], [4], and deep neural networks (DNNs) [5], [6]. Depending on the sparsity characteristics of the tensors (*e.g.*, sparsity, distribution of zero locations), sparse tensor algebra can introduce a significant number of *ineffectual computations*, whose results can be easily derived by applying the simple algebraic equalities of $X \times 0 = 0$ and $X + 0 = X$, without reading all the operands or doing the computations [7], [8].

As a result, many performant and energy-efficient sparse tensor algebra accelerators have been proposed to exploit ineffectual computations to reduce data movement and compute [9], [10], [11], [8], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. Based on the properties of its target applications (*e.g.*, convolution or matrix multiplication), each accelerator design proposes its unique hardware support. Various accelerators can: propose different architecture topology (*e.g.*, number of storage levels); employ different *dataflows* [9] (*i.e.*, the rules for scheduling data movement and compute in space and time); use different *encoding* to compress sparse tensors (*e.g.*, bitmask encoding); and design different hardware to eliminate operations associated with ineffectual computations (*e.g.*, intersection units). The joint design space for all these hardware mechanisms is therefore large and diverse.

To characterize either a single specific design or many designs as part of design space exploration, hardware designers can benefit from a modeling framework that is:

- ***Flexible:*** is capable of modeling a diverse range of potential designs with hardware support for different dataflows, compression encodings, etc.
- ***Fast:*** produces simulation results quickly. This is particularly important because properly characterizing a specific design requires finding the best schedule, i.e., *mapping*, for a given workload, which generally requires a search of a large mapspace [22], [23], [24].
- ***Accurate:*** produces simulation results correctly in both mapspace and design space exploration.

However, to the authors' knowledge, none of the existing modeling approaches for tensor accelerators provide the desired capabilities (Table I). On the one hand, *cycle-level design-specific models* [9], [10], [11], [8], [25], [12], [26], [27], [13], [14], [15], [28] capture the detailed implementations for their target designs (*e.g.*, memory control signals) and thus are very accurate. However, such models impede users from mapspace exploration due to their slow simulation speed and design space exploration due to their limited parameterization support. On the other hand, analytical models perform mathematical computations to analyze the important high-level characteristics of a class of accelerators and are fast. However, the existing *general analytical models* are only flexible for dense accelerator designs [24], [29], [30], [31], [32], [33], [34], [35], [36], [37], *i.e.*, they do not reflect the impact of sparsity-aware acceleration techniques, resulting in inaccurate modeling.

To address the limitations of existing work, we present

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00096

	Accuracy	Speed	Flexibility	Support Sparsity?
Cycle-Level Design-Specific	Very High	Slow	Low	Yes
General Analytical	High	Fast	High	No
Our Work	High	Fast	High	Yes

Table I: Comparison of existing tensor accelerator simulation frameworks with our proposed Sparseloop framework.

Sparseloop[1], the *first analytical modeling framework* for fast, accurate, and flexible evaluations of sparse (and dense) tensor accelerators, enabling early-stage exploration of the large and diverse sparse tensor accelerator design space. Table I compares Sparseloop to existing simulation frameworks.

This work makes the following key contributions:

(1) To systematically describe the large and diverse sparse tensor accelerator design space, we propose a taxonomy to classify the various sparsity-aware acceleration techniques into three *sparse acceleration features (SAFs)*: representation format, gating, and skipping.

(2) Based on the SAF classification, we propose Sparseloop, an analytical modeling framework for tensor accelerators.

- To both faithfully reflect workload data's impact on accelerator performance and ensure simulation speed, Sparseloop performs analysis based on statistical characterizations of nonzero value locations in the tensors.
- To keep the modeling complexity tractable and allow support for emerging workloads/designs, Sparseloop splits its modeling process into discrete steps, each of which focuses on evaluating a distinct design aspect (*e.g.*, dataflow, sparse acceleration features). This decoupling allows modeling of both dense and sparse designs in one infrastructure.

(3) With representative accelerator designs and workloads, we show that Sparseloop is fast, accurate, and flexible:

- Sparseloop runs more than 2000× faster than a cycle-level simulator.
- Sparseloop maintains relative performance trends and achieves 0.1% to 8% average error across designs.
- Sparseloop allows comparison of designs with different dataflows and sparse acceleration features, running workloads with various sparsity characteristics.
- Sparseloop can reveal design insights during accelerator design flows. Our case studies demonstrate Sparseloop's flexibility to quickly compare and explore diverse designs with different architectures, dataflows, and SAFs running various workloads.

II. Background and Motivation

In this section, we illustrate the complexity of describing and evaluating the sparse tensor accelerator design space.

[1]Sparseloop is open-source and publicly available at [38].

A. Large and Unstructured Design Space

Sparse tensor accelerators often employ different dataflows to exploit data reuse across multiple storage levels and feature various sparsity-aware acceleration techniques to eliminate data storage for zeros and *ineffectual operations* (IneffOps), *i.e.*, arithmetic operations and storage accesses associated with ineffectual computations. The vast number of potential design choices lead to a large and diverse design space.

Nonetheless, there is little structure in the design space for sparse tensor accelerators, as each prior design uses different terminology to describe a point in the design space. We present the design decisions made by representative designs to show the lack of uniformity in their architecture proposals.

For example, Eyeriss [9] uses a *row-stationary* dataflow, a *RLC* encoding for data stored in DRAM, and storage and compute units that stay idle for IneffOps. With the same dataflow, Eyeriss V2 [10] employs a *compressed sparse column* encoding for both on-chip and DRAM data, and avoids spending cycles for IneffOps by performing intersections near the compute units. SCNN [11] also uses a similar intersection-based acceleration, but features a *PlanarTiled-InputStationary-CartesianProduct* dataflow and *compressed-sparse-block* encoding. ExTensor [8] proposes a *hybrid* dataflow and a *hierarchical* encoding. It introduces the *hierarchical-elimination* acceleration technique, which aggressively eliminates IneffOps at multiple storage levels long before data reaches compute. Dual-side sparse tensor core (DSTC) [21] uses an *output-stationary* dataflow and *two-level BitMap* encoding. It designs an *operand-collector* hardware unit tailored to its dataflow to provide enough bandwidth after elimination of IneffOps.

Since different accelerators propose different sets of implementation choices, often described in design-specific naming conventions, it is challenging for designers to have a systematic understanding of the proposed dataflow and acceleration techniques in the design space, let alone a modeling framework to compare these designs systematically.

B. Sparsity Impacts Design Behavior

Evaluating the complex design space of sparse tensor accelerators is further complicated by the impact of the tensor *sparsity characteristics*, which include the density (*i.e.*, percentage of nonzero values in each tensor, $1-$sparsity) and the locations of nonzero values in each tensor.

To demonstrate this entanglement, we compare two designs supporting different data representations. For simplicity, both accelerators employ the same dataflow:

(1) Bitmask (Eyeriss-like): The first design supports bitmask encoding to represent sparse operand tensors. Bitmask uses a single bit to encode whether each value is nonzero or not. In each cycle, the design uses each bit to decide whether its storage and compute units should stay idle to

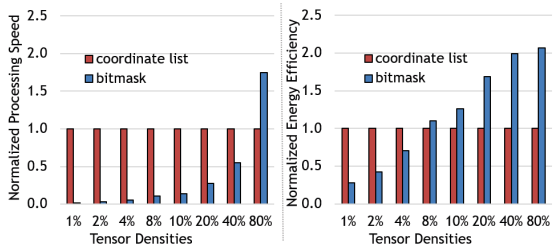


Figure 1: Processing speed and energy efficiency of architectures with different data representation support running sparse matrix multiplication workloads. Design behavior is dependent on data representations and tensor densities.

save energy, but it does not improve processing speed.[2]

(2) Coordinate list (SCNN-like): The second design employs a coordinate list encoding [39], [40], which indicates the location of each nonzero value via a list of its coordinates (*i.e.*, the indices in each dimension). Since the coordinate information directly points to the next effectual computation, the design only spends cycles on effectual operations, thus saving both energy and time.

In Fig. 1, we compare the processing speed and energy efficiency of the two designs running sparse matrix multiplication workloads of different densities. As shown in Fig. 1, the best design choice is a function of the input density. More specifically, since the bitmask-based design does not improve processing speed, with low-density tensors, *bitmask* always runs slower than *coordinate list*. However, since *coordinate list* needs to encode the exact coordinates with multiple bits, it incurs more significant encoding overhead per nonzero value. As the tensors become denser, *coordinate list* leads to lower energy efficiency and/or processing speed. This trend has also been observed in Sigma [15].

Even just varying the input tensor density, we already see non-trivial interactions between the benefits introduced by eliminated IneffOps and compressed sparse tensors, and the overhead introduced by extra encoding information. A more involved case study in Sec. VII-A will further showcase the complex interactions between dataflows, sparsity-aware acceleration techniques, and workload sparsity characteristics, illustrating the importance of co-designing various design aspects. Thus, for hardware designers to efficiently explore the trade-offs of various design decisions, there is a strong need to have a *fast modeling framework that, in addition to evaluating various dataflows, recognizes the impact of the different acceleration techniques and tensor sparsity characteristics on processing speed and energy efficiency*.

Our proposal: To address these two issues, we first introduce a new classification of the various sparsity-aware acceleration techniques (Sec. III), which unifies how to qualitatively describe these techniques in a systematic manner. Leveraging this taxonomy, we then propose an analytical modeling framework, Sparseloop, which quantitatively evaluates the diverse sparse tensor accelerator designs (Sec. IV to Sec. V-D). We show that Sparseloop is fast, accurate, and flexible in Sec. VI-B, VI-C, and VII.

[2]Of course, there exist other designs that use bitmasks to save both energy and time [17], [25]

III. Design Space Classification

The first step toward a systematic modeling framework is to have a unified taxonomy to describe various sparse tensor accelerators. We propose a new classification framework that simplifies how to describe a specific design in the complex design space. We then demonstrate how prior designs can be described in a straightforward manner.

A. High-Level Sparse Acceleration Features

To systematically describe sparse tensor accelerators in the design space, we classify common sparsity-aware acceleration techniques into three orthogonal high-level categories:

- Representation format
- Gating IneffOps
- Skipping IneffOps

We call each category a *sparse acceleration feature* (SAF).

