Systematic Modeling and Design of Sparse Deep Neural Network Accelerators

by

Yannan Wu

B.S., Cornell University (2017)
S. M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2023

Authored by:	Yannan Wu Department of Electrical Engineering and Computer Science May 17, 2023
Certified by:	Joel S. Emer Professor of Electrical Engineering and Computer Science Thesis Supervisor
Certified by:	Vivienne Sze Associate Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Systematic Modeling and Design of Sparse Deep Neural Network Accelerators

by

Yannan Wu

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Sparse deep neural networks (DNNs) are an important computation kernel in many data and computation-intensive applications (*e.g.*, image classification, speech recognition, and language processing). The sparsity in such kernels has motivated the development of many sparse DNN accelerators. However, despite the abundant existing proposals, there has not been a systematic way to understand, model, and develop various sparse DNN accelerators.

To address these limitations, this thesis first presents a taxonomy of sparsity-related acceleration features to allow a systematic understanding of the sparse DNN accelerator design space. Based on the taxonomy, it proposes Sparseloop, the first analytical modeling tool for fast, accurate, and flexible evaluations of sparse DNN accelerators, enabling early-stage exploration of the large and diverse sparse DNN accelerator design space. Across representative accelerator designs and workloads, Sparseloop achieves over 2000× faster modeling speed than cycle-level simulations, maintains relative performance trends, and achieves $\leq 8\%$ average modeling error.

Employing Sparseloop, this thesis studies the design space and presents HighLight, an efficient and flexible sparse DNN accelerator. Specifically, HighLight accelerates DNNs with a novel sparsity pattern, called hierarchical structured sparsity, with the key insight that we can efficiently accelerate diverse degrees of sparsity (including dense) by having them hierarchically composed of simple sparsity patterns. Compared to existing works, HighLight achieves a geomean of upto 6.4× better energy-delay product (EDP) across workloads with diverse sparsity degrees, and always sits on the EDP-accuracy Pareto frontier for representative DNNs.

Thesis Supervisor: Joel S. Emer
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Vivienne Sze
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to express my deepest appreciation to my advisors Professor Joel Emer and Professor Vivienne Sze for their support throughout my Ph.D. journey. Without them, I will never be in the place I am today. I really appreciate their help in preparing me to become a better researcher in every aspect. There have been many challenges during the past few years, and no matter how hard the problem was, they have always had trust in me and encouraged me to try again. In addition, I really appreciate the many opportunities they gave me to stand in front of different audiences and sell my research. Such exposures have really helped me learn to effectively communicate with people, which is a skill that I will benefit from for the rest of my career. It has been a truly rewarding experience to be their Ph.D. student for the past few years.

I would also like to thank my committee member Professor Daniel Sanchez for taking the time to serve on my committee and providing insightful feedback on my thesis as well as other projects and presentations in the past.

I was fortunate to collaborate with many excellent people during my time at MIT. Thanks Po-An for the great help, insightful suggestions, and constant encouragement for my two major Ph.D. projects. Thanks Angshu for teaching me the Timeloop logistics. Thanks Saurav, Fisher, and Michael for the many interesting discussions and paper-writing sessions.

I am grateful to spend the past years with the awesome EEMS group members: Amr, Yu-Hsin, Zhengdong, James, Tien-Ju, Hsin-Yu, Gladynel, Peter, Soumya, Jamie, Keshav, Tanner, Fisher, Zih-Sing, Dasong, Michael, and Andy. Thanks for all the chats about the ups and downs in life, research, and literally anything. They are definitely among the most crucial ones to help maintain my mental health during the most stressful times.

Thank you to all my MIT and non-MIT friends for all the fun outings, dinner gatherings, and random chats on various topics. Those have been really refreshing experiences and really helped me to still be open to learning other things that are not that closely related to the small circle I live in day-to-day.

I would also like to thank Janice for helping with all the lab logistics, allowing our coffee hours and lab outings always go smoothly.

I would like to recognize my sponsors for supporting my research.

I could not have undertaken this journey without the constant support from my husband Haoquan. Thanks for being there from literally my first day of class at MIT. I really appreciate the time he spent listening to my random complaints, and always being the one to come up with solutions to my problems. Thanks for teaching me to always stay positive and enjoy life. I'm sure we together can make our life better and better in the future.

Last but not least, a special thank you to my father Zeqing and mother Jiping. Despite the long distance and the twelve-hour difference, they have always been there for me. Thanks for always encouraging me to push on and offering help in every aspect.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep neural networks (DNNs) have brought important accuracy breakthroughs for modern artificial intelligence applications [62] (*e.g.*, image classification [36, 58], speech recognition [26], and language processing [84, 102]). However, when being processed on hardware, DNNs often introduce high latency and energy consumption due to their data and computation-intensive nature. To efficiently process DNNs, computer architects have proposed a variety of DNN accelerators, which focus on exploiting DNN-specific properties (*e.g.*, high data reuse) to improve hardware efficiency [13, 18, 87, 86, 79, 11]. Nowadays, DNN accelerators have become indispensable parts of both cloud and edge computing systems (*e.g.*, Google TPU [51], NVIDIA Tensor Core [74], Apple Neural Engine [71], and Tesla Dojo [95]).

Pioneering the development of next-generation DNN accelerators, researchers have identified another promising characteristic in modern DNNs: they often involve sparse tensors (*i.e.*, multi-dimension arrays that contain various percentages of zeros). This sparsity arises as a result of the complex interactions between various DNN optimization techniques (*e.g.*, the pruning of weight filters that clip certain weights to zeros [34] and the use of activation functions that sets certain activations to zero [1, 42, 69]). Such sparsity in DNNs introduces a significant number of *ineffectual computations* (*i.e.*, $X \times 0 = 0$ and $X + 0 = X$) which can be eliminated to enable further hardware efficiency improvements [39]. As a result, in recent years, computer architects have proposed DNN accelerators with diverse sparsity-related optimization techniques to reduce hardware operations (*e.g.*, memory accesses) associated with ineffectual computations. We refer to these accelerators as *sparse DNN accelerators*.

The rapid growth of sparse DNN accelerator research makes it increasingly important for computer architects to clearly understand the contributions of existing accelerator designs. In addition, it is often desirable for researchers to find a systematic way to develop novel designs that outperform existing work and have great potential for production. This thesis aims to achieve such goals by proposing: (i) sparse tensor accelerator modeling approaches, which enable systematic understanding and rapid exploration of sparse tensor accelerators; (ii) efficient and flexible sparse DNN accelerator designs, which are evaluated via the proposed modeling approaches.

1.1 Challenges and Limitations

We observe multiple challenges to more systematically approaching next-generation sparse DNN accelerator design. In this section, we discuss such challenges and briefly introduce the limitations of existing works.

1.1.1 The Need for A Well-Organized Design Space

In order to discover novel designs that meet our goals, it is important for researchers to first have a systematic understanding of the existing sparse DNN accelerator design space. Unfortunately, existing sparse DNN accelerator designs [15, 17, 78, 2, 81, 33, 73, 48, 24, 107, 31] often rely on design-specific terminologies to describe important design decisions. For example, SCNN [78] proposes a *"PlanarTiled-InputStationary-CartesianProduct"* dataflow and *"compressed-sparse-block"* compression format. Compared to dense tensor accelerators, sparse DNN accelerators need to consider many more design choices (*e.g.*, different compression formats, different ways to eliminate computes) and thus often have much more complex architectures. As a result, it becomes challenging for researchers to appreciate the key insights associated with each (set of) design choice(s) and draw a clear picture of the high-level insights, let alone the ability to understand the lower-level sophisticated interactions among the proposed design decisions. Thus, it is desirable for researchers to setup common terminologies for clear and systematic descriptions of the sparse DNN accelerator designs.

1.1.2 The Need for Early-Design-Stage Modeling Tool

In order to perform quantitative analysis to compare existing designs or in search of a better design, it is important for researchers to have a generally applicable modeling framework for evaluations of diverse sparse DNN accelerators. However, existing sparse DNN accelerator designs are often evaluated with design-specific simulators, which are usually hard to modify to represent other implementation choices, not to mention their unavailability to the public. Such a phenomenon makes it extremely challenging for researchers to compare to existing designs, as the simulators often have different assumptions about the hardware (*e.g.*, technology nodes, level of memory sub-system detail, etc.). In addition, design-specific simulators are often implemented with much more detail than necessary for early-stage design space exploration (*e.g.*, RTL simulations [17]). Although such simulations can introduce very accurate results, they often have slow simulation speeds. Such a detailed modeling approach impedes researchers from performing fast early-stage design explorations to discover new designs that can potentially outperform existing works. Thus, it is important for the community to have flexible, fast, and accurate open-source modeling frameworks to enable early design stage evaluation of sparse DNN accelerators.

1.1.3 The Need for Efficient and Flexible Designs

Although domain-specific accelerators are more efficient compared to general-purpose processors (*e.g.*, CPUs), it has often been challenging to design flexible accelerators to support a diverse set of workloads in its target domain. Thus, there is a strong pragmatic need for simultaneously efficient and flexible sparse DNN accelerator designs. Specifically, modern DNN workloads can have tensors that are sometimes dense and sometimes sparse with various sparsity degrees (*i.e.*, discrete values of sparsity). As a result, sparse DNN accelerators should make little assumptions about the sparsity degree of the workloads and implement simple sparsity-related acceleration techniques to provide disproportionate savings. However, existing sparse DNN accelerator designs often trade efficiency for flexibility or vice versa. Specifically, some designs focus on simple hardware acceleration that incurs low overhead but only translates a limited set of sparsity degrees into hardware savings (*e.g.*, the vector sparse tensor core [126] only efficiently accelerates one specific sparsity pattern). On the other hand, other sparse DNN accelerator designs focus on sophisticated hardware acceleration implementations that are very flexible to translate arbitrarily distributed zeros into hardware savings. However, their flexible support for arbitrarily distributed zeros often incurs considerable latency or energy overhead. The significant overhead can hurt hardware performance when there is not enough sparsity (*e.g.*, the dual-side sparse tensor core [107] aims to accelerate arbitrary sparsity patterns, but employs costly data movement). Therefore, there is a great need to develop efficient and flexible designs that are capable of translating a wide range of many sparsity degrees into reductions in energy and/or latency.

1.2 Thesis Contributions

We address the above-mentioned challenges by first proposing a systematic way to classify and describe the sparse DNN accelerator design space. Based on the taxonomy, we present an open-source early-design-stage sparse DNN accelerator modeling framework[1] to enable systematic studies of new design points in the space. Finally, we present an efficient and flexible sparse DNN accelerator that is evaluated with our proposed modeling framework. Specifically, this thesis makes the following contributions.

[1]Sparseloop is available at http://sparseloop.mit.edu/.

1.2.1 Systematic Description of Sparse DNN accelerators

To organize the unstructured design space of sparse DNN accelerators, this thesis first introduces a well-defined abstraction that describes high-level sparse DNN accelerator attributes for effective communications among researchers from different communities. Specifically, we propose a taxonomy to classify the sparsity-aware acceleration techniques into three high-level *sparse acceleration features (SAFs)*. The proposed taxonomy provides a well-defined set of terminologies to qualitatively describe how a sparse DNN accelerator translates sparsity into reductions in energy and/or latency, without getting into implementation details. With the taxonomy, we show that we can easily describe a variety of accelerators in a systematic fashion. This work is discussed in our publications [113, 112] and Chapter 3.

1.2.2 Sparseloop Modeling Tool

Based on a clear understanding of the sparse DNN accelerator design space, we present a systematic modeling tool, Sparseloop, for apples-to-apples comparisons and to get deeper insights into the impact of various sparse DNN accelerator design decisions. To allow easy characterization of either a single specific design or many designs as part of design space exploration, Sparseloop provides the following capabilities:

- High flexibility: Sparseloop can model a diverse range of potential designs with hardware support for different dataflows, compression formats, etc. Specifically, our case studies demonstrate Sparseloop's flexibility to quickly compare and explore diverse designs with different architectures, dataflows, and SAFs running various workloads.
- High speed: Sparseloop produces results quickly with a three-step progressive modeling procedure. 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 very large mapspace. Specifically, Sparseloop's modeling runs more than 2000× faster than a cycle-level simulation.

- High accuracy: Sparseloop produces high-fidelity modeling results in both mapspace and design space exploration. Specifically, Sparseloop maintains relative performance trends and achieves 0.1% to 8% average error in terms of energy and latency across representative DNN accelerators running different DNNs.

This work is discussed in Chapter 4.

1.2.3 Hierarchical Structured Sparsity for DNN Workloads

Numerous previous sparse DNN accelerator designs have proposed introducing various hardware-friendly sparsity patterns into DNNs with the goal to improve hardware performance while maintaining DNN accuracy. However, existing sparsity patterns often lead to undesirable flexibility and efficiency trade-offs in hardware (*e.g.*, hardware designed for a specific sparsity pattern only efficiently translates one sparsity degree into hardware savings). We propose a novel class of sparsity patterns, called *hierarchical structured sparsity (HSS)*. HSS can systematically represent diverse sparsity degrees (including dense) by hierarchically composing simple sparsity patterns. Since simple sparsity patterns can be easily translated into hardware savings with simple hardware, in addition to its flexibility, HSS also enables efficient low-cost hardware acceleration.