1) Representation Format: Representation format refers to the choice of encoding the locations of nonzero values in the tensor. To describe a representation format, we adopt a hierarchical expression that combines multiple per-dimension formats, similar to [41], [39], [40].

As shown in Fig. 2, we introduce several commonly used per-dimension formats with an example 1D tensor, *i.e.*, a vector. The most basic format is *Uncompressed (U)*, which represents the tensor with its exact values, thus directly showing the locations of nonzero values. *U* is identical to the original vector. However, to save storage space, and thus implicitly save energy (and time) associated with zero value accesses, sparse tensor accelerators tend to employ *compressed formats*, which represent a tensor with only nonzero values and some additional information about their original locations or *coordinate* [41], [39], [40]. We call this information *metadata*. We introduce four per-dimension compressed formats[3].

- *Coordinate Payload (CP)*: the coordinates of each nonzero value are encoded with multiple bits. The *payloads* are either the nonzero value or a pointer to another dimension. CP explicitly lists the coordinates and the corresponding payloads.
- *Bitmask (B)*: a single bit is used to encode whether each coordinate is nonzero or not.
- *Run Length Encoding (RLE)*: multiple bits are used to encode the run length, which represents the number of

[3]Of course, many more per-dimension formats exist and can be incorporated modularly into Sparseloop.

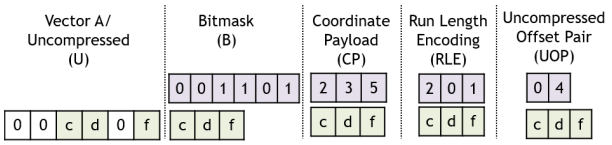


Figure 2: Example representation formats of a vector A. Purple vectors refer to metadata used to identify the original locations of the nonzero values.

Example Classic Representation Formats	Hierarchical Description
Compressed Sparse Row (CSR) [42]	UOP-CP
2D Coordinate List (COO) [43]	CP^2
3D Compressed Sparse Fibers (CSF) [44]	CP-CP-CP

Table II: Example representation formats and their hierarchical description based on per-dimension formats.

zeros between nonzeros (*e.g.*, an r-bit run length can encode up to a $2^r - 1$ run of zeros).

- *Uncompressed Offset Pairs (UOP)*: multiple bits are used to encode the start (inclusive) and end (noninclusive) positions of nonzero values.

As shown in Table II, full tensor representation formats can be described by combing the per-dimension formats in a hierarchical fashion. For example, CSR (compressed sparse row) [42] can be described by *UOP-CP*: Top level UOP encodes the start and end locations of the nonzeros in each row; bottom level CP encodes the exact column coordinates and its associated nonzeros. A format can also split and/or flattened tensor dimensions (*e.g.*, 2D COO flattens multiple dimensions into one dimension represented by CP with tuples as coordinates). We use a superscript to indicate the number of flattened dimensions.

2) Gating: Gating exploits the existence of IneffOps by letting the storage and compute units stay idle during the corresponding cycles. As a result, it saves energy but does not change processing speed. Gating can be applied to both compute and storage units in the architecture.

We use the dot-product workload in Fig. 3a to illustrate the impact of gating. Each row of Fig. 3b corresponds to a specific SAF implementation, each column is a processing step, and each cell lists the operations happening at the step. The first row presents the baseline processing without any SAFs applied, so it performs all IneffOps and takes six steps to complete.

The second row in Fig. 3b shows the result of applying gating to compute units, $Gate\,Compute$. The compute unit checks whether operands are zeros and stays power-gated if at least one operand is zero.

When gating is applied to storage units, it can be based on one of two approaches:

(1) Leader-follower intersection checks one operand, and if this operand is zero, it avoids accessing the other operand. We call the checked operand the *leader* and the operand with gated access the *follower*. In our classification, gating based on leader-follower intersection is represented by an arrow that points from the leader to the follower, *i.e.*, $Gate\ Follower \leftarrow Leader$. The third row in Fig. 3b shows $Gate\ B \leftarrow A$. Note that this approach may not eliminate all IneffOps (*e.g.*, step three in the example), and the savings introduced depend on the leader operand's sparsity characteristics.

(2) Double-sided intersection checks both operands (usually just via their associated metadata), and if either of them is zero, it does not access either operands' data. Double-sided intersection is represented with a double-sided arrow that points to both operands, *i.e.*, $Operand0 \leftrightarrow Operand1$. Double-sided intersection eliminates all IneffOps but may require more complex hardware.

In addition to reducing storage accesses, gating applied to a storage unit also leads to *implicit* gating of the compute unit connected to it (*e.g.*, step one in the third row of Fig. 3b), as the compute unit can now use the check for the storage unit to power-gate itself.

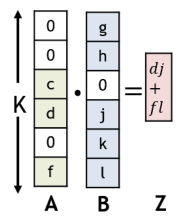
3) Skipping: Skipping refers to exploiting IneffOps by not spending the corresponding cycles. Since skipping directly skips to the next effectual computation, it saves both energy and time. Similar to gating, skipping can be applied to both the compute and storage units.

When skipping is applied to compute units, the compute units directly look for the next pair of operands until it finds effective computations to perform. When skipping is applied to storage units, it can also be based on leader-follower intersection or double-sided intersection. However, instead of letting the storage stay idle, with skipping applied, cycles are only spent on effectual accesses. The last row in Fig. 3b shows an example implementation of skipping B reads based on A's values ($Skip\ B \leftarrow A$). Similar to gating, a leader-follower implementation of skipping can still introduce some IneffOps, and skipping at storage can lead to implicit skipping at the compute units. Since skipping needs to quickly locate the next effectual operation to skip to, it usually requires more complex hardware than gating does (*e.g.*, ExTensor's intersection unit implements smart look-ahead optimizations to locate effectual operations in time [8]). Inefficient implementations can lead to more overhead than savings in time and energy.

B. Dataflow is Orthogonal to Sparsity-Aware Acceleration

In addition to the SAFs, dataflow choice is another important decision made by various accelerators [45]. A taxonomy of dataflows for various tensor algebra workloads has already been well studied in existing work (*e.g.*, for DNNs [45], [40], and matrix multiplications [16], [13], [14]).

We make the observation that the dataflow choice is *orthogonal to* the chosen SAFs. Dataflows define the scheduling of data movement and compute in time and space, and SAFs define the actual amount of data that is moved or



(a)

processing steps →

SAFs	1	2	3	4	5	6
None	read(0) read(g) compute(0,g)	read(0) read(h) compute(0,h)	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(0) read(k) compute(0,k)	read(f) read(l) compute(f,l)
Gating on compute (*Gate Compute*)	read(0) read(g) ~~compute(0,g)~~	read(0) read(h) ~~compute(0,h)~~	read(c) read(0) ~~compute(c,0)~~	read(d) read(j) compute(d,j)	read(0) read(k) ~~compute(0,k)~~	read(f) read(l) compute(f,l)
Gating on B storage accesses based on A (*Gate B ← A*)	read(0) ~~read(g)~~ ~~compute(0,g)~~	read(0) ~~read(h)~~ ~~compute(0,h)~~	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(0) ~~read(k)~~ ~~compute(0,k)~~	read(f) read(l) compute(f,l)
Skipping on B storage accesses based on A (*Skip B ← A*)	read(c) read(0) compute(c,0)	read(d) read(j) compute(d,j)	read(f) read(l) compute(f,l)			

(b)

Figure 3: (a) Sparse dot product workload. (b) Example ways of processing the example workload. 1st row: baseline processing without SAFs; 2nd row: Gating applied to compute; 3rd row: Gating applied to B reads based on A's values; 4th row: Skipping applied to B reads based on A's values.

Design	Workload	Format[4]	Gating/Skipping
Eyeriss [9]	DNN	offchip: I/O: B-RLE W:U onchip: I: UB O/W:U	Innermost Storage : $Gate\ W \leftarrow I$, $Gate\ O \leftarrow I$
Eyeriss V2 [10]	DNN	I/W: B-UOP-CP O:U	Innermost Storage : $Skip\ W \leftarrow I$, $Skip\ O \leftarrow I\ \&W$; $Gate\ Compute$
SCNN [11]	DNN	I/W: B-UOP-RLE O: U	Innermost Storage : $Skip\ W \leftarrow I$, $Skip\ O \leftarrow I\ \&W$; $Gate\ Compute$
ExTensor [8]	MM	A/B: UOP-CP×5 Z: U	All Storage : $Skip\ A \leftrightarrow B$, $Skip\ Z \leftarrow A\ \&B$
DSTC [21]	MM	A/B: B-B Z: U	2^{nd}-to-innermost & Innermost Storage : $Skip\ A \leftrightarrow B$, $Skip\ Z \leftarrow A\ \&B$

Table III: Summary of representative sparse tensor accelerators described with the proposed SAFs based on tensors from example target workloads. For DNN: I: input activation, W: Weights, O: output activation. For Matrix Multiplication (MM): A,B: operand tensors, Z: result tensor. Note that the designs have different dataflows, which are not listed.

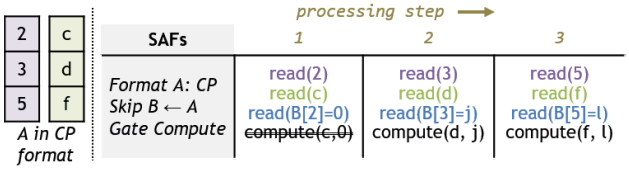


Figure 4: Example of combining compressed format, skipping, and gating SAFs in one design.

number of computes performed. As a result, the space of sparse tensor accelerators is the cross product of dataflow choices and SAF choices (further information on how this impacts modeling is in Section IV). Of course, a particular dataflow might mesh well with a specific SAF implementation, leading to an efficient design, while another may not.

C. Describing Sparse Tensor Accelerators

General accelerator designs often implement multiple SAFs that work well with each other to efficiently improve hardware performance. Fig. 4 illustrates the idea with a simple example, for the same workload in Fig. 3a, Fig. 4 employs a CP representation format for vector A, $Skip\ B \leftarrow A$ and $Gate\ Compute$. By representing A with CP, $Skip\ B \leftarrow A$ is implemented by directly reading the appropriate B values based on A's metadata. Furthermore, by applying $Gate\ Compute$, Fig. 4 eliminates the compute unit's IneffOps for cases with nonzero A and zero B.