Furthermore, to precisely define HSS and distinguish it from prior work, we propose an approach to precisely specify different sparsity patterns, allowing researchers to clearly compare sparsity patterns in a qualitative fashion or in terms of their exact zero-value locations. This work is discussed in Chapter 5.

1.2.4 Efficient and Flexible DNN Acceleration with HSS

Inspired by the concept of HSS, we propose a simultaneously efficient and flexible DNN accelerator, HighLight, that is able to translate diverse sparsity degrees (including dense) into reductions in energy and/or latency with low-overhead hardware. The key insight of our design is that we leverage *the properties of the multiplication of fractions* to: (i) represent diverse structured sparsity degrees and (ii) enable mod-

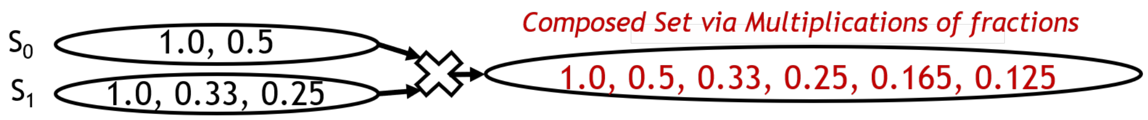


Figure 1-1: Composing two sets of density degrees, S_0 and S_1, by multiplying the fractions in each set.

ularized low-sparsity-tax hardware support for each set of fractions to exploit the structured sparsity degrees. Fig. 1-1 illustrates the idea of multiplication of fractions with two *composable* sets of density degrees (where $density = 1 - sparsity$) represented as fractions. Composing the densities from S_0 and S_1 results in six density degrees. Thus, hardware with modularized support for each set naturally supports all six derived degrees.

Compared to existing works, HighLight achieves a geomean of up to 6.4× better energy-delay product (EDP) across workloads with diverse sparsity degrees (including dense), and always sits on the EDP-accuracy Pareto frontier for representative DNNs. This work is discussed in Chapter 6.

1.3 Related Publications

The contributions of this thesis have resulted in multiple publications and unpublished manuscripts that are either related to modeling methodology or accelerator designs.

Chapter 3 and Chapter 4 are based on three previous publications [112, 113, 109]:

- Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer, *Sparseloop: An Analytical, Energy-Focused Design Space Exploration Methodology for Sparse Tensor Accelerators*, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2021

- Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Joel S. Emer, and Vivienne Sze, *Sparseloop: An Analytical Approach to Sparse Tensor Accelerator Modeling*, in IEEE/ACM International Symposium on Microarchitecture (MICRO), 2022.

- Yannan Nellie Wu, Vivienne Sze, and Joel S. Emer, *An Architecture-Level Energy and Area Estimator for Processing-In-Memory Accelerator Designs*, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2020

Chapter 5 and Chapter 6 are based on a previous unpublished manuscript [110]:

- Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel S. Emer, *Highlight: Efficient and flexible DNN acceleration with hierarchical structured sparsity*, Under submission, 2023.

There are also publications [106, 32] and unpublished manuscripts [114] that are not included in this thesis. They are done in close collaboration with other colleagues. My main contributions include participating in developing the general modeling flows [106, 32] and proposing the high-level idea of integrating speculative execution into tiling strategies [114].

- Francis Wang, Yannan Nellie Wu, Matthew Woicik, Joel S. Emer, Vivienne Sze, *Architecture-Level Energy Estimation for Heterogeneous Computing Systems*, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2021.

- Michael Gilbert, Yannan Nellie Wu, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. *Looptree: Enabling exploration of fused-layer dataflow accelerators.* In 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2023.

- Zi Yu Xue, Yannan Nellie Wu, Joel Emer, and Vivienne Sze. *Accelerating sparse tensor algebra by overbooking buffer capacity*, Under submission, 2023.

Chapter 2

Background

2.1 Overview of Tensor Algebra in DNNs

Tensor algebra is one of the most important computation kernels in deep neural networks (DNNs). In this section, we present basic background knowledge for (dense and sparse) tensor algebra kernels and briefly introduce the sources of sparsity.

2.1.1 The Basics

Tensors are multi-dimensional arrays with an arbitrary number of dimensions. Following the conventions in [39, 75], we call a tensor with N dimensions an *N-tensor* for brevity. For example, tensor A in Fig. 2-1 is a 2-tensor with dimensions M (*i.e.*, rows) and K (*i.e.*, columns). Each tensor has multiple *point*s, each of which is associated with a scalar value (*e.g.*, there are $M \times K$ points in tensor A). Each point's location in the tensor can be described with a set of coordinates (*e.g.*, e's location in tensor A can be described with coordinates $m, k = 3, 2$, abbreviated as $A_{3,2}$).

The *shape* of the tensor is defined by the number of distinct coordinates in each dimension (*e.g.*, tensor A has a shape of $M \times K$, where $M = K = 4$). As we have introduced in Chapter 1, many applications involve sparse tensors (*e.g.*, A in Fig. 2-1 is a sparse tensor, with zeros indicated by the blank spaces). Thus, in addition to tensor shapes, we are also interested in characterizing the nonzero values

Term	**Definition**
tensors	multidimensional arrays with an arbitrary number of dimensions
points	scalar values inside a tensor
coordinates	tuples that describe the locations of the points in the tensor
shape	number of distinct coordinates in each dimension
occupancy	number of nonzero values in the tensor
density	percentage of nonzero values in the tensor
sparsity	percentage of zero values in the tensor; 1-density

Table 2.1: Summary of terminologies related to tensor algebra basics.

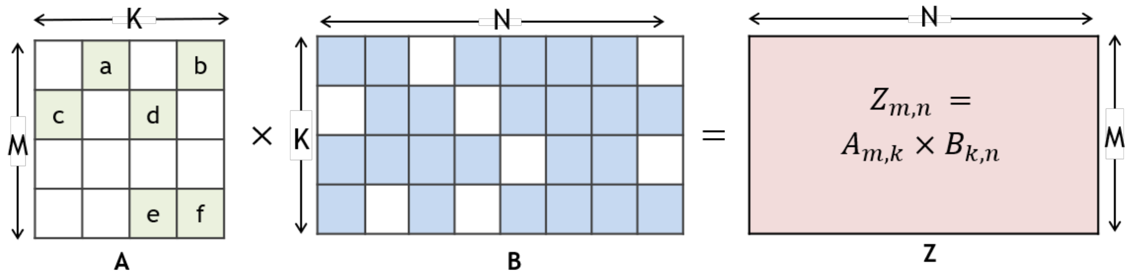


Figure 2-1: Example sparse matrix multiplication kernel, which can be commonly seen in fully connected layers in DNNs. We will use the following color coding throughout the thesis — green: weight tensor; blue: input activation tensor; red: output activation tensor. Blank spaces in the tensors refer to locations with zeros.

in the tensor. We introduce the concept of tensor *occupancy* [94], which indicates the number of nonzero values in the tensor (*e.g.*, tensor A has an occupancy of six with nonzeros *a, b, c, d, e, f*). We call the percentage of nonzero values in a tensor the *density* of the tensor, and the percentage of zero values in a tensor the *sparsity* of the tensor, which equals to $1 - density$. Specifically, DNN layers involve input activation tensors, weight tensors, and output activation tensors, whose shape and sparsity vary from layer to layer within a DNN model and across DNN models. We summarize the introduced terminologies in Table 2.1.

With the tensors described, it is important to precisely and succinctly specify the application's tensor algorithm, which defines the involved mathematical operations. We can describe such algorithms with the well-known Einsum notation [29, 56]. For example, the matrix multiplication kernel in Fig. 2-1 is specified as $Z_{m,n} = A_{m,k} \times B_{k,n}$, where the A and B values along the same k dimension are reduced (*i.e.*, contracted) and the m and n dimensions are populated to the output tensor Z. The reduction is implicit in the Einsum notation without the summation symbol $\sum_k$. Sparse tensor algebra algorithms 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 $X + 0 = X$. For example, for the matrix multiplication kernel in Fig. 2-1, $A_{0,0} \times B_{0,0}$ is one of the ineffectual operations since $A_{0,0} = 0$. Since the Einsum notation does have limitations on the algorithms that it can represent, we employ an extended version of the Einsum notation [39, 94] to express additional computation kernels that are used in DNNs (*e.g.*, convolution). Specifically, the extended Einsum notation allows arithmetic operations on the indices of each tensor (*e.g.*, a one-dimensional convolution can be represented as $O_p = I_{p+r} \times F_r$).

2.1.2 Sparsity Patterns

Sparse DNNs can have sparse tensors with various distributions of zero value locations, which we refer to as *sparsity patterns*. The sparsity patterns could arise due to the nature of the DNNs, or intentionally be introduced into the tensors with the goal of reducing the number of effectual operations and thus improve hardware performance. At a high level, we can classify these sparsity patterns into *unstructured sparsity* or *structured sparsity*.

Unstructured sparsity refers to an unconstrained distribution of zeros (*i.e.*, any point can have a zero). For example, in DNN workloads, unstructured sparsity can naturally arise due to their use of non-linear activation functions [1, 42] that set certain activations to zero. It can also be introduced via unstructured pruning [34, 63, 35], which eliminates unimportant weights by setting them to zero based on their importance regardless of their locations.

By contrast, structured sparsity refers to distributions of zeros with spatial constraints (*i.e.*, only certain points can have zeros). In DNN workloads, structured sparsity is often introduced via *structured pruning* [37, 68, 72, 64], which eliminates weights or activations within certain sub-tensors. For example, one of the most popular DNN structured sparsity patterns is the *G:H sparsity pattern*, which mandates (at most) G elements to be nonzero within a block of H elements, and thus results in a density of G/H. For example, NVIDIA sparse tensor core (STC) [73] employs a 2:4 pattern, which sparsifies two elements in every block of four elements [68], resulting in $2/4 = 50\%$ density and $100\% - 50\% = 50\%$ sparsity.

2.2 Overview of DNN Accelerators

Due to the data and compute-intensive nature of DNNs, it is often inefficient to process them on general-purpose processors (*e.g.*, CPUs). Thus, many DNN accelerators have been designed that exploit DNN's characteristics to improve hardware efficiency). In this section, we discuss the basic concepts related to general (dense and sparse) DNN accelerator design and briefly introduce the high-level opportunities for sparsity-related acceleration in sparse DNN accelerators.

Architecture Organization DNN accelerators are often organized to have multiple levels of storage for efficient on-chip data reuse, flexible networks-on-chip (NoCs) for data communication, and a certain amount of processing elements (PEs) for parallel computation. We also refer to the storage or compute levels as *architecture levels*. Fig. 2-2 shows an example architecture organization with three levels of storage (*i.e.*, *Main Memory, Global Buffer, and Buffer*), multiple PEs with multipliers for parallel computation, and NoCs between storage levels or compute units. For each storage level, the buffers are typically explicitly orchestrated with address generation logic (*e.g.*, the *Cgen* and *Calc*) that are responsible for generating statically configured read and write addresses. Different tensor accelerators can choose to have a different number of architecture levels, different storage capacities at each level, a different number of parallel compute units, etc.

Dataflow and Mapping With the architecture organization defined, it is important for computer architects to efficiently schedule the DNN workloads on the hardware, *i.e.*, find a good workload-hardware *mapping*. Different mappings can result in

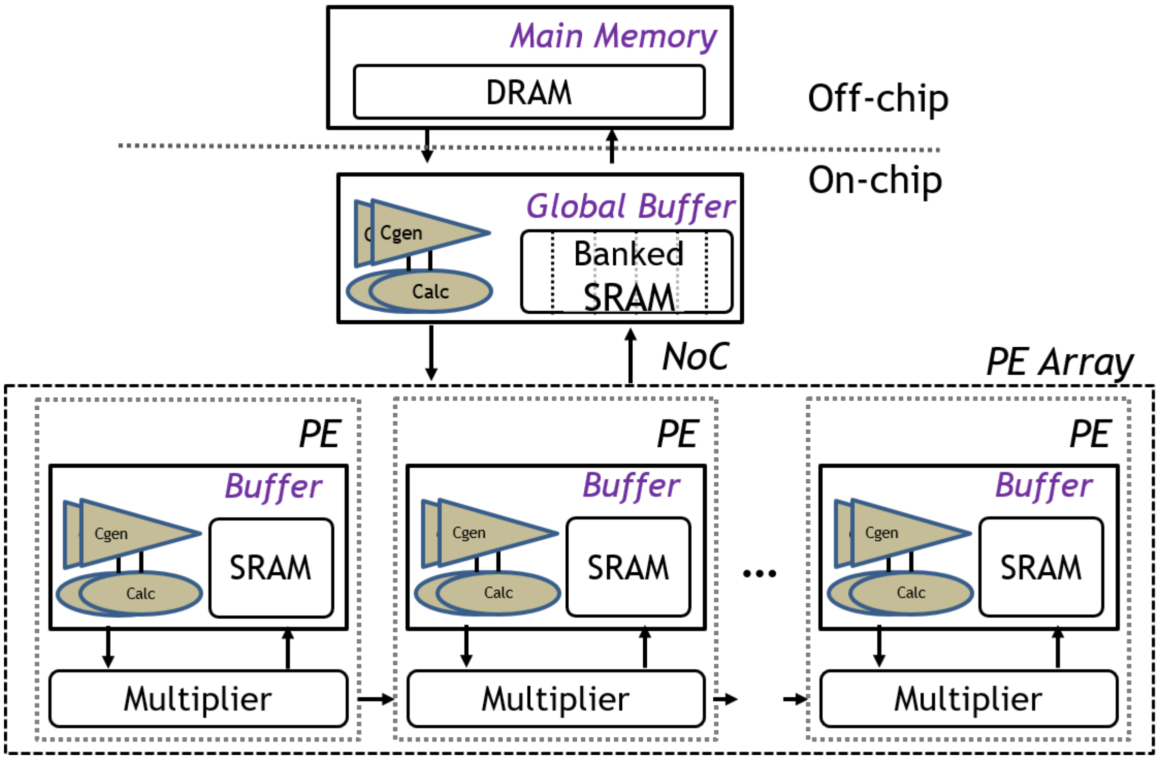


Figure 2-2: Example sparse DNN accelerator architecture organization with three levels of storage (*i.e.*, main memory, global buffer, and buffer), parallel compute (*i.e.*, the multipliers), and network-on-chip (NoC). *PE* stands for processing elements. *Cgen* and *Calc* are representative components for storage access address generation.

drastically different hardware performance (*e.g.*, Timeloop [77] observes that, for a layer in the VGG model [88], there is an 11× difference in energy efficiency when it is being processed with different mappings). A specific accelerator architecture with certain hardware implementation constraints (*e.g.*, storage capacity, NoC flexibility) can only support a specific set of mappings.

A key component of the mappings is their *dataflow* [14, 94]. At a high level, a dataflow specifies the order of data transfers and computes in space and time. For example, an A-stationary dataflow for the example workload in Fig. 2-1 can store A in on-chip *Buffer*s in Fig. 2-2, keep it stationary for reuse, and stream B and Z from off-chip *Main Memory* to on-chip *Buffer*s to perform the computations. A particular accelerator architecture may support one or more dataflows. Please refer to [94, 14, 77] for more detailed discussions of various DNN accelerator dataflows.

To increase data reuse, workload tensors are often partitioned into smaller sub-tensors, which can be stored in smaller buffers in the memory hierarchy to significantly reduce memory traffic [40]. We refer to such the partitioning strategy as *tiling*, and the partitioned sub-tensors as *tiles*. Tiling strategies are also defined by the mappings.

Leveraging Sparsity Unlike dense tensor algebra, which requires the underlying hardware to perform all computations specified by the algorithm, sparse tensor algebra in sparse DNNs provides opportunities for hardware savings with its ineffectual computations. Thus, in addition to considering the hardware organization and dataflow choice, sparse DNN accelerators also pay significant attention to various ways to eliminate hardware operations associated with ineffectual operations, including both the actual arithmetic operations as well as the related data movement across storage levels.

Specifically, based on application characteristics, different sparse DNN accelerators often use different *encoding* to compress sparse tensors (*e.g.*, bitmask encoding) to only store nonzero values for storage capacity savings; and design different hardware that exploits the encoding to eliminate operations associated with ineffectual computations (*e.g.*, intersection units). Hardware designed to exploit unstructured sparse workloads often implements more sophisticated sparsity-related support to exploit the arbitrary distributed nonzero values (*e.g.*, high throughput intersection units, costly workload balancing mechanisms). In contrast, hardware designed for structured sparsity can rely on the defined structure and thus introduce less hardware complexity.

2.3 Overview of DNN Accelerator Modeling

Modeling tools play an important role in facilitating researchers to get a quantitative understanding of the performance of various designs running different workloads. In this section, we introduce the basic concepts of analytical DNN accelerator modeling, with a focus on the Timeloop-Accelergy analytical modeling framework [77, 108, 109].

2.3.1 Analytical Modeling Basics

Analytical modeling performs higher-level evaluations of the important components (*e.g.*, number of levels in the memory hierarchy and number of parallel compute units) in the accelerator architecture and captures the high-level runtime activities of such components (*e.g.*, number of storage accesses) to produce accurate-enough modeling results. Since these models work on abstracted hardware modules, they are usually well-parameterized and modularized to support a wide range of architecture designs. Thus, analytical models best serve researchers for performing early-stage evaluation and exploration of a wide variety of dataflows or hardware architectures. Since this thesis focuses on topics related to the early-stage modeling and design of sparse DNN accelerators, analytical modeling approaches fit well into the picture.

2.3.2 Timeloop-Accelergy Modeling Framework

To demonstrate how an analytical modeling framework functions, we use the Timeloop-Accelergy framework [77, 108, 109], one of the most widely used analytical **dense** tensor accelerator modeling frameworks, as an example to briefly discuss the modeling flow. Furthermore, our proposed analytical **sparse** DNN accelerator modeling framework in Chapter 4 uses Timeloop-Accelergy as a basis, so this section also serves as background preparation for Chapter 4.

As shown in Fig. 2-3, the framework takes in a tensor algebra kernel specification (*e.g.*, fully connected layers in DNNs can be specified as matrix multiplication kernels) as an Einsum (as described in Sec. 2.1.1) and tensor shapes, the hardware architecture organization specification, and a desired mapping specification to generate the total energy consumption and cycle counts. Internally, the framework consists of two parts: the Timeloop performance analysis frontend [77] and the Accelergy (and its plug-ins) energy/area estimation backend [108].

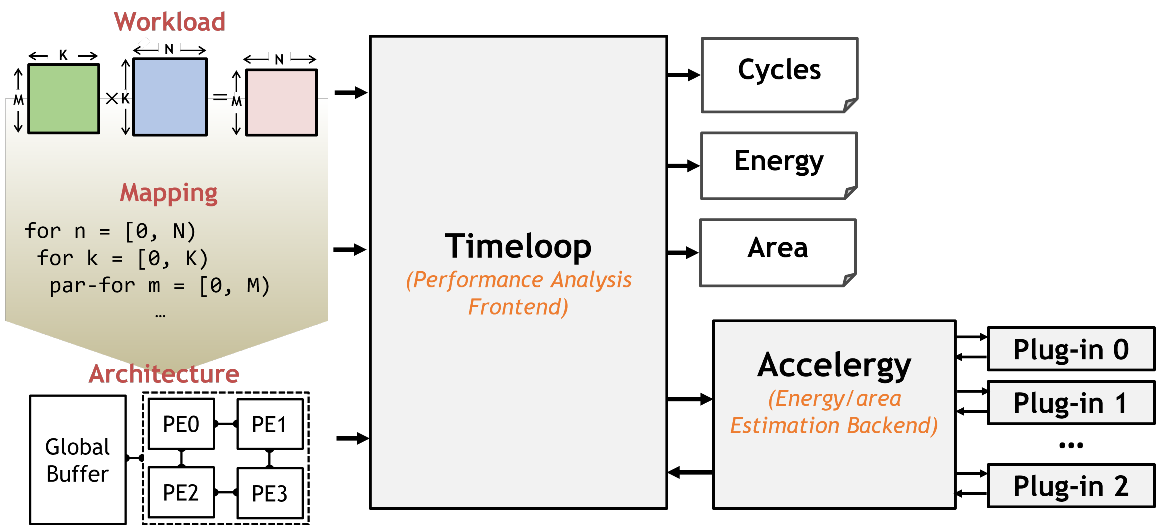


Figure 2-3: The Timeloop-Accelergy analytical modeling framework [109] for dense tensor accelerators. Timeloop [77] serves as a performance modeling frontend, whereas Accelergy [108] serves as an energy and area estimation backend.

Timeloop Frontend

Given the workload and mapping, Timeloop is responsible for analyzing the traffic seen by the components in the architecture (*e.g.*, the number of reads to the *Global Buffer*). To allow fast simulation speed, Timeloop exploits the fact that tensor algebra algorithms induce regular storage access patterns and parallelism (*i.e.*, all the computations, storage accesses, and NoC communication are well-defined by the mapping). As a result, instead of simulating the entire execution flow, Timeloop analyzes only the transient and steady-state behaviors of the system and analytically extrapolates the full runtime activity. Such an analytical approach greatly reduces the amount of simulation time, especially when the workloads involve large tensors.

Timeloop not only supports evaluations of user-defined mappings, it also provide support for automatic mapspace exploration. More details about Timeloop can be found in the paper [77].

Accelergy Backend

On the other hand, Accelergy is responsible for characterizing the per-operation energy costs, and area cost for the components in the provided architecture (*e.g.*, every read to the *Global Buffer* in Fig. 2-2 costs 10pJ). As a result, Accelergy is responsible for more detailed micro-architecture characterization but does not get involved in runtime activity analysis (*i.e.*, Accelergy is oblivious of the workload and mapping). As shown in Fig. 2-3, Accelergy has a flexible interface to communicate with various energy/area estimation plug-ins that can be designed for different types of components, different technology nodes, etc. More details about Accelergy can be found in the papers [108, 109].

Internally, Accelergy receives the architecture specification from Timeloop, performs estimations, and passes the per-operation energy and each component's area cost estimations to Timeloop to generate the final energy and area estimations.

2.3.3 Limitations of Existing Analytical Modeling Approaches

Although there exist various flexible analytical modeling frameworks for tensor accelerators, to the authors' knowledge, existing works are all designed for dense tensor accelerator modeling and either do not provide support for sparse DNN accelerators at all [77, 109, 47, 85, 60, 54, 124], or only provide design-specific knobs for designers to explore [115, 117]. For example, Procrustes [115] supports modeling of one compression format for one operand only. In other words, no prior work aims to flexibly model general sparse DNN accelerators with various sparsity-related accelerations applied. Since at each architecture level, different sparsity-related accelerations can introduce different amounts of savings and overhead, the lack of such trade-off analysis prevents designers from using such analytical models for sparse DNN accelerator design space exploration.

Chapter 3

Design Space Classification

Despite the novelty of proposed sparse DNN accelerator designs, their use of design-specific terminologies is often confusing for computer architects to internalize and derive inspiration from, potentially hindering the evolution of next-generation tensor accelerators. In this chapter, we first discuss the importance of understanding the design space and propose a new classification framework that simplifies how to describe a specific design in a complex design space. We then demonstrate how prior designs can be described in a straightforward manner.

3.1 Motivation

3.1.1 Large and Unstructured Design Space

Sparse DNN 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 leads to a large and diverse design space.

Nonetheless, there is little structure in the design space for sparse DNN 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 the descriptions of their architectures.

For example, Eyeriss [15] uses a *row-stationary* dataflow, *run-length compression* for data stored in DRAM, and storage and compute units that stay idle for IneffOps. With the same dataflow, Eyeriss V2 [17] 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. When cycles must be spent for ineffectual compute, Eyeriss V2 lets the hardware stay idle. SCNN [78] features a *PlanarTiled-InputStationary-CartesianProduct* dataflow to eliminate ineffectual computations without performing expensive intersections and employs *compressed-sparse-block* encoding. ExTensor [39] 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) [107] 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. SparTen [33] employs an *output stationary* dataflow, and introduces *SparseMap* encoding. SparTen proposes its unique *inner join* hardware unit tailored to its dataflow and encoding to eliminate cycles spent on 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.

3.1.2 Sparsity Impacts Design Behavior

Evaluating the complex design space of sparse DNN 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 sparse DNN accelerator de-

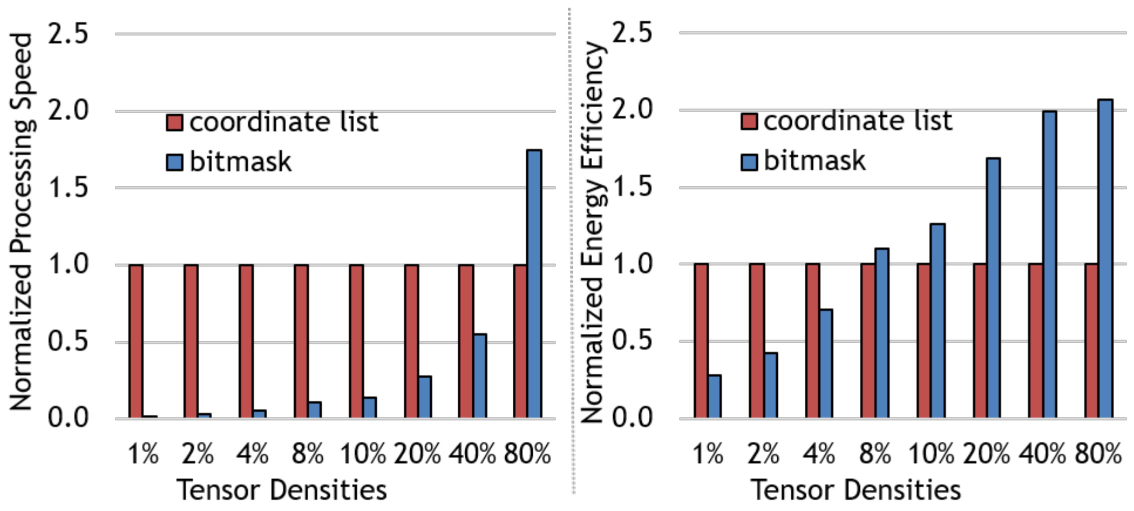


Figure 3-1: Processing speed and energy efficiency of architectures with different data representation support running sparse matrix multiplication workloads. Design behavior is dependent on data representations and tensor densities.

signs supporting different data representations. For simplicity, both accelerators employ the same dataflow:

- 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 save energy, but it does not improve processing speed.[1]

- Coordinate list (SCNN-like): The second design employs a coordinate list encoding [22, 94], 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. 3-1, we compare the processing speed and energy efficiency of the two designs running sparse matrix multiplication workloads of different densities. As shown in Fig. 3-1, the best design choice is a function of the input density. Specifically, since the bitmask-based design does not improve processing speed, with low-density

[1]Of course, there exist other designs that use bitmasks to save both energy and time [33, 120].