Realistic sparse tensor accelerators often feature multiple storage levels to exploit data reuse opportunities and a set of spatial compute units for parallel computation. Thus, to systematically describe each design, we need to define the SAFs implemented at each level in the architecture. Based on our proposed classification, Table III describes the acceleration techniques of the representative designs introduced in Sec. II.

For example, SCNN [11], a sparse DNN accelerator, uses a three-level, B-UOP-RLE representation format[4] to compress input activation (IA) and weights (W). In the innermost storage level, *i.e.*, the level closest to the compute units, SCNN performs $SkipW \leftarrow IA$ and $Skip\ OA \leftarrow IA\ \&W$, where OA refers to output activation. Gating is applied to compute units, *i.e.*, $Gate\ Compute$, to eliminate leftover IneffOps, similar to Fig. 4's strategy. ExTensor [8] is an accelerator for general sparse tensor algebra. We use matrix multiplication as an example workload, which involves operand tensors A, B and result tensor Z. ExTensor partitions and compresses tensors with a six-level format and performs $Skip\ A \leftrightarrow B$ and $Skip\ Z \leftarrow A\&B$ at all storage levels.

[4]Some per-dimension formats are applied to split or flattened dimensions.

Thus we hope it is clear how this taxonomy allows the design-specific terminologies in existing proposals to be translated into systematic descriptions. More importantly, it also allows future sparse accelerators to be described accurately and compared *qualitatively* in the same way.

IV. Sparseloop Overview

The design space taxonomy in Sec. III lays the foundation for the modeling methodologies of Sparseloop, an analytical modeling framework that *quantitatively* evaluates the processing speed and energy efficiency of sparse tensor accelerators. In this section, we will discuss the modeling challenges and Sparseloop's key methodologies to address those challenges.

A. Modeling Challenges

There are three key challenges associated with ensuring the modeling framework's speed, accuracy, and flexibility.

Multiplicative factors of the design space. To faithfully model various sparse tensor accelerator designs, the analysis framework needs to understand the compound impact of their sparsity-specific design aspects (*e.g.*, the diverse SAFs shown in Table III) together with general design aspects (*e.g.*, architecture topology, dataflow, etc.). Simultaneously modeling the interactions between a considerable number of design aspects incurs high complexity, slowing down the modeling process. Building specific models for each design cannot scale to cover the entire design space, either.

Tradeoff between accuracy and modeling speed. High fidelity modeling requires time-consuming sparsity-dependent analysis. Since sparsity characteristics impact a sparse accelerator's performance, carefully examining the exact data in each tensor could ensure accuracy. However, the downside of actual-data based analysis is that it can cause intolerable slowdown during mapspace exploration, especially for workloads with numerous and evolving data sets, *e.g.*, DNNs.

Evolving designs/workloads. Finally, diverse and constantly evolving designs/workloads require flexibility and extendability in the modeling framework. Since the interactions between the processing schedules and workload data characteristics are convoluted, the framework must be flexible and modularized enough to allow easy extensions for future designs/workloads.

B. Sparseloop Solutions to the Challenges

To solve the challenges, Sparseloop makes two important observations for sparse accelerator modeling: (1) the runtime behaviors of sparse accelerators (*e.g.*, number of storage accesses and computes) can be progressively modeled; (2) the sparsity-dependent behavior in sparse accelerators can be statistically modeled with negligible errors.

Based on observation (1), to maintain modeling complexity, Sparseloop performs decoupled modeling of distinct design aspects: Sparseloop evaluates dataflow independent of SAFs, as the storage accesses and computes introduced by the dataflow are irrelevant to how the IneffOps get eliminated; Sparseloop evaluates SAFs independent of micro-architecture, as the number of eliminated IneffOps introduced by the SAFs is orthogonal to the cost of performing each elimination or the savings brought by each eliminated IneffOp. Thus, as Fig. 5 shows, Sparseloop's modeling process is split into three steps, each with tractable complexity.

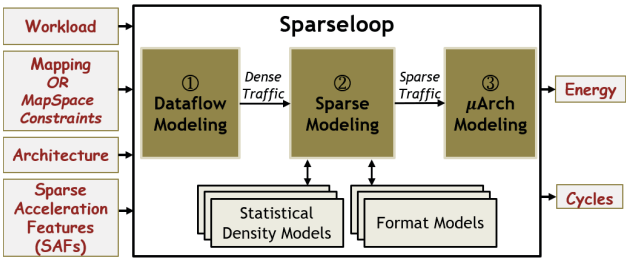


Figure 5: Sparseloop High-Level Framework.

- ***Dataflow modeling:*** analyzes the uncompressed data movement and dense compute, *i.e.*, dense traffic, incurred by the user-specified mapping input.
- ***Sparse modeling:*** analyzes and reflects the impact of SAFs by filtering the dense traffic to produce sparse data movement and sparse compute, *i.e.*, sparse traffic.
- ***Micro-architecture modeling:*** analyzes the exact hardware operation cost (*e.g.*, multi-word storage access cost) and generates the final energy consumption and processing speed based on the sparse traffic.

Based on observation (2), Sparseloop enables systematic recognition of the impact of SAFs at the sparse modeling step. To balance accuracy and speed, sparse modeling performs analysis based on *statistical characterizations* of nonzero value locations in workload tensors and their subtensors, by leveraging various statistical density models.

Finally, as shown in Fig. 5, to ensure extendability, the sparse modeling step interacts with statistical density models and per-dimension format models as decoupled modules so that these models can be extended to support future sparse workloads and representation formats.

V. Sparseloop Framework

We first discuss the inputs to Sparseloop in Sec. V-A, and describe the modeling steps in Sec. V-B, V-C, and V-D.

A. Inputs

As shown in Fig. 5, Sparseloop needs four inputs: workload specification, architecture specification, SAFs specification, and mapping or mapspace constraints. Fig. 6 shows a set of example specifications to show the input semantics, and more detailed syntax can be found at [38].

Workload specification describes the shape and statistical density characteristics of the workload tensors (*e.g.*, in

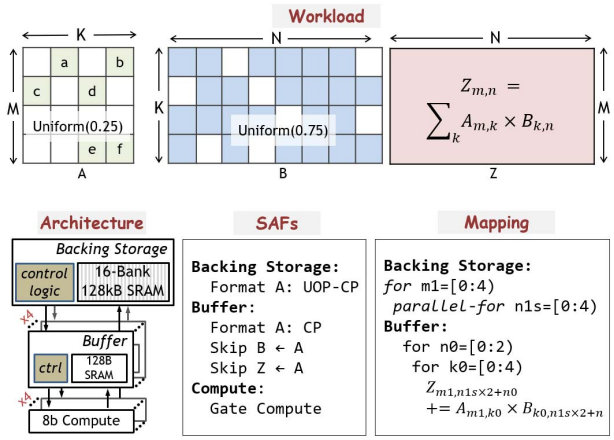


Figure 6: Example input specifications to Sparseloop. Blank spaces in the workload tensors refer to locations with zeros. The locations of zeros are just for illustrative purposes.

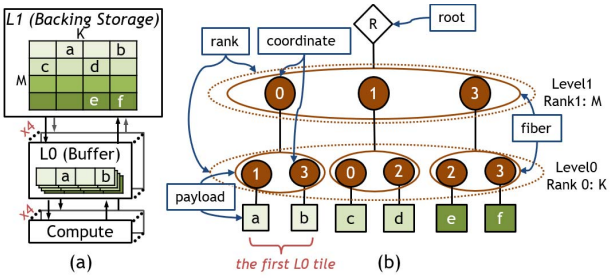


Figure 7: (a) Example coordinate-space tiling for tensor A based on inputs specified in Fig. 6. The shades represent tiles processed at different time steps. (b) Fiber tree representation of the tensor A. Each level of the tree corresponds to a *rank* of the tensor and contains one or more *fibers* that correspond to the rows or columns of the tensor. The leaves of the tree are the (nonzero) data values of the tensor.

Fig. 6, A is 4x4 and has a density of 25% with a uniform distribution). Workload specification also includes the tensor algorithm specification, which is based on the well-known Einsum notation [46], [41] (*e.g.*, the matrix multiplication kernel is specified as $Z_{m,n} = \sum_k A_{m,k} \times B_{k,n}$, where the A and B values along the same k dimension are reduced and the m and n dimensions are populated to the output tensor Z). Sparseloop understands any algorithm described with an extended Einsum notation, similar to existing works [24], [8].

Architecture specification describes the hardware organization of the architecture (*e.g.*, two levels of storage and four compute units) and the hardware attributes of the component in the architecture (*e.g.*, *Backing Storage* is 128kB).

SAFs specification describes the SAFs applied to the storage or compute levels and the relevant attributes associated with each SAF (*e.g.*, Fig. 6 specifies skipping at *Buffer*, with A as the leader and B as the follower).

Mapping describes an exact schedule for processing the workload on the architecture. It is represented by a set of loops [24]. Each iteration of the *for* loop represents a time step, and the iterations in a *parallel-for* loop represent operations happen simultaneously at different spatial instances (*e.g.*, $n1s$ loop shows that different columns of B are simultaneously processed in four *Buffer*s).

Mapspace Constraints describes a set of constraints on allowed schedule (*e.g.*, allowed loop orders). Sparseloop then explores the potential mappings that satisfy the provided partial loops and locates the best one for a specific workload.

B. Step One: Dataflow Modeling

Dataflow modeling derives the uncompressed data movement and dense compute, which we refer to as the *dense traffic*. Such dense analysis has been studied in several existing works [24], [32], [29], [47], [31]. Since each modeling step in Sparseloop is well-abstracted, various strategies can be plugged into Sparseloop's modeling process. In our implementation, we adopt Timeloop's [24] strategy.