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. Thus, for workloads with density $< 4\%$, *coordinate list* has better speedup and energy efficiency; as workload tensors get denser, it gradually loses its advantage. This trend has also been observed in Sigma [81].

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. 4.9.1 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.*

3.2 High-Level Sparse Acceleration Features

To systematically describe sparse DNN 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).

3.2.1 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 [56, 22, 94].

As shown in Fig. 3-2, 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 DNN accelerators tend to employ *compressed formats*, which represent a tensor with only nonzero values and some additional information about their original locations or *coordinate* [56, 22, 94]. We call this information *metadata*. We introduce four per-dimension compressed formats[2].

- *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 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 3.1, full tensor representation formats can be described by combing the per-dimension formats in a hierarchical fashion. For example, CSR

[2]Of course, many more per-dimension formats exist.

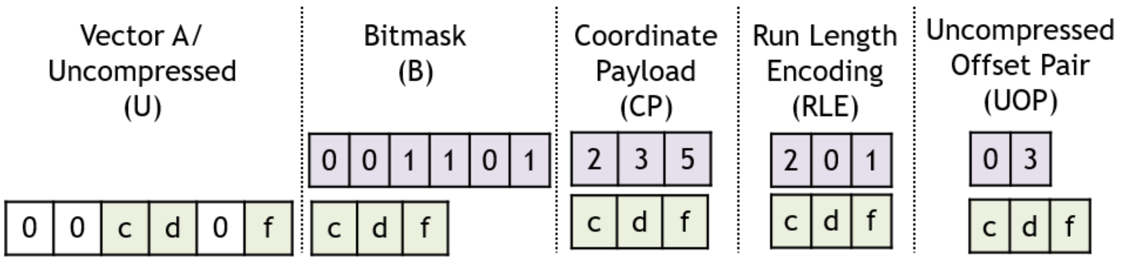


Figure 3-2: Example representation formats of a vector A. Purple vectors refer to metadata used to identify the original locations of the nonzero values.

Example Classic Representation Formats	**Hierarchical Description**
Compressed Sparse Row (CSR) [83]	UOP-CP
2D Coordinate List (COO) [98]	CP^2
Compressed Sparse Block (CSB) [9]	UOP-CP-CP
3D Compressed Sparse Fibers (CSF) [89]	CP-CP-CP

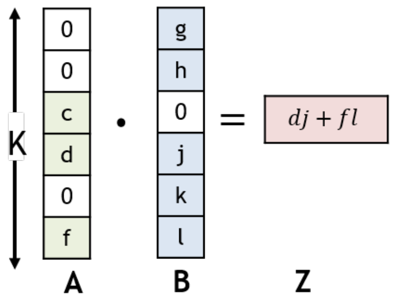
Table 3.1: Example representation formats and their hierarchical description based on per-dimension formats.

(compressed sparse row) [83] 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 partition a tensor dimension into multiple dimensions and/or flatten multiple tensor dimensions into one. For example, 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.

3.2.2 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 with vectors A and B in Fig. 3-3a to illustrate



(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 3-3: (a) Sparse dot product workload. (b) Example ways of processing the example workload. Each row is an example processing schedule, and each column corresponds to a step, and the hardware operations that happen in the cycle are listed. 1st row: baseline dense 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.

the impact of gating. In a dot product, the corresponding entries in the same row of each vector are multiplied together and accumulated into a scalar output (*i.e.*, Z in Fig. 3-3a). At a high level, the ineffectual computations can be detected by intersecting the two operand vectors. When there exists at least one zero in the pair of operands, the computation is ineffectual.

Fig. 3-3b presents the processing steps for the example dot product. Each row 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 the storage accesses and computes are performed regardless of whether the corresponding computation is ineffectual, and takes six steps to complete. The second row in Fig. 3-3b 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. 3-3b shows $Gate\,B \leftarrow A$). The leader-follow intersection 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. 3-3b), as the compute unit can now use the check for the storage unit to power-gate itself.

3.2.3 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. 3-3b shows an example implementation of skipping B reads based on A's values ($\mathit{Skip}\ B \leftarrow A$). 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 [39]). Inefficient implementations can lead to more overhead than savings in time and energy.

3.3 Dataflow Choice is Orthogonal to Sparsity-Aware Acceleration

In addition to the SAFs, dataflow choice is another important decision made by various accelerators [14]. A taxonomy of dataflows for various tensor algebra workloads has already been well-studied in existing work. For example, the DNN dataflow taxonomy proposed in [14, 94] and the matrix multiplication dataflow taxonomy proposed in [119, 90, 76]).

Design	Workload	Format[4]	Gating/Skipping
Eyeriss [15]	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 [17]	DNN	I/W: B-UOP-CP O:U	Innermost Storage : $Skip\,W \leftarrow I$, $Skip\,O \leftarrow I\,\&W$; $Gate\,Compute$
SCNN [78]	DNN	I/W: B-UOP-RLE O: U	Innermost Storage : $Skip\,W \leftarrow I$, $Skip\,O \leftarrow I\,\&W$; $Gate\,Compute$
ExTensor [39]	MM	A/B: UOP-CP×5 Z: U	All Storage : $Skip\,A \leftrightarrow B$, $Skip\,Z \leftarrow A\,\&B$
DSTC [107]	MM	A/B: B-B Z: U	2nd-to-innermost & Innermost Storage : $Skip\,A \leftrightarrow B$, $Skip\,Z \leftarrow A\,\&B$

Table 3.2: 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.

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 number of computes performed (*i.e.*, where and when to perform data movement and compute). As a result, the space of sparse DNN accelerators is the cross product of dataflow choices and SAF choices (further information on how this impacts modeling is in Chapter 4). Of course, a particular dataflow might work well with a specific SAF implementation, leading to an efficient design, while another may not.

3.4 Describing Sparse DNN accelerators

General accelerator designs often implement multiple SAFs that work well with each other to efficiently improve hardware performance. Fig. 3-4 illustrates the idea with a simple example, for the same workload in Fig. 3-3a, Fig. 3-4 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. 3-4 eliminates the compute unit's IneffOps for cases with nonzero A and zero B.

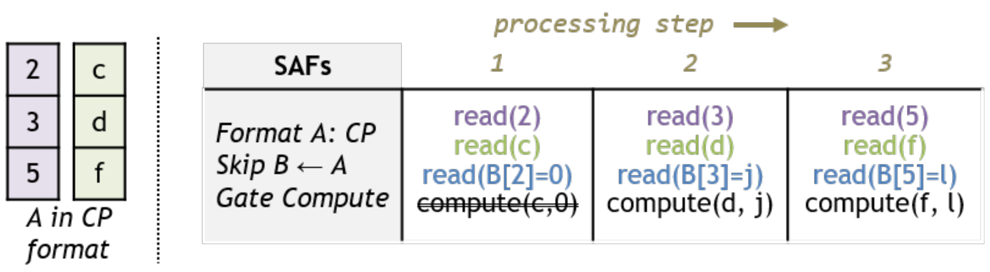


Figure 3-4: Example of combining compressed format, skipping, and gating SAFs in one accelerator design.

Realistic sparse DNN 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 3.2 describes the acceleration techniques of the representative designs introduced in Sec. 3.1.1.

For example, SCNN [78], a sparse DNN accelerator, uses a three-level, B-UOP-RLE representation format[3] to compress input activations (IA) and weights (W). In the innermost storage level (*i.e.*, the level closest to the compute units), SCNN performs $Skip W \leftarrow IA$ 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 the strategy shown in Fig. 3-4. ExTensor [39] 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.

In summary, we have demonstrated how this taxonomy allows the design-specific terminologies in existing proposals to be translated into systematic descriptions. Furthermore, it will also allow future sparse accelerators to be described accurately and compared *qualitatively* in the same way.

[3]Some per-dimension formats are applied to partition or flattened dimensions.

3.5 Conclusions

Sparse DNN accelerators are important for efficiently processing many popular workloads. However, the lack of a unified description language and a modeling infrastructure to enable the exploration of various designs impedes further advances in this domain. This chapter proposes a systematic classification of sparsity-aware acceleration techniques into three high-level sparse acceleration features (SAFs): representation format, gating, and skipping. The proposed taxonomy of various sparse acceleration features provides a well-defined terminology to describe how a sparse DNN accelerator exploits sparsity qualitatively, without getting into implementation details. Thus, we believe the proposed abstraction can facilitate succinct and effective communication between researchers who are developing DNN accelerators.

Chapter 4

Sparseloop

The design space taxonomy in Chapter 3 lays the foundation for developing a general modeling tool for sparse DNN accelerators. In this chapter, we introduce Sparseloop, an analytical modeling framework that *quantitatively* evaluates the processing speed and energy efficiency of sparse DNN accelerators. In this chaper, we will discuss the challenges faced by sparse DNN accelerator modeling and Sparseloop's key methodologies to address those challenges.

4.1 Modeling Challenges

Since workload data characteristics and SAF implementations can have a significant impact on a sparse DNN accelerator's performance, 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 DNN 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 3.2) 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). On the other hand, ignoring workload's sparsity characteristics allows faster modeling but suffers from inaccurate results.

- **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 and core to accurate modeling results, the framework must be flexible and modularized enough to allow easy extensions for future designs/workloads.

4.2 Sparseloop Solutions to the Challenges

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

Based on observation (1), to maintain modeling complexity, Sparseloop performs progressive modeling of distinct design aspects with decoupled modeling steps: (i) Sparseloop evaluates dataflow independent of SAFs, as the storage accesses and com-

[1]A prerequisite for progressive modeling is the target accelerator needs to perform explicit decoupled data orchestration [80], where the data reuse pattern is statically defined. This is often the case for sparse DNN accelerators. However, general sparse tensor accelerators might not always have statically defined data reuse (*e.g.*, accelerators that employ caches introduce dynamic reuse of data).

[2]A prerequisite of having acceptable errors with statistical modeling is that the locations of the nonzero values in the workload tensors can be described with a distribution. Otherwise, an actual data modeling approach that examines the exact data in the tiles needs to be employed. As we will discuss later, Sparseloop does have support for examining actual data, but can come at the cost of orders of magnitude slower modeling speed.

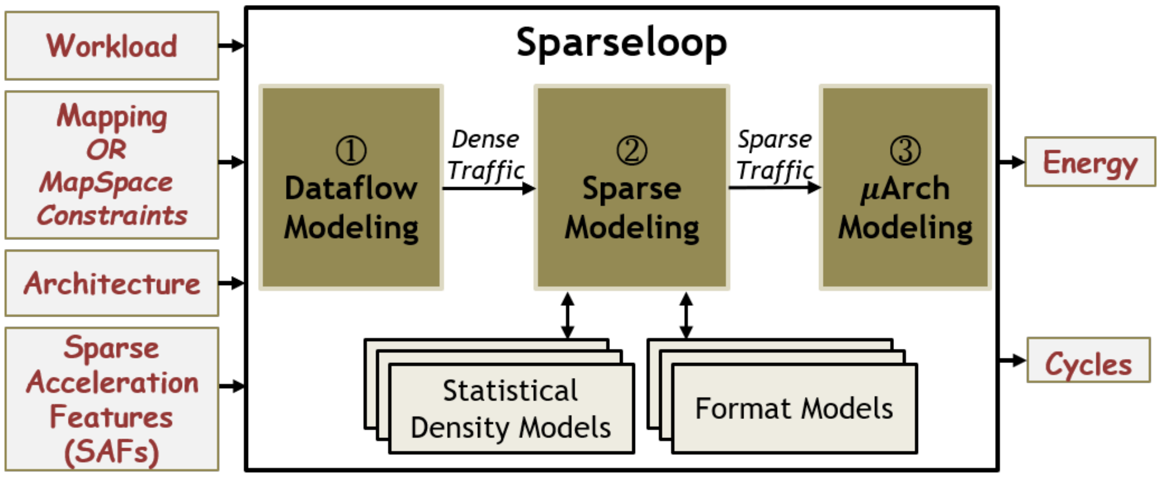


Figure 4-1: Sparseloop High-Level Framework.

putes introduced by the dataflow are irrelevant to how the IneffOps get eliminated; (ii) 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. 4-1 shows, Sparseloop's modeling process is split into three steps, each with tractable complexity.

- ***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 sub-tensors, by leveraging various statistical density models.

Finally, as shown in Fig. 4-1, 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.

4.3 Sparseloop Framework

We first discuss the inputs to Sparseloop in Sec. 4.4, and describe the modeling steps in Sec. 4.5, 4.6, and 4.7.

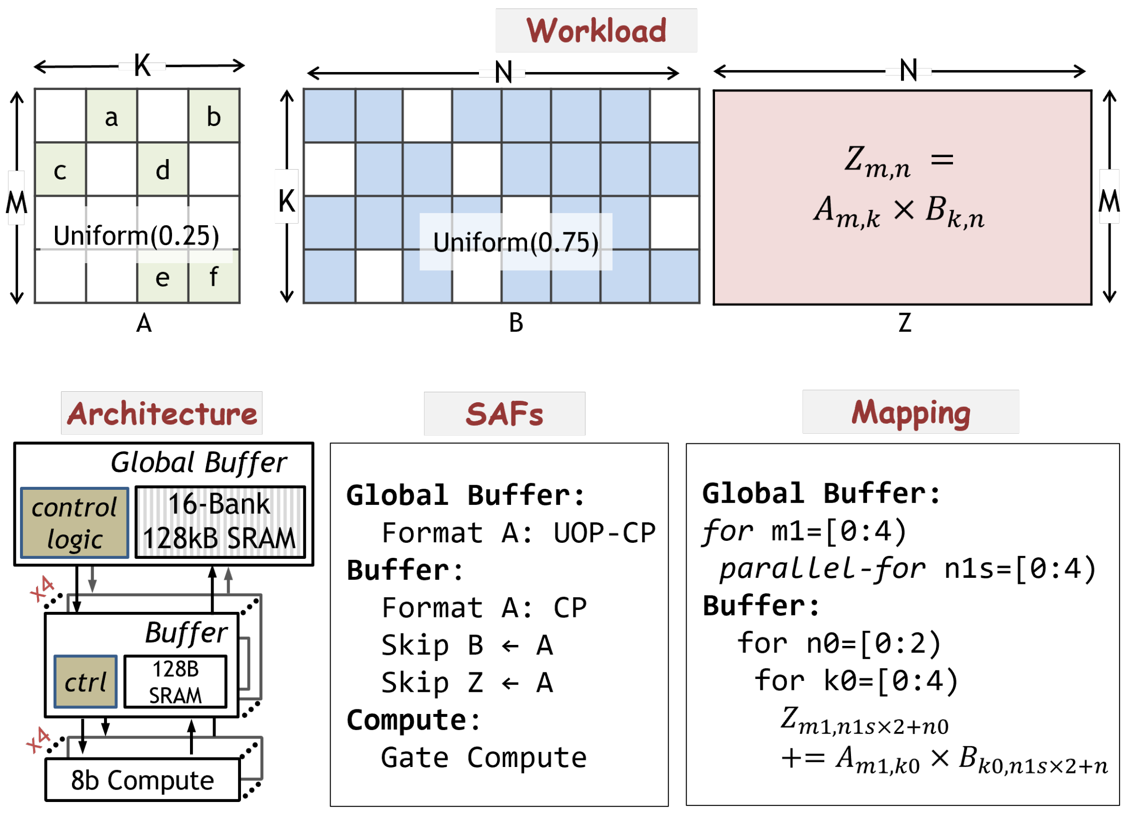
4.4 Inputs

As shown in Fig. 4-1, to generate the processing speed and energy consumption of an accelerator design running a workload, Sparseloop needs four inputs: workload specification, architecture specification, SAFs specification, and mapping or mapspace constraints. Fig. 4-2 shows a set of example specifications to show the input semantics, and more detailed syntax can be found at [111].

Workload specification describes the shape and statistical density characteristics of the workload tensors (*e.g.*, in Fig. 4-2, 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 [29, 56] (*e.g.*, the matrix multiplication kernel is specified as $Z_{m,n} = A_{m,k} \times B_{k,n}$. More explanations in Chapter 2.1.1). Sparseloop understands any algorithm described with an extended Einsum notation, similar to existing works [77, 39].

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.*, *Global Buffer* 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. 4-2 specifies skipping



```
Global Buffer:
  Format A: UOP-CP
Buffer:
  Format A: CP
  Skip B ← A
  Skip Z ← A
Compute:
  Gate Compute
```

```
Global Buffer:
for m1=[0:4)
 parallel-for n1s=[0:4)
Buffer:
  for n0=[0:2)
   for k0=[0:4)
     $Z_{m1,n1s\times2+n0}$
     $+= A_{m1,k0} \times B_{k0,n1s\times2+n}$
```

Figure 4-2: 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.

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 [77]. Each iteration of the *for* loop represents a time step, and the iterations in a *parallel-for* loop represent operations that happen simultaneously at different spatial instances (*e.g.*, $n1s$ loop shows that different columns of B are simultaneously processed in four *Buffers*). Mapping also defines how the tensors are *tiled* for reuse at different levels in the memory hierarchy (recall from Sec. 2.1.1, tiling refers to partitioning the tensors into smaller sub-tensors to fit in buffers of different capacities in the memory hierarchy).

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.

4.5 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 [77, 60, 47, 67, 85]. 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 [77] strategy.

Dataflow modeling is performed based on an abstract architecture topology (*e.g.*, Fig. 4-3(a) shows the abstract representation of the architecture in Fig. 4-2), 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* [94]. For example, in Fig. 4-3(a), at *L1 (Global Buffer)*, the tensor A in Fig. 4-2 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 (Buffer)*. 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 [77].

4.6 Step Two: Sparse Modeling

The sparse modeling step is responsible for reflecting the overhead and savings introduced by various SAFs. As shown in Fig. 4-4, 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 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

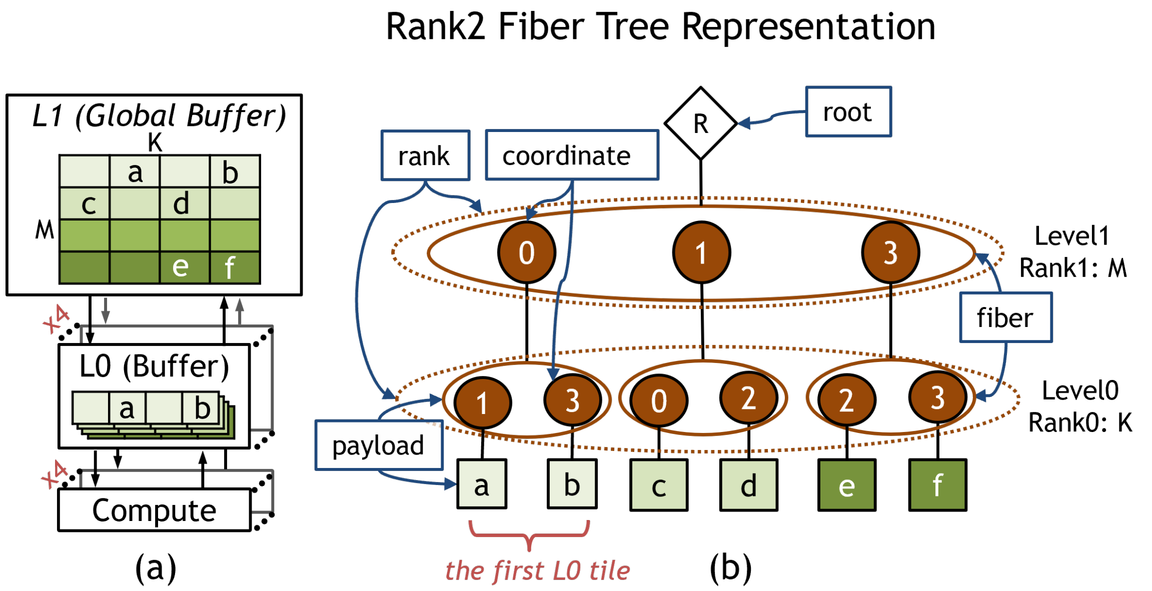


Figure 4-3: (a) Example coordinate-space tiling for tensor A based on inputs specified in Fig. 4-2. The shades represent tiles processed at different time steps. (b) Fiber tree representation of the tensor A. Each level of the tree corresponds to a *rank* of the tensor and contains one or more *fibers* that correspond to the rows or columns of the tensor. The leaves of the tree are the (nonzero) data values of the tensor.

and compute units. We now discuss how each module in Fig. 4-4 interacts with others and the insight behind this design.

4.6.1 Format-Agnostic Tensor Description

As shown in Fig. 4-4, 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 [94] to achieve format-agnostic tensor description. In Fig. 4-3(b), we present the fibertree representation of the sparse tensor *A* stored in *L1 (Global Buffer)* of Fig. 4-3(a). 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*[3] and is named. Thus this 2D tensor has 2 ranks, with *Rank1* being the rows and named *M* (),

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

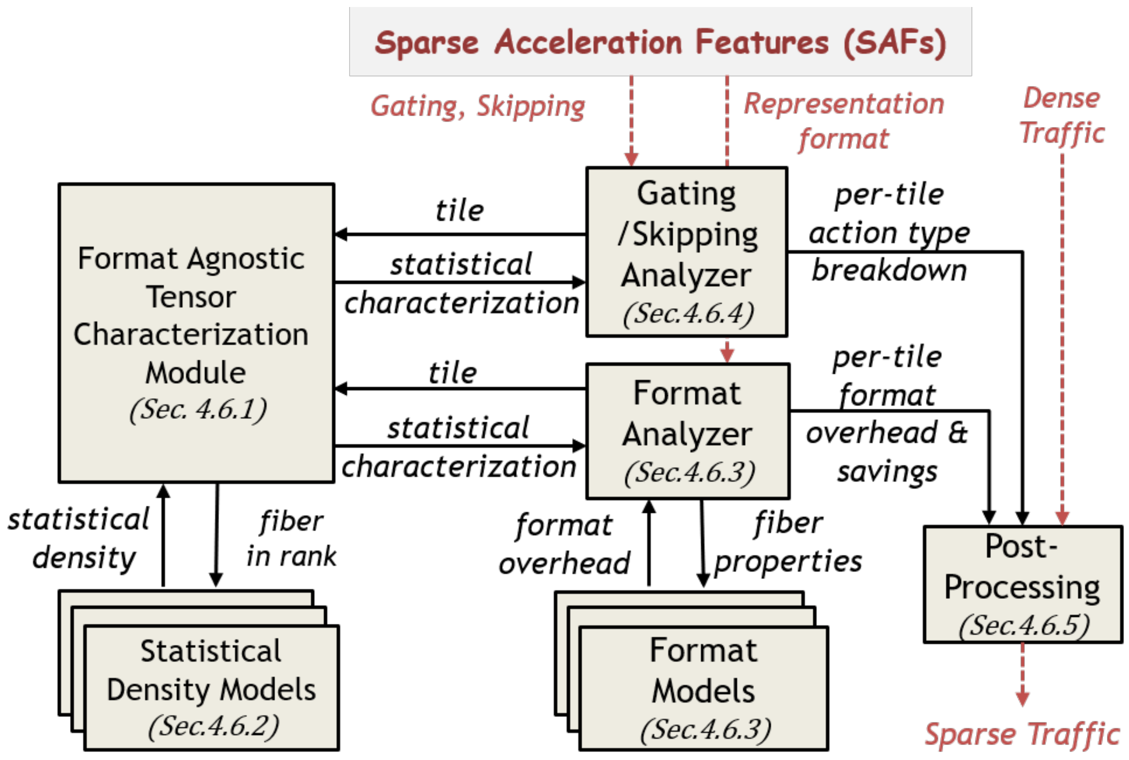

Sparse Acceleration Features (SAFs)
Gating, Skipping
Representation format
Dense Traffic
Format Agnostic Tensor Characterization Module (Sec. 4.6.1)
tile
statistical characterization
Gating /Skipping Analyzer (Sec.4.6.4)
per-tile action type breakdown
tile
statistical characterization
Format Analyzer (Sec.4.6.3)
per-tile format overhead & savings
statistical density
fiber in rank
format overhead
fiber properties
Post-Processing (Sec.4.6.5)
Statistical Density Models (Sec.4.6.2)
Format Models (Sec.4.6.3)
Sparse Traffic

Figure 4-4: 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.

and *Rank1* being the columns and named K. In Fig. 4-3(b), 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. 4-3, the first tile processed in *L0 (Buffer)* corresponds to the first fiber in *Rank0*. Thus, fibertree-based description enables format-agnostic sparsity-dependent analysis: to analyze the tiles of interest, the analyzers can examine the appropriate fibers to

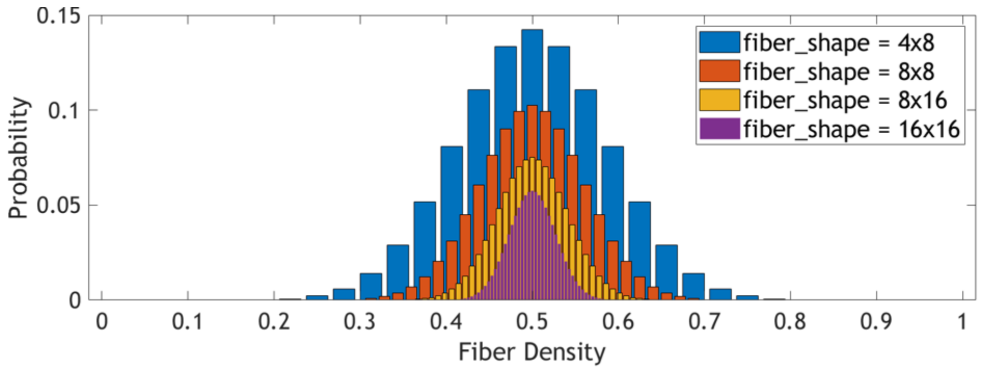


Figure 4-5: Fiber density probabilities for fibers with various shapes in a tensor with 50% randomly distributed nonzeros. Different fiber shapes result in different fiber occupancy distributions.

obtain sparsity information independent of the tensor's representation format.