Dataflow modeling is performed based on an abstract architecture topology (*e.g.*, Fig. 7a shows the abstract representation of the architecture in Fig. 6), workload tensors' shapes, and the specified mapping. According to the mapping, each workload tensor is hierarchically partitioned into smaller tiles based on coordinates, with each tile stored in a specific storage level, and this process is referred to as *coordinate-space tiling* [40]. For example, in Fig. 7a, at *L1*, the tensor A in Fig. 6 is partitioned into four tiles (with different shades of blue) based on the *m1 for loop* in the mapping, each of which is a row of the tensor. Each tile is then sequentially sent to *L0*. To derive the data movement for each storage level, dataflow modeling analyzes the stationarity of the tiles and the amount of data transferred, both temporally and spatially, between consecutive tiles. The number of computes is derived based on the input tensor algorithm. More detailed description of dense traffic calculations can be found in Timeloop [24].

C. Step Two: Sparse Modeling

The sparse modeling step is responsible for reflecting the overhead and savings introduced by various SAFs. As shown in Fig. 8, sparse modeling first evaluates the impact of SAFs locally on per-tile traffic with SAF-specific analyzers, *i.e.*, the *Gating/Skipping Analyzer* and the *Format Analyzer*, and then post processes the local traffic with simple scaling to reflect SAFs' impact on overall traffic.

Such decomposition of local and global traffic analysis allows sparse modeling to reflect SAFs' impact *on top of* the dense traffic to produce *sparse traffic* for storage and compute units. We now discuss how each module in Fig. 8 interacts with others and the insight behind this design.

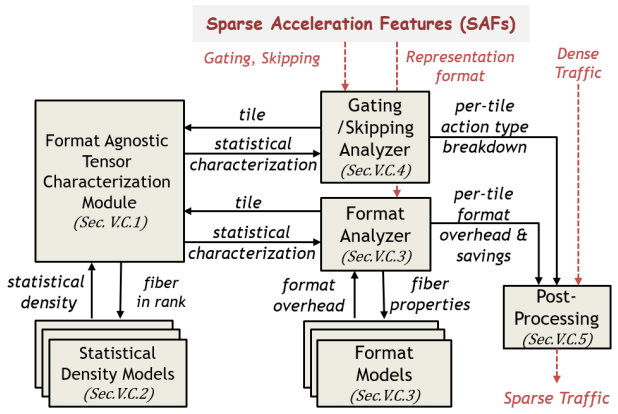


Figure 8: Various modules in sparse modeling step. Hashed red arrows refer to inputs and outputs of this step. The modules are labeled with their corresponding section numbers.

1) ***Format-Agnostic Tensor Description****:* As shown in Fig. 8, to allow tractable complexity and extendability, sparse modeling performs decoupled analysis of SAFs with different analyzers. Describing sparsity characteristics independent of representation format is core to performing such decoupled analysis. We adopt the *fibertree* concept [40] to achieve format-agnostic tensor description. In Fig. 7b, we present the fibertree representation of the sparse tensor A stored in *L1* of Fig. 7a. With the example, we first introduce the key fibertree concepts relevant to Sparseloop.

In fibertree terminology, each dimension of a tensor is called a *rank*[5] and is named. Thus this 2D tensor has 2 ranks, with the rows being named M (rank1), and columns being named K (rank0). In Fig. 7b, each level of the tree corresponds to a tensor rank in a specific order. Each rank contains one or more *fibers*, representing the rows or columns of the tensor. Each fiber contains a set of coordinates and their associated *payloads*. For intermediate ranks, the payload is a fiber from a lower rank (*e.g.*, coordinate 0 in rank1 has a fiber in rank0 as its payload); for the lowest rank, the payload is a simple value. By omitting the coordinate for all-zero payloads, *i.e.*, empty elements, a fibertree-based description accurately reflects the tensor's sparsity characteristics (*e.g.*, rank1's fiber having empty coordinate 2 indicates that the third row is all-zero).

Each fiber in the tree corresponds to a *tile* being processed. For example, in Fig. 7, the first tile processed in *L0* corresponds to the first fiber in *Rank K*. Thus, fibertree-based description enables format-agnostic sparsity-dependent analysis: to analyze the tiles of interest, the analyzers can examine the appropriate fibers to obtain sparsity information independent of the tensor's representation format.

2) ***Statistical Density Models****:* Examining every fiber (thus analyzing the behavior of every tile) is too time-consuming for mapspace and design space exploration. To enable faster analysis, Sparseloop performs statistical characterizations of the fibers in the fibertree. As shown in Fig. 8, Sparseloop can use various statistical density models of the workload tensor to derive statistical density for fibers in each rank (*e.g.*, for the example in Fig. 7b, the fibers in rank0 have a density of 50% with a probability of 0.75 and a density of 0% with a probability of 0.25). For a given density model, the derived statistical density can differ significantly across fibers with different shapes (*i.e.*, fibers from different ranks in the tree). For example, Fig. 9 shows the distribution of fiber densities in a tensor with uniformly distributed nonzero values. In a uniform distribution, a tile's shape varies inversely with the deviation in its density.

[5]A rank can also correspond to split or flattened dimensions.

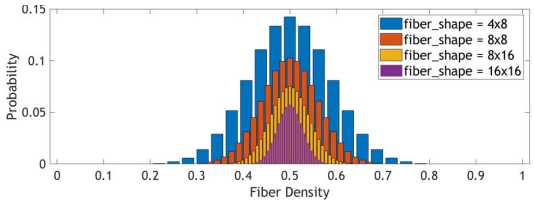


Figure 9: Fiber density probabilities for fibers with various shapes in a tensor with 50% randomly distributed nonzeros.

Density Models	Sparsity Pattern	Example Applications
Fixed structured	Even distribution Coord. independent	Structurally pruned DNNs [18]
Uniform	Random distribution Coord. independent	Randomly pruned DNNs [21] & Activation sparsity
Banded	Diagonal distribution Coord. dependent	SuiteSparse [48] Scientific simulations [49]
Actual Data	Non-statistical Coord. dependent	Graph analytics with special patterns [4]

Table IV: Summary of density models supported by Sparseloop. New models can be easily added via Sparseloop's interface.

To estimate the density of fibers with a given shape, a density model either performs *coordinate-independent* modeling (*i.e.*, fibers at different coordinates have similar density distributions) or *coordinate-dependent* modeling (*i.e.*, fiber's density is a function of its coordinates). Sparseloop supports four popular density models: *fixed-structured, uniform, banded, and actual data*. Table IV describes their properties and use cases in terms of relevant applications (*e.g.*, randomly pruned DNNs [21] and scientific simulations [49]). The modularized implementation of density models ensures Sparseloop's extensibility to modeling of emerging workloads with different nonzero value distributions.

3) ***Format Analyzer****:* With fibers statistically characterized, the format analyzer is responsible for deriving the representation overhead for the tiles stored in different storage levels. Since different tiles correspond to different fibers, it's important for the analyzer to identify the tile in each

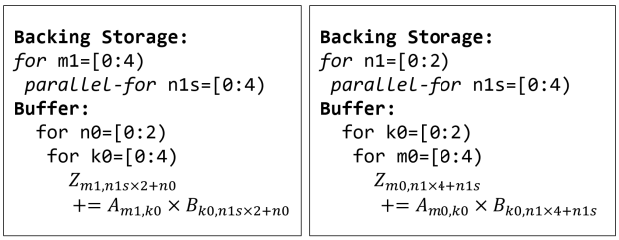


Figure 10: Example mappings that lead to intersections with different impact.

storage and obtain the appropriate statistical characterization of the corresponding fiber from the *format-agnostic tensor characterization module*.

As shown in Fig. 8, for each fiber, the analyzer statistically models the overhead of each rank with the appropriate per-rank *Format Model*. Different formats introduce different amounts of overhead. For example, the ***RLE*** format model calculates the overhead based on the number of non-empty elements in the fiber, $Overhead_{RLE} = \#\,non\text{-}empty\text{-}elements \times run\_length\_bitwidth$; whereas, the bitmask (*B*) format model produces the same overhead regardless of fiber density, $Overhead_B = total\,\#\,elements \times 1$. The statistical format overhead allows Sparseloop to derive important analytical estimations, *e.g.*, the average and worst-case overhead. Sparseloop supports five per-rank format models: *B*, *CP*, *UOP*, *RLE*, and *Uncompressed B*, and thus supports any representation format that can be described with these models. The framework can be easily extended to support other formats.

4) ***Gating/Skipping Analyzer****:* The *Gating/Skipping Analyzer* evaluates the amount of eliminated IneffOps introduced by each gating/skipping SAF. Since gating/skipping focuses on improving efficiency for each tile being transferred and/or each compute being performed, regardless of the total number of operations, the analyzer evaluates the impact of SAFs locally on per-tile traffic and breaks down the original per-tile dense traffic into three fine-grained action types: i) actually happened, ii) are skipped, and iii) are gated.

As discussed in Sec. III, gating/skipping is based on various intersections, which eliminate IneffOps by locating the empty tiles, *i.e.*, tiles with all zeros. In a leader-follower intersection, when the leader tile is empty, the IneffOps associated with the follower are eliminated. Whereas in a double-sided intersection, any tile being empty leads to eliminations of IneffOps associated with the other tile. Since a double-sided intersection can be modeled as a pair of leader-follower intersections ($B \leftrightarrow A = B \leftarrow A + A \leftarrow B$), we focus on discussing the modeling of SAFs based on leader-follower intersections.

The key to modeling the amount of eliminated IneffOps introduced by a SAF based on leader-follower intersection is to correctly identify the associated leader and follower tiles, and thus the fibers representing the tiles in the fibertree. Since different fibers can have significantly different probability of being empty, the same SAF can lead to different impacts. We observe that the leader and follower tiles of a specific intersection can be determined based on the data reuse defined in the mapping. For example, Fig. 10 shows two mappings that lead to different intersection behaviors for $Skip\,B \leftarrow A$ at *Buffer*. In *Mapping* ①, since the innermost $k0$ loop iterates through different pairs of A and B values, a specific $B_{k,n}$ is only used to compute with a single $A_{m,k}$. Thus, the leader tile is a single A value and the follower tile is a single B value, *i.e.*, if $A_{m,k}$ is zero, the access to $B_{k,n}$ will be eliminated. Whereas in *Mapping* ②, since the innermost $m0$ loop only iterates through different A values, a specific $B_{k,n}$ is reused across $A_{0:3,k}$ (*i.e.*, a column of A). Thus, the leader tile is $A_{0:3,k}$ and the follower tile is a single B value $B_{k,n}$, *i.e.*, if the entire column of A is empty, the access to $B_{k,n}$ will be eliminated. Since it is less likely for the entire column of A to be empty, under *Mapping* ②, $Skip\,B \leftarrow A$ eliminates fewer IneffOps (*e.g.*, columns of A are never empty in Fig. 6).