4.6.2 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. 4-4, 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. 4-3(b), 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. 4-5 shows the distribution of fiber densities in a tensor with uniformly distributed non-zero values. In a uniform distribution, a tile's shape varies inversely with the deviation in its density.

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 4.1 describes their properties and use cases in terms of relevant applications (*e.g.*, randomly pruned DNNs [107]

Density Models	**Sparsity Pattern**	**Example Applications**
Fixed structured	Even distribution, Coord. independent	Structurally pruned DNNs [73]
Uniform	Random distribution, Coord. independent	Randomly pruned DNNs [107], DNN activation sparsity
Banded	Diagonal distribution, Coord. dependent	SuiteSparse [57], Scientific simulations [8]
Actual Data	Non-statistical, Coord. dependent	Graph analytics with special patterns [43]

Table 4.1: Summary of density models supported by Sparseloop. New models can be easily added via Sparseloop's interface.

and scientific simulations [8]). The modularized implementation of density models ensures Sparseloop's extensibility to modeling of emerging workloads with different nonzero value distributions.

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

As shown in Fig. 4-4, 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-empty-elements* $\times$ *run_length_bitwidth*; whereas, the bitmask (B) format model produces the same overhead regardless of fiber density, $Overhead_B$ = $\#$ *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

```
Global Buffer:
for m1=[0:4)
 parallel-for n1s=[0:4)
Buffer:
  for n0=[0:2)
   for k0=[0:4)
     $Z_{m1,n1s\times2+n0}$
       $+= A_{m1,k0} \times B_{k0,n1s\times2+n0}$
```

Mapping ①

```
Global Buffer:
for n1=[0:2)
 parallel-for n1s=[0:4)
Buffer:
  for k0=[0:2)
   for m0=[0:4)
     $Z_{m0,n1\times4+n1s}$
       $+= A_{m0,k0} \times B_{k0,n1\times4+n1s}$
```

Mapping ②

Figure 4-6: Example mappings that lead to intersections with different impact.

supports any representation format that can be described with these models. The framework can be easily extended to support other formats.

4.6.4 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 Chapter 3, 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. In contrast, 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 a 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. 4-6 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). In contrast, in *Mapping 2*, since the innermost $m0$ loop only iterates through different A values, a specific $B_{k,n}$ is reused across $A_{0,k}$, $A_{1,k}$, $A_{2,k}$, and $A_{3,k}$ (*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. 4-2).

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. 4-2, skipping at *Global Buffer* reduces operations happened at both *Buffer* and *Compute*).

4.6.5 Traffic Post-processing

As shown in Fig. 4-4, 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.

4.7 Step Three: Micro-architectural Modeling

As the last step in Sparseloop's three-step evaluation process, 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). The fine-grained sparse traffic was generated based on a minimum access granularity, referred to as *block size*, of one at the sparse modeling step. However, for storage levels with block sizes larger than one, storage access segmentation can happen due to poorly-aligned storage accesses (*e.g.*, 10 reads of a storage unit with a block size of four result in $\lceil 10/4 \rceil = 3$ block accesses, where the last two data fetched in the last block access are invalid). For compressed tiles, the number of block accesses is refined based on the statistical model of tile density.

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 the Accelergy [108] energy estimation back end to evaluate the cost of each fine-grained action, which is combined with its corresponding sparse traffic to derive accurate energy consumption.

4.8 Evaluations

We first introduce our experimental methodology and then demonstrate that Sparseloop is fast and accurate. Please note that we provide an open-source codebase for the evaluations and please refer to Appendix A for more information.

4.8.1 Methodology

Sparseloop is implemented in C++ on top of Timeloop[77], 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 DNN accelerators in one unified infrastructure.* We use Accelergy [108, 109] as the energy estimation back end. Sparseloop is also compatible with Timeloop's mapspace exploration mechanism, which iterates over the potential mappings in the mapspace and selects the mapping with the best hardware performance for a specific accelerator design running a specific workload. *Our implementation allows modeling of both dense and sparse DNN accelerators in one unified infrastructure.*

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 [77, 47, 60, 67]. Experimental results in the next sections are all evaluated on an Intel Xeon Gold 6252 CPU.

4.8.2 Modeling 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 [65] has CHPCs that are less than 0.5 when running popular DNN layers with various architecture configurations (*e.g.*, the number of rows and columns in the compute array). The main reasons include: i) instead of statistical analysis,

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 4.2: *Computes simulated per host cycle (CPHC)* for designs modeled by Sparseloop. Compared to cycle-level tensor accelerator simulator STONNE [65], which has less than 0.5 CPHC, Sparseloop is over 2000× faster.

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).

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 4.2 shows Sparseloop's CPHCs for example DNN accelerators [15, 17, 78] running representative workloads [36, 27, 88, 58]. 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 3.2) always introduces lower SAF modeling complexity and thus shorter modeling time, leading to a higher CPHC. Overall, Sparseloop is over 2000× faster compared to STONNE [65].

4.8.3 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 [78], Eyeriss [15], Eyeriss V2 [17], and dual-side sparse tensor core (DSTC) [107], and

Accelerator Design	**Baseline Model**				**Average Accuracy**	**Major Sources of Error**
	Source	Type	Sparsity Pattern	Output		
SCNN	Simulators obtained from authors [78, 17]	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 [15, 107, 73]	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 4.3: High-level summary of performed validations based on available data from existing work. Overall, Sparseloop achieves 0.1% to 8% average error across different accelerator designs.

Sparse Tensor Core (STC) [73]. **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 an 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; and 3) approximated impact of design-specific micro-architectural implementations. Table 4.3 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.

SCNN

We first validate Sparseloop on SCNN [78] with a customized simulator that was used in the paper: it performs analytical modeling based on *statistically* character-

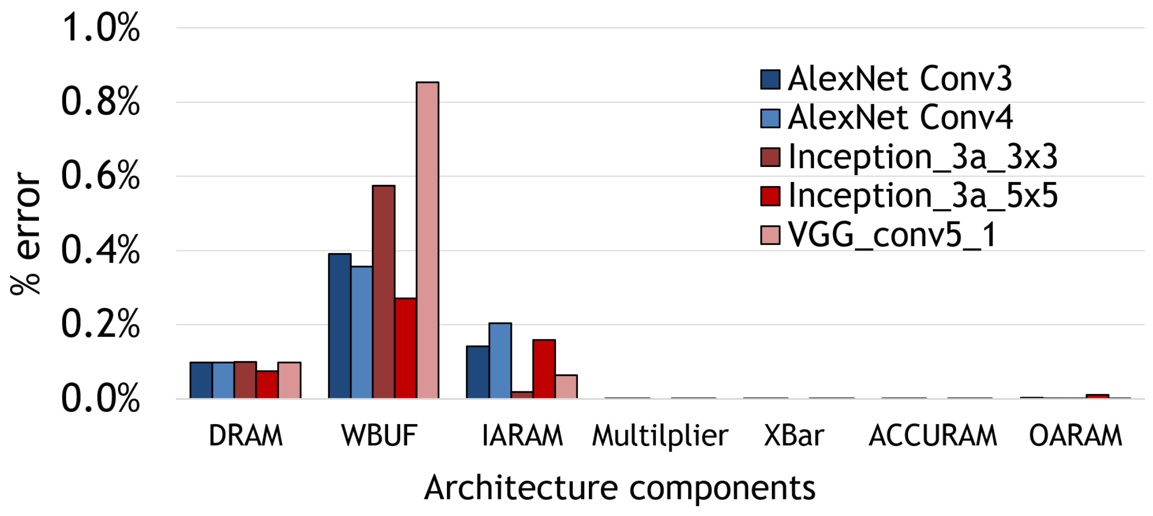


Figure 4-7: Runtime activity validation for SCNN [78]. Achieves less than 1% error for all components.

ized 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. 4-7 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.

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. 4-8 shows the validation on the number of cycle counts. In terms of total cycle counts for processing the entire MobileNet [46], 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. 4-8

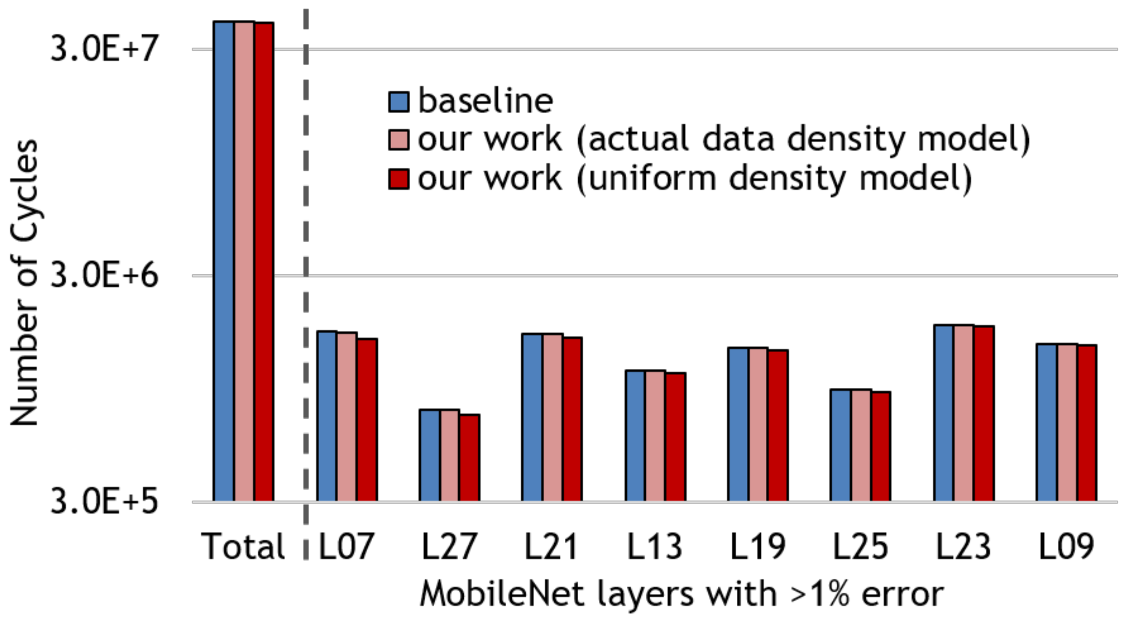


Figure 4-8: Processing latency validation for Eyeriss V2 processing element [17] running MobileNet [46]. Among all the layers, we only show total cycle counts for layers with more than 1% error.

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).

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. 4-9. We modeled the tensors with a uniform density model, captured the performance trends across density degrees, and obtained an average error of less than 8%. In addition

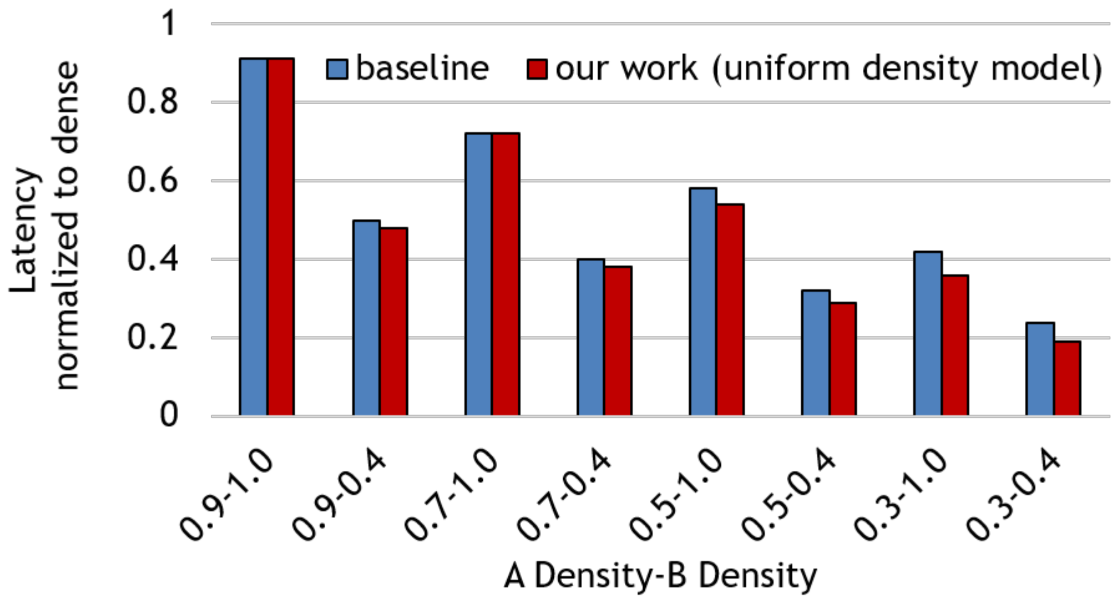


Figure 4-9: Processing latency of dual-side sparse tensor core [107] running matrix multiplication workloads with various operand tensor densities, normalized to dense processing latency. Average error is 7.6%.

to errors introduced by deviations from the expected nonempty intersection ratio, Sparseloop also performs optimistic modeling of micro-architectural details. 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.

Eyeriss

We validate on Eyeriss [15] with baselines obtained from the paper and based on taped-out silicon. We first validate DRAM compression rates for AlexNet [58], as shown in Table 4.4. 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 PE components that were not modeled.

	Conv1	Conv2	Conv3	Conv4	Conv5
Eyeriss[15]	1.2	1.4	1.7	1.8	1.9
Sparseloop	1.2	1.4	1.7	1.9	1.9

Table 4.4: Eyeriss [15] DRAM compression rate validation.

Sparse Tensor Core

Finally, we validate the Ampere GPU's sparse tensor core accelerator (STC) based on publicly available architecture descriptions [21, 91, 73]. 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. 4-10 shows the high-level STC architecture and an example processing flow of a 2:4 structured sparse matrix multiplication workload (algorithm defined in Fig. 4-2). We will discuss more details about STC in Section 4.9.1.

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 work reports that STC achieves 2× speedup compared to dense processing [73, 21, 91]. Because of the fully defined behaviors with the structured sparsity, Sparseloop also produces an exact 2× speedup (*STC* design in Fig. 4-11), achieving 100% accuracy.

4.9 Case Studies

In this section, we demonstrate Sparseloop's flexibility with two case studies. Similary, more information about the open-source code base can be found in Appendix A.

4.9.1 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 [73] and a research-based proposal DSTC [107].

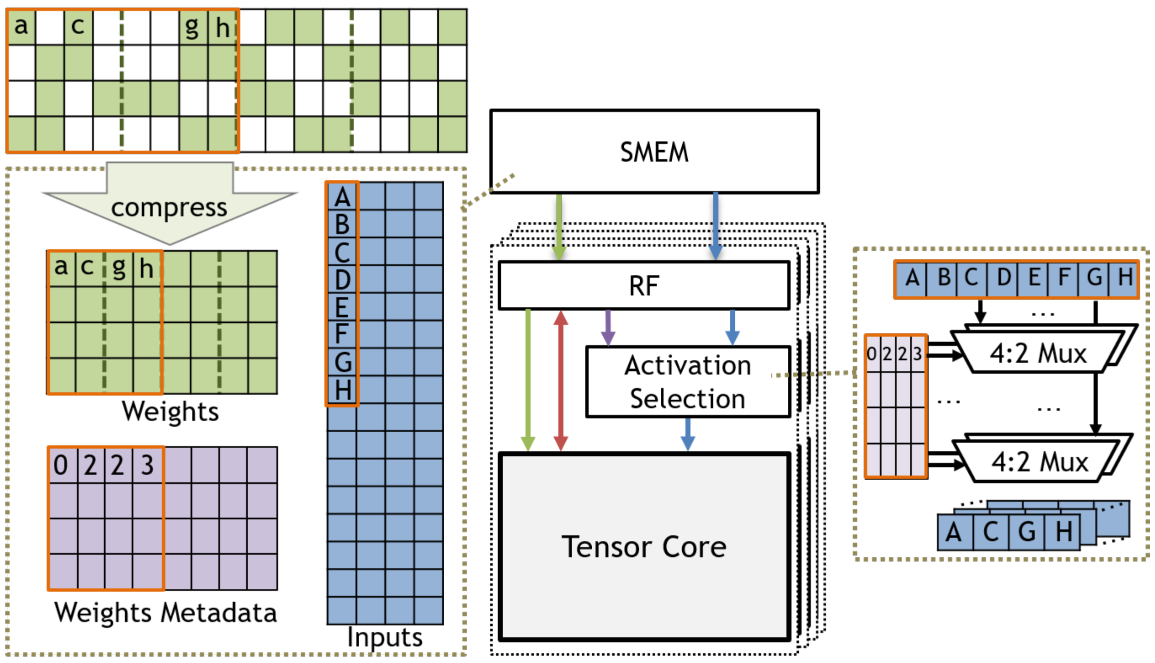


Figure 4-10: Modeled sparse tensor core architecture (including the *SMEM* in streaming processor for a more holistic view) and example processing of a 2:4 structured workload.

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.

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. 4-10, 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. 4-11 compares the cycles spent and energy consumed by DSTC and STC running ResNet50 [36] pruned to various sparsity degrees. ResNet50 contains sparse weights (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. 4-11(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.

Takeaway: STC and DSTC have different limitations, and neither is always able to provide the better efficiency across workloads with different density degrees.

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 [126] or unstructured [34]) 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. Based on such observations, We will discuss some simple extensions to STC in the next sections, and a more involved design in Chapter 6.

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. 4-2) with the default 2:4 structured sparsity. In the

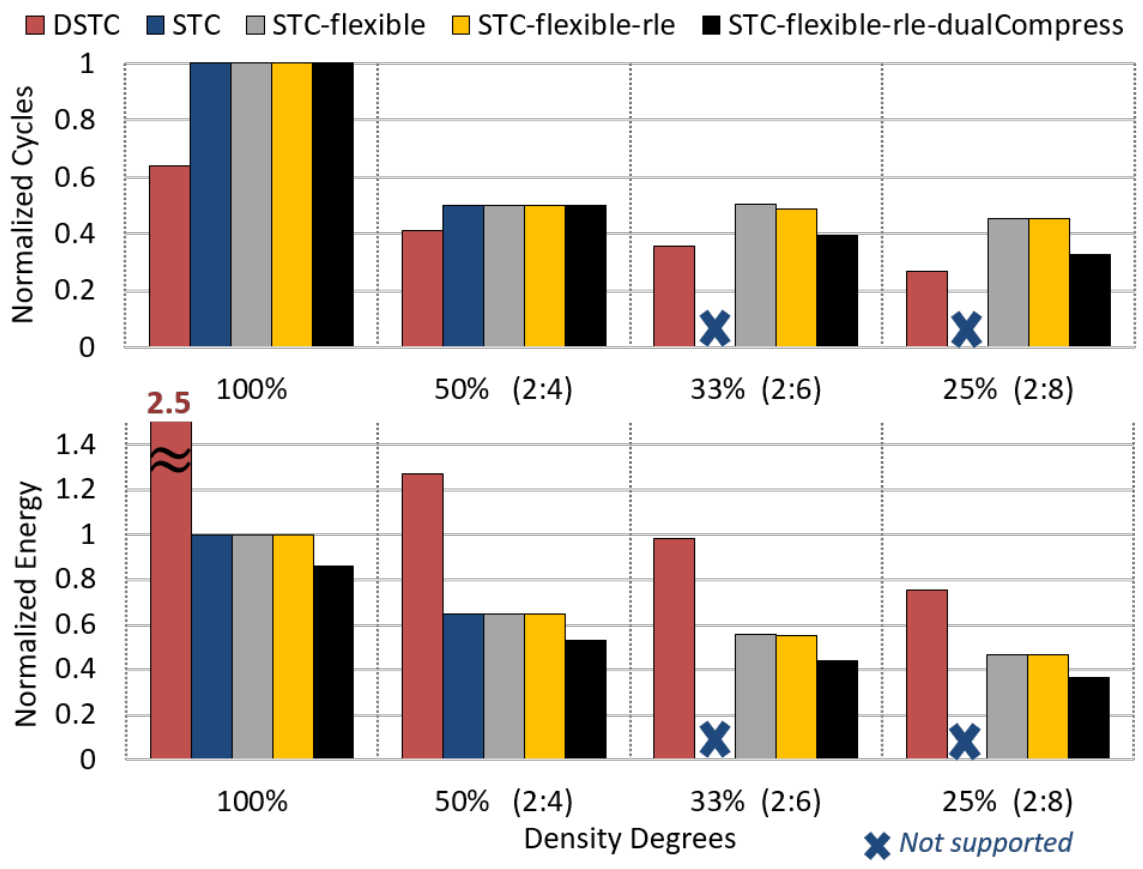


Figure 4-11: Sparseloop's analysis on the normalized total cycles spent and energy-delay product for various designs of tensor core accelerator running representative ResNet50 [36] layers pruned to various sparsity degrees. The accelerators are controlled to have similar amount of hardware resources.

case of a DNN, tensor A in Fig. 4-2 corresponds to the structured sparse weights in Fig. 4-10.

As shown in Fig. 4-10, 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. 4-10, 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. 4-11, 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).

Takeaway: Surprisingly, Sparseloop's modeling shows that the naively extending the STC design to support more sparsity ratios does not lead to theoretically achievable speedup.

Identify Design Limitations

STC-flexible's approach does not improve performance due to *SMEM* bandwidth limitation. Fig. 4-12 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. 4-12). 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. 4-12, 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).

Takeaway: Sparseloop identifies the design's performance is bottlenecked by its limited bandwidth.

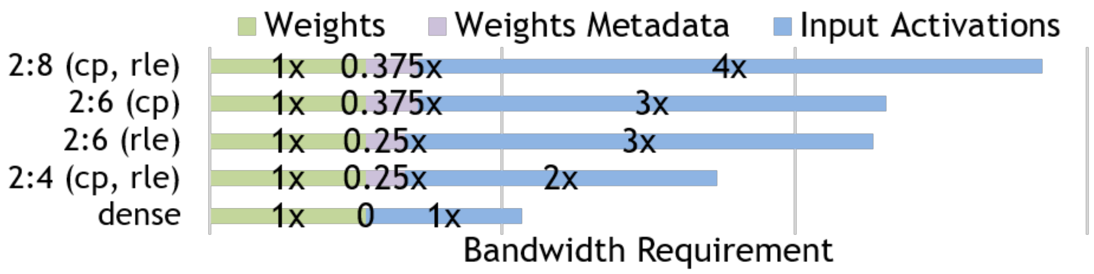


Figure 4-12: Sparseloop's analysis on bandwidth requirements for getting ideal speedup for various operands and associated metadata (if any).

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 (compression) format can alleviate the overhead introduced by metadata, especially for 2:6 structured sparsity. Thus, as shown in Fig. 4-11, 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. 4-11. 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. 4-11, *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. Overall, as shown in Fig. 4-11, we derived *STC-flexible-rle-dualCompress* that, compared to

DSTC, always introduces lower energy consumption and has a similar processing speed most of the time for the studied sparsity degrees.

Takeaway: Sparseloop identifies that bandwidth limitations can be alleviated by applying compressed format SAF to both operands in the workload.

4.9.2 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 be used to study other matrix-matrix multiplication accelerator designs for general sparse tensor algebra workloads. Specifically, we model workloads with more diverse sparsity degrees and matrix-matrix multiplication 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 for different applications: (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.

Design Choices

Workloads: We use matrix multiplication with sparse input tensors (spMspM) of various density degrees as example workloads. spMspM, represented as $Z_{m,n} = A_{m,k} \times B_{k,n}$ 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 4.5(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 4.5(b), we consider two sets of SAFs choices: (1) *Innermost-*

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

Skip 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.

Interactions Among Design Choices

Fig. 4-13 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 DNN 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 very sparse workloads (*e.g.*, with less than 1% density) since this design performs early off-chip traffic eliminations, but it fails to reduce EDP with DNN workloads due to its inability to perform effective off-chip intersections on denser

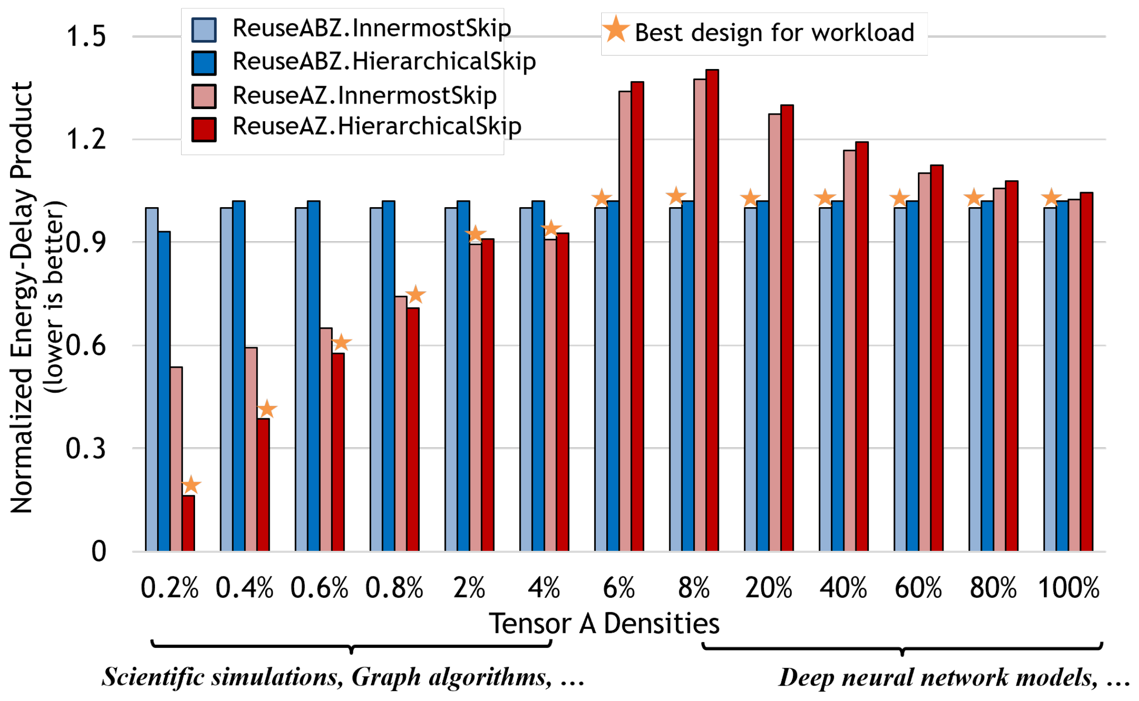


Figure 4-13: Normalized energy-delay product of different combinations of dataflow-SAFs running matrix multiplications with various density degrees, which are labeled with relevant example workloads. Sparseloop shows (1) dataflow and SAFs should be co-designed to ensure potential savings; (2) the correct combination needs to be chosen for different applications to realize the potential savings.

operands and its lack of on-chip B reuse. Thus, a design's dataflow-SAFs combinations need to be chosen based on the 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. 4-13, *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. 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.