Based on the mapping and the statistical fibertree characterization, the analyzer defines the behaviors of each gating or skipping SAF, and derives a breakdown of the original dense traffic for each tile into fine-grained action types (*e.g.*, for each B tile transferred from *Buffer* to *Compute*, there are 50% skipped reads, 50% actual reads, 0% gated reads). Furthermore, when a gating/skipping SAF is applied to upper storage levels in the architecture, the analyzer propagates the savings introduced to lower levels (*e.g.*, for the architecture in Fig. 6, skipping at *Backing Storage* reduces operations happened at both *Buffer* and *Compute*).

5) ***Traffic Post-processing****:* As shown in Fig. 8, after the analyzers evaluate the impact of their respective SAFs based on per-tile traffic, sparse modeling performs post-processing to first reflect the interactions between the SAFs (*e.g.*, how much format overhead is skipped due to a skipping SAF) and then scale the per-tile breakdowns based on the number of tiles transferred to derive the final sparse traffic.

D. Step Three: Micro-architectural Modeling

Micro-architecture modeling first evaluates the validity of the provided mapping. A mapping is valid only if the largest tiles, which are derived based on statistical tile densities and format overheads, meet the capacity requirement of their corresponding storage levels. If the mapping is valid, micro-architecture modeling evaluates the impact of micro-architecture on generated sparse traffic. The analysis focuses on capturing general micro-architectural characteristics, *e.g.*, segmented block accesses for storage levels, instead of the design-specific micro-architectural analysis, *e.g.*, impact of an exact routing protocol.

Micro-architectural modeling then evaluates the processing speed and energy consumption. For processing speed, cycles are spent for *actual* and *gated* storage accesses and computes. The model considers available bandwidth at each level in the architecture to account for bandwidth throttling. For energy consumption, we use an energy estimation back end (*e.g.*, Accelergy [50]) to evaluate the cost of each fine-grained action, which is combined with its corresponding sparse traffic to derive accurate energy consumption.

VI. Evaluations

We first introduce our experimental methodology and then demonstrate that Sparseloop is fast and accurate.

A. Methodology

Sparseloop is implemented in C++ on top of Timeloop[24], an analytical modeling framework for *dense tensor accelerators*. Sparseloop reuses Timeloop's dataflow analysis, adds the new sparse modeling step, and improves Timeloop's micro-architectural analysis to account for the impact of various fine-grained actions introduced by the SAFs. *As a result, Sparseloop allows modeling of both dense and sparse tensor accelerators in one unified infrastructure*. We use Accelergy [50], [30] as the energy estimation back end. For DNN workloads, Sparseloop performs per-layer evaluations *with the appropriate dataflow and SAFs*, and aggregates the results to derive the energy/latency for the full network. This methodology is consistent with state-of-the-art tensor accelerator modeling frameworks [24], [29], [32], [47]. Experimental results in the next sections are all evaluated on an Intel Xeon Gold 6252 CPU.

B. Simulation Speed

Fast modeling speed allows designers to quickly explore each design's large mapspace as well as various designs. We evaluate Sparseloop's modeling speed with the metric *computes simulated per host cycle* (CPHC), which refers to the number of accelerator computes simulated for each cycle in the host machine that runs the modeling framework. CPHC carries similar information as MIPS (million instructions per second), a popular metric for evaluating simulators for conventional processors.

Detailed cycle-level simulators often have CPHCs that are lower than 1. For example, STONNE [28] has CHPCs that are less than 0.5 when running popular DNN layers with various architecture configurations, *e.g.*, number of rows and columns in the compute array. The main reasons include: i) instead of statistical analysis, cycle-level simulators iterate through actual data to perform all computations, which take significant time for large workloads with millions of computations such as DNNs; ii) detailed control logic needs to be simulated for every cycle and all the components (*e.g.*, memory interface protocols, exact intersection checks).

Accelerator Designs	**Workloads**			
	ResNet50	BERT-base	VGG16	AlexNet
Eyeriss	5.2k	13.3k	53.8k	21.4k
Eyeriss V2 PE	2.7k	12.5k	20.4k	13.2k
SCNN	1.1k	4.3k	3.7k	5.2k

Table V: *Computes simulated per host cycle (CPHC)* for designs modeled by Sparseloop. Compared to cycle-level tensor accelerator simulator STONNE [28], which has less than 0.5 CPHC, Sparseloop is over 2000× faster.

Sparseloop achieves much higher CPHCs with its analytical modeling approach since Sparseloop avoids performing analysis on all computations by performing statistical analysis on transient and steady state design behaviors only; and does not simulate detailed cycle-level control logic. Table V shows Sparseloop's CPHCs for example DNN accelerators [9], [10], [11] running representative workloads [51], [52], [53], [54]. The CPHCs are dependent on accelerator architecture characteristics (*e.g.*, SAFs complexity, number of levels, etc.), employed dataflow, and DNN workload characteristics (*e.g.*, sparsity, tensor shapes, number of layers, etc.). For example, compared to Eyeriss V2 and SCNN, Eyeriss' less powerful SAF support (more details in Table III) always introduces lower SAF modeling complexity and more simulated computes, leading to a higher CPHC. Overall, Sparseloop is over 2000× faster compared to STONNE [28].

C. Validation

High modeling accuracy, in terms of both absolute values and relative trends, allows designers to correctly analyze design trade-offs at an early stage. To demonstrate Sparseloop's accuracy, we validate on five well-known accelerator designs: SCNN [11], Eyeriss [9], Eyeriss V2 [10], and dual-side sparse tensor core (DSTC) [21], and Sparse Tensor Core (STC) [18]. **Overall, Sparseloop maintains relative trends and achieves 0.1% to 8% average error.** Based on available information from existing work, validations are performed on baseline models that capture increasing amount of design details: from analytical models based on statistical sparsity patterns to cycle-level models/real hardware designs based on actual sparsity patterns. At a high-level, common sources of error include: 1) statistical approximation of actual data 2) approximated component characteristics 3) approximated impact of design-specific micro-architectural implementations. Table VI summarizes the validations.

In the next sections, we present more detailed validation discussions. In order to validate our work against prior works, we need to use the workloads reported in those works, despite the reported workloads being old (though popular at the time of the work's publication) or different across designs. This is mainly due to the fact that other workloads are either not available in the reported results or not directly supported by the available simulators.

Accelerator Design	Baseline Model				Average Accuracy	Major Sources of Error
	Source	Type	Sparsity Pattern	Output		
SCNN	Simulators obtained from authors [11], [10]	Design-specific Analytical	Statistical	Runtime activities	99.9%	None
Eyeriss V2 PE			Actual	Processing latency	>98%	Statistical approximations
Eyeriss	Results directly from paper or technical report [9], [21], [18]	Real hardware		Compression rate Energy savings	>95%	(1) Statistical approximations (2) Approximated component energy characterizations
DSTC		Cycle-level simulator validated on silicon [55]		Processing latency	92.4%	(1) Statistical approximations; (2) Optimistic modeling of microarchitectural details
STC		Real hardware		Processing latency	100%	None *(structured sparsity introduces deterministic behaviors)*

Table VI: High-level summary of performed validations based on available data from existing work. Overall, Sparseloop achieves 0.1% to 8% average error across different designs. More details in Sections VI-C1, VI-C2, VI-C3, VI-C4 and VI-C5.

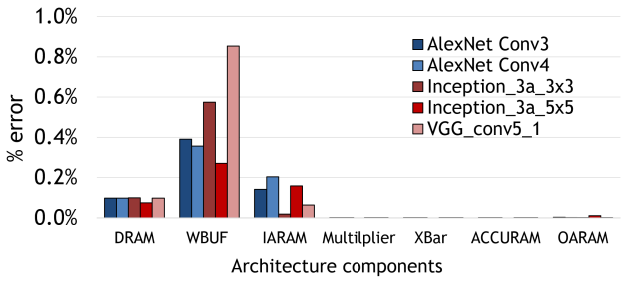


Figure 11: Runtime activity validation for SCNN [11]. Achieves less than 1% error for all components.

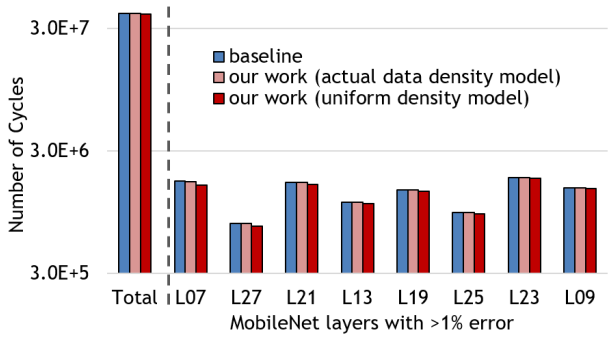


Figure 12: Processing latency validation for Eyeriss V2 processing element [10] running MobileNet [56]. We only show total cycle counts and layers with more than 1% error.

1) SCNN: We first validate Sparseloop on SCNN [11] with a customized simulator that was used in the paper: it performs analytical modeling based on *statistically* characterized data. SCNN baseline assumes uniform distribution and captures the runtime activities of the components in the architecture (*e.g.*, the number of reads and writes to various storage levels). Fig. 11 shows the error rate of the runtime activities for each component in the architecture. Sparseloop , running with a uniform density model, is able to capture all runtime activities accurately with less than 1% error for all components in the architecture.

2) Eyeriss V2 : Since Eyeriss V2's SAFs are implemented in its processing element (PE), we focus on validating the PE based on a baseline model that performs *actual sparsity pattern* based analytical modeling. To quantitatively demonstrate the sources of error, we validate Sparseloop with both an actual-data density model and a uniform density model.

Fig. 12 shows the validation on the number of cycle counts. In terms of total cycle counts for processing the entire MobileNet [56], Sparseloop achieves more than 99% accuracy and is able to capture the relative trends across different layers with both density models. However, for layers with both sparse operands compressed, modeling based on a uniform density model results in up to 7% error for layer27. Fig. 12 shows the layers with more than 1% error. The errors are mainly attributed to the statistical approximation of the expected nonempty intersection ratio between two sparse tensors, as the exact nonempty ratio deviates from case to case, *e.g.*, when both operands have identical nonzero value locations, the intersection nonempty ratio is equal to the tensor densities. With an actual-data density model, Sparseloop accounts for the exact intersections, and thus accurately captures the cycle counts (at the cost of a slower modeling speed).

3) Dual-Side Sparse Tensor Core: For DSTC, the baselines are also obtained directly from the papers whose reported results are based on a cycle-level simulator that is validated on real hardware [55]. We validate on the normalized processing latency running matrix multiplication workloads with various operand tensor density degrees, as shown in Fig. 13. We modeled the tensors with a uniform density model, captured the performance trends across den-

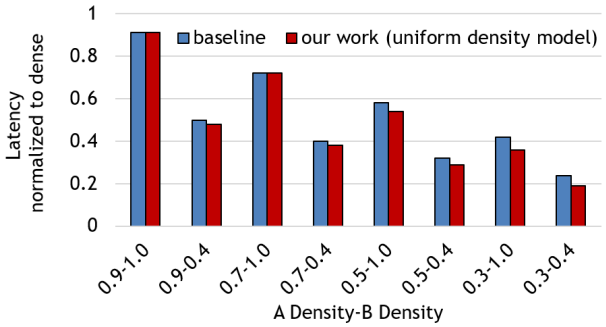


Figure 13: Processing latency of dual-side sparse tensor core [21] running matrix multiplication workloads with various operand tensor densities, normalized to dense processing latency. Average error is 7.6%.

	Conv1	Conv2	Conv3	Conv4	Conv5
Eyeriss[9]	1.2	1.4	1.7	1.8	1.9
Sparseloop	1.2	1.4	1.7	1.9	1.9

Table VII: Eyeriss[9] DRAM compression rate validation.

sity degrees, and obtained an average error of less than 8%. In addition to errors introduced by deviations from the expected nonempty intersection ratio, Sparseloop also performs optimistic modeling of micro-architectural details. More specifically, Sparseloop assumes no storage bank conflicts but DSTC's baseline results contain bank conflicts when operand tensors are relatively sparse (*e.g.*, 30% density), thus introducing higher processing latency.

4) Eyeriss: We validate on Eyeriss [9] with baselines obtained from the paper and based on taped-out silicon. We first validate DRAM compression rates for AlexNet [54], as shown in Table VII. Overall, we achieve 1% error on average and the discrepancy could be due to imperfect compression with the actual data. We also validate on the PE array energy reduction ratio due to on-chip gating. Eyeriss claims that the energy savings of the processing elements can achieve 45%. Our results show a max energy efficiency improvement of 43%. The discrepancy could be due to not modeled PE components with unknown energy characteristics.

5) Sparse Tensor Core: Finally, we validate the Ampere GPU's sparse tensor core accelerator (STC) based on publicly available architecture descriptions [57], [58], [18]. STC focuses on accelerating structured sparse workloads with a 2:4 sparsity structure, which demands at most two nonzero values in every block of four values. Fig. 14 shows the high-level STC architecture and an example processing flow of a 2:4 structured sparse matrix multiplication workload (algorithm defined in Fig. 6). We will discuss more details about STC in Section VII-A.

To validate STC, we use the fixed structured density model parameterized with the 2:4 structure along each channel to model the structured sparse weight tensor. Existing

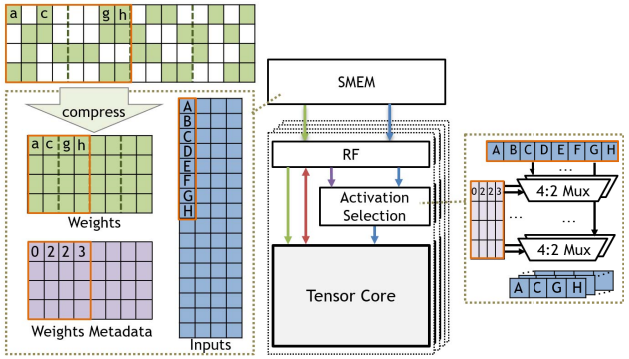


Figure 14: Modeled sparse tensor core architecture (including the *SMEM* in streaming processor for a more holistic view) and example processing of a 2:4 structured workload.

work reports that STC achieves 2× speedup compared to dense processing [18], [57], [58]. Because of the fully defined behaviors with the structured sparsity, Sparseloop also produces an exact 2× speedup (*STC* design in Fig. 15), achieving 100% accuracy.

VII. Case Studies

In this section, we demonstrate Sparseloop's flexibility with two case studies.

A. Investigating Next Generation Sparse Tensor Core

In recent years, various techniques have been proposed to add sparsity support to tensor core (TC). In this case study, we use Sparseloop to first compare two variations: the commercialized NVIDIA STC [18] and a research-based proposal DSTC [21]. Based on the comparison, we then discuss the potential opportunities for next-generation STC, and showcase an example design flow that uses Sparseloop to identify current STC design's limitations and explore various solutions to such limitations to unlock more potential.

1) DSTC vs. NVIDIA STC: We perform an apples-to-apples comparison of the two designs. Since both designs are TC-based, both architectures contain the *SMEM-RF-Compute* hierarchy as shown in Fig. 14, and are controlled on allocated hardware resources, including compute, storage capacity, and memory bandwidth. To model realistic systems, we only provision a subset of a real GPU's *SMEM* bandwidth to the accelerators, since other processes running on the GPU share the same *SMEM* storage. At a high-level, DSTC employs complex sparsity support and a special outer product dataflow to exploit **arbitrary sparsity in both operands** to perform compression and skipping. In contrast, STC ignores input sparsity and uses low-overhead sparsity support to compress and perform skipping **on weights with 2:4 structured sparsity only**.

Fig. 15 compares the cycles spent and energy consumed by DSTC and STC running ResNet50 [51] pruned to various sparsity degrees. ResNet50 contains sparse weights

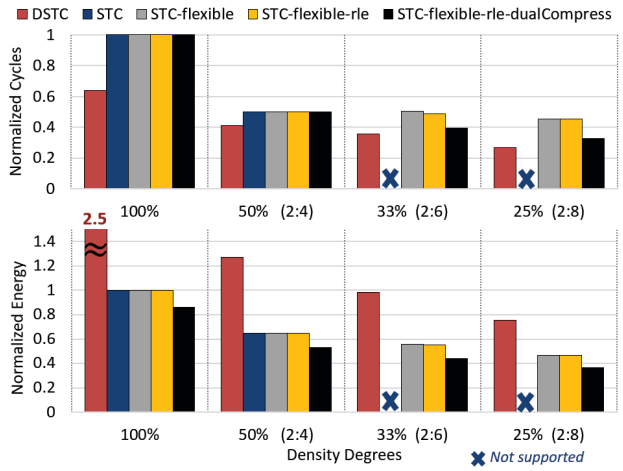


Figure 15: Sparseloop's analysis on the normalized total cycles spent and energy-delay product for various designs of tensor core accelerator running representative ResNet50 [51] layers pruned to various sparsity degrees. The accelerators are controlled to have similar amount of hardware resources.

(if pruned) and sparse inputs. Compared to STC, DSTC's dataflow for supporting arbitrary sparsity incurs a significant amount of data movement. As a result, when processing denser workloads (*e.g.*, unpruned ResNet50 in this example or BERT-like networks with dense input activations), even if DSTC is able to always introduce lower cycles counts, the savings brought by SAFs cannot compensate for the additional energy spent and thus the overall hardware efficiency is low. However, STC provides very limited support for different workloads. Furthermore, for sparser workloads, *e.g.*, 25% dense ResNet50 in Fig. 15(a), despite DSTC's overhead, it's able to achieve a much higher overall efficiency because of the speedup introduced by a significant amount of skipping.

Opportunities for STC: only supporting 2:4 sparsity in the current STC design leads to missed opportunities, as many modern DNNs can be pruned to >50% sparsity (structured [59] or unstructured [60]) while maintaining reasonable accuracy. Thus, one possible feature for a next generation STC to have is to efficiently exploit the savings brought by more sparsity degrees but still keep the sparsity structured to reduce SAF overhead.

2) Naive STC Extension To Support More Ratios: In order to extend the existing STC to support more sparsity degrees, we first introduce the existing high-level processing of STC running matrix multiplication workloads (algorithm defined in Fig. 6) with the default 2:4 structured sparsity. In the case of a DNN, tensor A in Fig. 6 corresponds to the structured sparse weights in Fig. 14.

As shown in Fig. 14, the weight tensor is compressed with an offset-based coordinate payload format, where each nonzero carries an offset coordinate to indicate its position in the block of four values, *e.g.*, the nonzero weight g is the third element in its block, and thus carries a metadata of *2*. This format matches our *CP* format in earlier sections. The compressed weight tensor and the uncompressed input tensor are stored in *SMEM*. For each iteration of processing, the weights, weight metadata, and dense inputs are fetched out. However, since inputs are uncompressed, as shown in Fig. 14, a *tile with four weights corresponds to a tile with eight inputs.* Thus, to ensure correctness, a 4:2 selection needs to be performed on the inputs for each block of four weights. Since only nonzero weights need to be processed, the 2:4 processing is 2× faster than dense processing.

Thus, naively supporting more sparsity degrees in STC simply involves extending the above discussed sparsity support with input activation selection logic for more ratios, *e.g.*, 2:6 and 2:8. We name this naive extension as *STC-flexible*. As shown in Fig. 15, Sparseloop's modeling indicates that *STC-flexible* does support and introduce extra energy reductions for lower density workloads. However, no desirable speedup is obtained with the higher sparsity, *e.g.*, theoretically, 2:6 structured sparsity should introduce 3× speedup. **In fact, surprisingly, the *baseline* processing barely brings any additional speedup with the naive extension for 2:6 and 2:8 workloads.**

3) Identify Design Limitations: STC-flexible's approach does not improve performance due to *SMEM* bandwidth limitation. Fig. 16 shows Sparseloop's analysis on the required bandwidth for processing workloads with various sparsity ratios. To ensure full utilization of the tensor core, the same number of nonzero weights needs to be processed spatially regardless of the workload sparsity, *i.e.*, we always need 1× weights as shown in Fig. 16. As discussed above, STC stores inputs in uncompressed format. Thus, the sparser the weight tensor, the more inputs need to be fetched in a cycle, *e.g.*, in Fig. 16, 4× inputs need to be fetched for workloads with 2:8 sparse weights. In addition to the bandwidth pressure imposed by inputs, the metadata also needs to be described with more bits as the block size gets larger. The amount of additional metadata overhead is dependent on the chosen representation format, *e.g.*, run length encoding (RLE) requires fewer bits than offset-based CP for 2:6 sparse workloads. **As a result, STC is bottlenecked by the limited bandwidth, which is provisioned for 2:4 structured sparsity, and thus cannot obtain the theoretical speedup for sparser workloads.**

4) Explore Solutions to Overcome Limitations: With Sparseloop, we can perform early design stage exploration on potential solutions. Without loss of generality and for the ease of presentation, we discuss two example directions with low-hanging fruit: 1) improve representation format support to reduce metadata overhead; 2) introduce additional compression SAFs for inputs.

First, we evaluate if a different representation (compres-

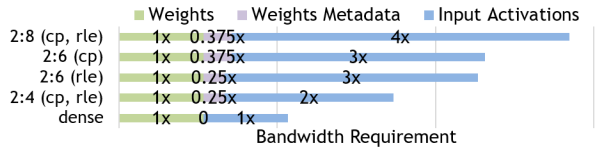


Figure 16: Sparseloop's analysis on bandwidth requirements for getting ideal speedup for various operands and associated metadata (if any).

sion) format can alleviate the overhead introduced by metadata, especially for 2:6 structured sparsity. Thus, as shown in Fig. 15, we enabled *RLE* support for *STC-flexible* to form *STC-flexible-rle*. At a high-level, compared to the STC's original *CP* support, *RLE* support does provide similar or better processing speed. However, since the majority of the overhead comes from transferring actual data, the benefits are too insignificant to bring *STC-flexible-rle* over DSTC.

We then target the more important bottleneck: the uncompressed input data traffic. To solve the problem, we added bitmask-based compression to input such that both operands are compressed to form *STC-flexible-rle-dualCompress* design in Fig. 15. To keep the compute easily synced, we did not add input-based skipping. As a result, all of the obtained speedups come from bandwidth requirement reduction. As shown in Fig. 15, *STC-flexible-rle-dualCompress* can actually introduce similar speed even if it cannot exploit input sparsity for skipping. This is because even if DSTC exploits both operands for speedup, its dataflow has more frequent streaming of operands, introducing additional pressure to *SMEM* bandwidth as well. **Thus, with this example, we have demonstrated that exploiting more sparsity does not guarantee more speedup, and it is very important to make sure the dataflow and SAF overhead is reasonable.**

Overall, as shown in Fig. 15, we derived *STC-flexible-rle-dualCompress* that, compared to DSTC, always introduces lower energy consumption and has similar processing speed most of the time for the studied sparsity degrees.

B. Co-design of Dataflow, SAFs and Sparsity

Looking beyond the deep learning workloads and tensor core accelerators discussed in the previous case study, this section demonstrates how Sparseloop can model workloads with more diverse sparsity degrees and accelerator designs that employ various dataflows and SAFs. With a set of small-scale experiments, we show various broad insights for designing sparse tensor accelerators: (1) the best design for one application domain might not be the best for another; (2) combining more energy or latency saving features together does not always make the design more efficient. Thus, careful co-design of dataflow, SAFs and sparsity is necessary for achieving desired latency/energy savings.

1) Design Choices: **Workloads**: We use matrix multiplication with sparse input tensors (spMspM) of various

(a)

Dataflow Choices	Tensor Reuse		
	A	B	Z
ReuseABZ	Innermost storage	Shared buffer	Innermost storage
ReuseAZ	Innermost storage	None	Innermost storage

(b)

SAFs Choices	Operand Intersection	
	Off-chip	On-chip
InnermostSkip	None	$SkipB \leftrightarrow A$
HierarchicalSkip	$SkipB \leftrightarrow A$	$SkipB \leftrightarrow A$

Table VIII: Choices for different design aspects: (a) dataflows (b) SAFs (representation formats and other minor SAFs are identical and thus are not shown for simplicity)

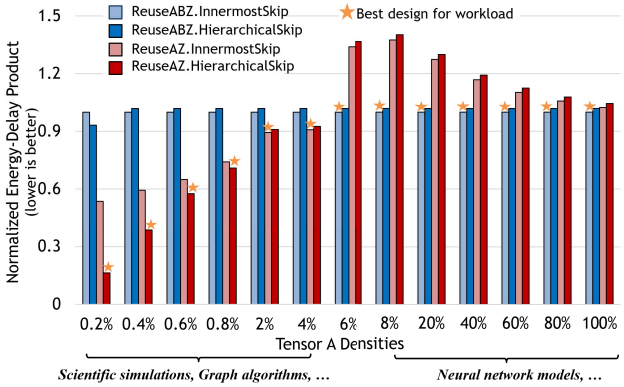


Figure 17: Normalized energy-delay product of different combinations of dataflow-SAFs running matrix multiplications with various density degrees, which are labeled with relevant example workloads. Sparseloop shows (1) dataflow and SAFs should be co-designed to ensure potential savings; (2) the correct combination needs to be chosen for different applications to realize the potential savings.

density degrees as example workloads. spMspM, represented as $Z_{m,n} = \sum_k A_{m,k} \times B_{k,n}$ as an Einsum, is an important kernel in many popular applications, such as scientific simulations, graph algorithms and DNNs, each of which can have different tensor density degrees.

Dataflows: Given a hardware budget of 256 compute units and 128KB on-chip storage, we consider two choices shown in Table VIII(a): (1) *ReuseABZ* that reuses all tensors on-chip; (2) *ReuseAZ* that doesn't have on-chip reuse for B.

SAFs: As shown in Table VIII(b), we consider two sets of SAFs choices: (1) *InnermostSkip* that performs $SkipB \leftrightarrow A$ at the innermost on-chip storage (2) *HierarchicalSkip* that hierarchically performs $SkipB \leftrightarrow A$ at DRAM and innermost storage to reduce both off-chip and on-chip data movement.

2) Interactions Among Design Choices: Fig. 17 compares the energy-delay-product (EDP) of different dataflow-SAF combinations running spMspM with various A tensor density degrees. At each density degree, the EDPs are normalized to *ReuseABZ.InnermostSkip*'s EDP.

We first make the observation that the best design for

one application domain might not be the best for another. For example, while *ReuseABZ.InnermostSkip* is the best design for NN workloads (*i.e.*, A density >6%), for sparser workloads, such as scientific simulations or graph algorithm, this design is sub-optimal due to its large off-chip bandwidth requirement. On the other hand, *ReuseAZ.HierarchicalSkip* performs the best with hyper-sparse workloads since this design performs early off-chip traffic eliminations, but it fails to reduce EDP with NN workloads due to its inability to perform effective off-chip intersections on denser operands and its lack of on-chip B reuse. Thus, a design's dataflow-SAFs combinations need to be chosen based on target application's sparsity characteristics to realize potential savings.

We also show that combining more energy or latency saving features together does not always make the design more efficient. For example, *ReuseABZ.HierarchicalSkip* combines a dataflow that reuses all tensors with SAFs that skip both off-chip and on-chip traffic to form a design with the most number of latency/energy savings features. However, as shown in Fig. 17, *ReuseABZ.HierarchicalSkip* is never the best design in terms of EDP. This is because the *ReuseABZ* dataflow prevents the off-chip skipping SAF from eliminating B's off-chip data movement. More specifically, since *ReuseABZ* reuses each on-chip B tile for multiple A tiles, B tile transfers can be eliminated by the off-chip skipping SAF only when all values in its corresponding A tiles are zeros, which rarely happens. **Thus, dataflow and SAFs need to be carefully co-designed to ensure there exist opportunities for reasonable savings.**

VIII. Related Work

There is ample prior work in modeling frameworks for tensor accelerator designs. These models can be classified into two classes: *cycle-level models* and *analytical models*.

Cycle-level models evaluate the detailed cycle-level behaviors of potential designs. Many of them assume a specific target platform, such as ASIC [61] or FPGA [62], [63], [64] and perform register-transfer-level (RTL) analysis, which includes low-level hardware details (*e.g.*, pipeline stages). There are also platform-independent models, *e.g.*, STONNE[28], which perform cycle-level architectural analysis without RTL implementations. While these cycle-level models are very accurate, they hinder the exploration among the vast number of dataflows due to their long simulation time [22], [23], [24]. Furthermore, cycle-level models are often not well parameterized in terms of architecture topology, employed SAFs, dataflow, etc. These assumptions adversely limit the explorable designs.

Analytical models [24], [65], [27], [30], [29], [31], [32], [33], [34] perform higher level analytical evaluations without considering per-cycle processing details of the design. Since these models work on abstracted hardware models, they are usually well parameterized and modularized to support a wider range of architecture designs. However, to the authors' knowledge, they either do not recognize the sparse workloads or SAFs at all [24], [30], [29], [31], [32], [33], [34], or only target design-specific SAFs [65], [27]. For example, Procrustes[65] supports modeling of B format for one operand only. That is, no prior work aims to flexibly model general sparse tensor accelerators with various SAFs applied. Since at each architecture level, different SAFs can introduce different amounts of savings and overhead, the lack of trade-off analysis for SAFs prevents designers from using such analytical models for design space exploration.

IX. Conclusion

Sparse tensor accelerators are important for efficiently processing many popular workloads. However, the lack of a unified description language and a modeling infrastructure to enable exploration of various designs impedes further advances in this domain. This paper proposes a systematic classification of sparsity-aware acceleration techniques into three high-level sparse acceleration features (SAFs): representation format, gating, and skipping. Exploiting this classification, we develop Sparseloop, an analytical modeling framework for sparse tensor accelerators. We further observe that the analyses of dataflow, SAFs, and micro-architecture are orthogonal to each other. Based on the orthogonality, we design Sparseloop's internal analysis as three decoupled steps to keep its modeling complexity tractable. To balance modeling accuracy and simulation speed, Sparseloop uses statistical characterizations of tensors.

Sparseloop is over 2000× faster than cycle-level simulations, and models well-known sparse tensor accelerators with accurate relative trends and 0.1% to 8% average error. With case studies, we demonstrate that Sparseloop can be used in accelerator design flows to help designers to compare and explore various designs, identify performance bottlenecks (*e.g.*, memory bandwidth), and reveal broad design insights (*e.g.*, co-design of sparsity, SAF and dataflow).

Acknowledgment

We thank Haoquan Zhang for discussions on statistical analysis of tensor density. We would also like to thank the anonymous reviewers for their constructive feedback.

Part of the research was done during Yannan Nellie Wu's internship at NVIDIA Research. This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. This research was also funded in part by DARPA contract HR0011-18-3-0007, NSF PPoSS 2029016 and Ericsson.

APPENDIX

A. Abstract

In this artifact, we provide the source code of Sparseloop, its energy estimation backend based on Accelergy [50], and input specifications to key experimental results presented in the paper. To allow easy reproduction, we provide a docker environment with all necessary dependencies, automated scripts, and a Jupyter notebook that includes detailed instructions on running the evaluations. The artifact can be executed with any X86-64 machine with docker support and more than 10GB of disk space.

B. Artifact check-list (meta-information)

- **Algorithm:** Analytical modeling of sparse tensor accelerator performance (energy and cycles).
- **Program:** C++, python.
- **Run-time environment:** Dockerfile.
- **Hardware:** Any X86-64 machine.
- **Output:** Plots or tables generated from scripts.
- **Experiments:** Analytical modeling of various sparse tensor accelerators running various workloads.
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** Less than 30min if directly pulling docker image; less than 2 hours if building docker from the source.
- **How much time is needed to complete experiments (approximately)?:** Less than 1 hour to finish running all experiments in the provided default mode.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** Yes, DOI 10.5281/zenodo.7027215

C. Description

1) How to access: The artifact is hosted both on github (https://github.com/Accelergy-Project/micro22-sparseloop-artifact) and on an archival repository with DOI 10.5281/zenodo.7027215 (https://doi.org/10.5281/zenodo.7027215).

D. Installation

Since we provide a docker, the installation process mainly involves obtaining the docker image that contains the dependencies, the compiled Sparseloop, and the energy estimation backend. Please follow the provided instructions (https://github.com/Accelergy-Project/micro22-sparseloop-artifact/blob/main/README.md) to obtain and start the docker.

E. Evaluation and expected results

We provide a jupyter notebook in *workspace/2022.micro.artifact/notebook/artifact_evaluations.ipynb* to guide through the evaluations. Please navigate to the notebook in your docker Jupyter notebook file structure GUI.

Each cell in the notebook provides the background, instructions, and commands to run each evaluation with provided scripts. The evaluations include the following key results from the paper:

- Comparison of performance and energy for accelerators supporting different representation formats (Fig. 1).
- Validations on various sparse tensor accelerators (Fig. 12, Table VII, Fig. 13, and the STC design in Fig. 15.)
- Example design flow using Sparseloop to perform apples-to-apples comparison, identify design limitations, and explore various solutions to the limitation (Fig. 15).

The output of each evaluation will either produce a figure or the content of a table. The easiest way to check validity is to compare the generated figure/table with the ones in the paper. However, raw results can also be accessed in the *workspace/evaluation_setups* folder. Please note that we had to use energy estimation data based on public technology node instead of our proprietary technology node, so the exact data might not match for certain evaluation(s). We explicitly point out such cases in the notebook.

F. Experiment customization

The input specifications in the *workspace/evaluation_setups* folder can be updated to specify different hardware setups (*e.g.*, different buffer sizes). Moreover, we also provide options in the scripts to enable map space search using Sparseloop (*e.g.*, *--use_mapper* option can be enabled).

G. Methodology

Submission, reviewing and badging methodology:

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

REFERENCES

[1] M. Zhao, R. Panda, S. Sapatnekar, and D. Blaauw, "Hierarchical analysis of power distribution networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 21, no. 2, pp. 159–168, 2002.

[2] S. Bouaziz, A. Tagliasacchi, and M. Pauly, "Sparse iterative closest point," in *Proceedings of the 11th Eurographics/ACMSIGGRAPH Symposium on Geometry Processing (SGP)*, 2013, pp. 113–123.

[3] C. Ravazzi, R. Tempo, and F. Dabbene, "Learning influence structure in sparse social networks," *IEEE Transactions on Control of Network Systems*, vol. 5, no. 4, pp. 1976–1986, 2018.

[4] N. Henry, J.-D. Fekete, and M. J. McGuffin, "Nodetrix: a hybrid visualization of social networks," *IEEE Transactions on Visualization and Computer Graphics (IEEE Trans Vis Comput Graph)*, vol. 13, no. 6, pp. 1302–1309, 2007.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016, p. 265–283.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2019, pp. 8024–8035.

[7] M. Nikolić, M. Mahmoud, and A. Moshovos, "Characterizing sources of ineffectual computations in deep learning networks," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 86–87.

[8] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 319–333.

[9] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 52, no. 1, pp. 127–138, 2017.

[10] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, vol. 9, no. 2, pp. 292–308, 2019.

[11] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.

[12] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 1–13.

[13] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 766–780.

[14] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 724–736.

[15] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[16] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, p. 687–701.

[17] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 151–165.

[18] NVIDIA, "Nvidia a100 tensor core gpu architecture," NVIDIA, Tech. Rep., 2020.

[19] J.-W. Jang, S. Lee, D. Kim, H. Park, A. S. Ardestani, Y. Choi, C. Kim, Y. Kim, H. Yu, H. Abdel-Aziz, J.-S. Park, H. Lee, D. Lee, M. W. Kim, H. Jung, H. Nam, D. Lim, S. Lee, J.-H. Song, S. Kwon, J. Hassoun, S. Lim, and C. Choi, "Sparsity-aware and re-configurable npu architecture for samsung flagship mobile soc," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 15–28.

[20] C. Deng, Y. Sui, S. Liao, X. Qian, and B. Yuan, "Gospa: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1110–1123.

[21] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-side sparse tensor core," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1083–1095.

[22] S.-C. Kao and T. Krishna, "Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm," in *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[23] P. Chatarasi, H. Kwon, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, "Marvel: A data-centric approach for mapping deep learning operators on spatial accelerators," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, pp. 1–26, 2021.

[24] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[25] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[26] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 261–274.

[27] T.-J. Yang, Y.-H. Chen, J. Emer, and V. Sze, "A method to estimate the energy consumption of deep neural networks," in *Proceedings of the 51st Asilomar Conference on Signals, Systems, and Computers (Asilomar)*, 2017, pp. 1916–1920.

[28] F. M. Matrínez, J. L. Abellán, M. E. Acacio, and T. Krishna, "Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators," *IEEE Computer Architecture Letters (CAL)*, no. 01, 2021.

[29] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *Proceedings of the 48th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2021, pp. 554–566.

[30] Y. N. Wu, V. Sze, and J. S. Emer, "An architecture-level energy and area estimator for processing-in-memory accelerator designs," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 116–118.

[31] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 58–68.

[32] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[33] L. Ke, X. He, and X. Zhang, "Nnest: Early-stage design space exploration tool for neural network inference accelerators," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2018, pp. 1–6.

[34] Y. Zhao, C. Li, Y. Wang, P. Xu, Y. Zhang, and Y. Lin, "Dnn-chip predictor: An analytical performance predictor for dnn accelerators with various dataflows and hardware architectures," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2020, pp. 1593–1597.

[35] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[36] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin, "Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 40–50.

[37] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. Herr, C. Hughes, T. Mattson, and P. Dubey, "T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *Proceedings of the 27th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 181–189.

[38] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer. Timeloop code base. [Online]. Available: https://github.com/NVlabs/timeloop

[39] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 2, pp. 123:1–123:30, 2018.

[40] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 2, pp. 1–341, 2020.

[41] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "Taco: A tool to generate tensor algebra kernels," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 943–948.

[42] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, 2011.

[43] The SciPy community. Scipy documentation. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html

[44] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2015, pp. 1–7.

[45] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[46] A. Einstein, "The foundation of the general theory of relativity," *Annalen der Physik*, vol. 354, no. 7, pp. 769–822, 1916.

[47] L. Mei, P. Houshmand, V. Jain, J. S. P. Giraldo, and M. Verhelst, "Zigzag: A memory-centric rapid DNN accelerator design space exploration framework," *arxiv:2007.11360*, 2020.

[48] S. Kolodziej, M. Mahmoudi Aznaveh, M. Bullock, J. David, T. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The suitesparse matrix collection website interface," *Journal of Open Source Software (J. Open Source Softw.)*, vol. 4, pp. 1–4, 2019.

[49] R.-U. Börner, O. G. Ernst, and S. Güttel, "Three-dimensional transient electromagnetic modelling using rational Krylov methods," *Geophysical Journal International (Geophys. J. Int)*, vol. 202, no. 3, pp. 2025–2043, 2015.

[50] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[51] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv: 1512.03385*, 2015.

[52] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[53] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2015.

[54] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, vol. 25, 2012.

[55] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accelsim: An extensible simulation framework for validated gpu modeling," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[56] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv: 1704.04861*, 2017.

[57] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany, "The a100 datacenter gpu and ampere architecture," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 48–50.

[58] W. Sun, A. Li, T. Geng, S. Stuijk, and H. Corporaal, "Dissecting tensor cores via microbenchmarks: Latency, throughput and numerical behaviors," *arXiv: 12206.02874*, 2022.

[59] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 359–371.

[60] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016, pp. 1–14.

[61] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, "Magnet: A modular accelerator generator for neural networks," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[62] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[63] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 575–580.

[64] A. Rahman, S. Oh, J. Lee, and K. Choi, "Design space exploration of fpga accelerators for convolutional neural networks," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017, pp. 1147–1152.

[65] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis, "Procrustes: a dataflow and accelerator for sparse deep neural network training," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 711–724.