Takeaway: Dataflow and SAFs need to be carefully co-designed to ensure there exist opportunities for reasonable savings.

4.10 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*.

4.10.1 Cycle-level models

One of the most popular classes of models is cycle-level models, which evaluate the detailed per-cycle behaviors of potential designs (*e.g.*, simulating the exact packets arriving at an NoC router for each cycle). Many of them perform register-transfer-level (RTL) analysis [103, 121, 70, 82], which include low-level hardware details (*e.g.*, pipeline stages, storage interface protocols). The RTL implementations can then be synthesized to target different platforms (*e.g.*, ASIC, FPGA). There are also models that perform cycle-level architectural analysis without RTL implementations (*e.g.*, STONNE [65] implement cycle-level simulation in C++, which captures much fewer hardware details).

The detailed simulations performed by such cycle-level models often lead to very accurate results, making them good candidates for obtaining a deep understanding of the design's performance. On the other hand, their fine-grained modeling often results in long simulation time [53, 12, 77], hindering early-stage design space exploration among the vast number of designs (e.g., a variety of dataflows to support). Furthermore, cycle-level models are often not well-parameterized in terms of architecture organizations, employed sparsity-related hardware optimizations, dataflow, etc. These assumptions often make it hard to employ the existing model to evaluate the diverse designs in the design space.

4.10.2 Analytical models

Unlike cycle-level models, analytical models perform higher-level evaluations without considering per-cycle processing details of the design. Nonetheless, analytical models still aim to capture the impactful activities of the components to produce accurate-enough modeling results. Since these models work on abstracted hardware modules, they are usually well-parameterized and modularized to support a wider range of architecture designs. However, as discussed in Sec. 2.3, we find none of the existing work systematically captures the impact of workload sparsity and various sparsity-related hardware implementations [77, 109, 47, 85, 60, 54, 124].

4.11 Conclusions

This chapter presents Sparseloop, a fast, flexible, and accurate modeling framework for sparse DNN accelerators. Based on the observation that the analyses of dataflow, SAFs, and micro-architecture are orthogonal to each other, we design Sparseloop's internal analysis as three decoupled steps to keep its modeling complexity tractable. To balance modeling accuracy and modeling speed, Sparseloop uses statistical characterizations of tensors. Sparseloop is over 2000× faster than cycle-level simulations, and models well-known sparse DNN 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). Overall, Sparseloop has become the first analytical modeling tool for sparse DNN accelerators. We expect Sparseloop to become a popular, extendable tool can greatly facilitate the quantitative understanding of various designs.

Chapter 5

Hierarchical Structured Sparsity

Numerous previous sparse DNN accelerator designs have proposed introducing various hardware-friendly sparsity patterns into DNNs with the goal to improve hardware performance while maintaining DNN accuracy. However, the existing sparsity patterns often lead to undesirable flexibility and efficiency trade-offs in hardware (*e.g.*, hardware designed for a specific sparsity pattern only efficiently translates one sparsity degree into hardware savings). In this chapter, we introduce the idea of *hierarchical structured sparsity (HSS)* to address such limitations in existing work.

5.1 Precise Sparsity Specification

To clearly compare existing sparsity patterns and distinguish our proposed HSS from existing work, it is necessary to precisely describe various sparsity patterns. However, conventional sparsity pattern classification approaches are often based on names that provide an informal characterization of just the dimensions on which the pattern is imposed, and thus fail to distinguish between different sparsity pattern proposals [6, 44] (*e.g.*, the term *sub-channel* is repeatedly used to describe many different patterns in Table 5.1).

To solve the problem, we propose a precise way of specifying various sparsity patterns based on the fibertree abstraction [94]. As shown in Table 5.1, our specification can easily distinguish between existing sparsity patterns and cleanly reflects the

Example Pattern	**Conventional Classification**	***Fibertree-based Specification*** *Rank (<rule>)...*
[34]	Unstructured	CRS(Unconstrained)
[37] (Fig. 5-2(a))	Channel	C(Unconstrained)$\rightarrow$R$\rightarrow$S
[72]	Sub-kernel	C$\rightarrow$RS(G:H), *with any G, H*
[68] (Fig. 5-2(b))	Sub-channel	RS$\rightarrow$C$_1$$\rightarrow$C$_0$(2:4)
[126]	Sub-channel	RS$\rightarrow$C$_1$$\rightarrow$C$_0$(4:16)
[64]	Sub-channel	RS$\rightarrow$C$_1$$\rightarrow$C$_0$(G$\leq$8:8)
Example Two-rank HSS (Fig. 5-3)	Sub-channel	RS$\rightarrow$C$_{N\text{-}1}$$\rightarrow$ C$_{N\text{-}2}$(3:4)$\rightarrow$...$\rightarrow$C$_0$(2:4)

Table 5.1: Informal conventional classification and the precise fibertree-based specifications for example sparsity patterns. For fibertree-based specificatons, ranks without pruning rules, *i.e.*, N/A rules in the figures, do not have (). Partitioned ranks are indicated by appending a number to the rank name, e.g., C is split into C$_1$ and C$_0$. Note that there could be multiple G and H values allowed for each rank.

properties of an example sparsity pattern that belongs to our proposed hierarchical structured sparsity (HSS).

5.1.1 Fibertree Abstraction

The fibertree abstraction [94] provides a systematic and precise way to express the sparsity in the tensors, *without getting into the complications about how the tensor is eventually stored in buffers (*e.g.*, compressed or uncompressed*). Since the sparsity specification focuses on understanding the nature of the sparsity rather than how it can be compressed, we use fibertrees as a basis for our proposed methodology.

For ease of presentation, we use the three-dimensional weight tensor in Fig. 5-1(a) as an example, which has C channels, R rows, and S columns. Fig. 5-1(b) shows the fibertree representation of the tensor. The fibertree has three levels, each of which is referred to as a *rank* and corresponds to a dimension of the tensor (*e.g.*, the lowest rank, *Rank0*, corresponds to dimension S). Each rank contains multiple *fibers*, each of which contains a set of *coordinates* and their associated *payloads*. For intermediate

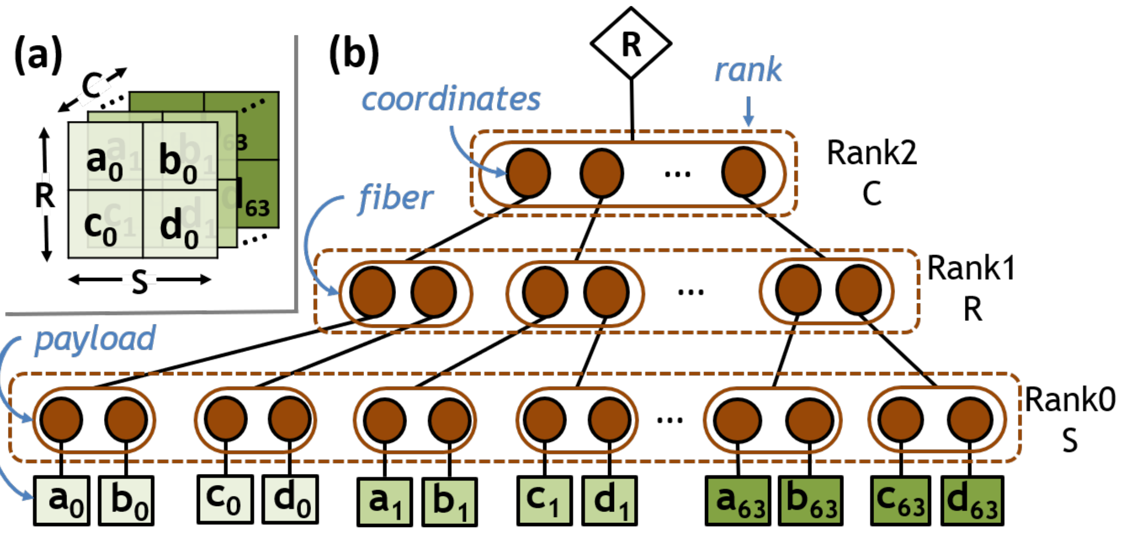

(a)
C
R
S
a_0
b_0
c_0
d_0
63
(b)
R
rank
coordinates
fiber
payload
Rank2
C
Rank1
R
Rank0
S
a_0 b_0 c_0 d_0 a_1 b_1 c_1 d_1 a_{63} b_{63} c_{63} d_{63}

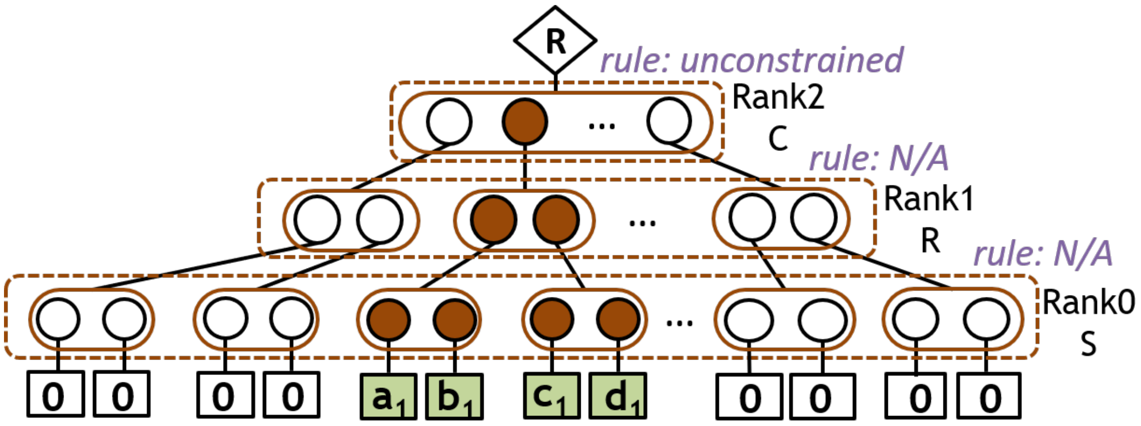
Figure 5-1: (a) Example dense weight tensor. C: channels, R: kernel height, S: kernel width. (b) Corresponding fibertree-based abstraction of the tensor. Each dimension of the tensor corresponds to a level of the tree, referred to as a *Rank*.

ranks, the payload is a fiber from a lower rank (*e.g.*, the first coordinate in *Rank1* has the first fiber in *Rank0* as its payload); for a coordinate in *Rank0*, the payload is a simple value (*e.g.*, the first coordinate in *Rank0* has the value a_0 as its payload). For a dense tensor, all of the coordinates exist and all of the payloads are nonempty (as indicated by the solid brown circles) or nonzero (as indicated by the green squares with alphabetical labels).

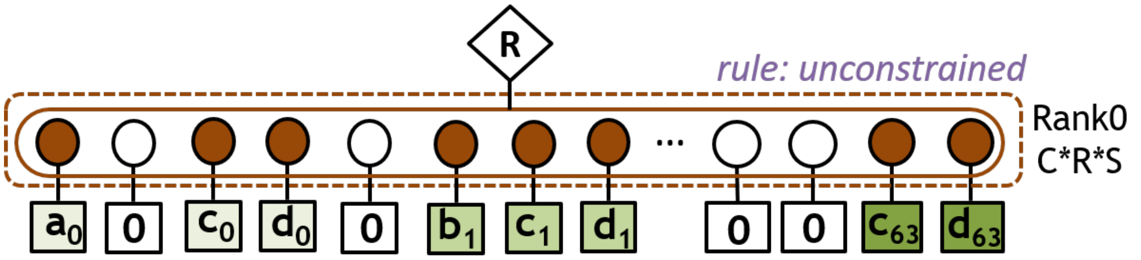
5.1.2 Fibertree-based Sparsity Specification

With the fibertree fundamentals presented, we now discuss how to use such an abstraction to describe sparsity in a tensor. Sparsity is introduced via pruning away the *coordinates* in the dense fibertree. At a high level, to define a specific sparsity pattern, a rank order needs to be specified and each *rank* is assigned a pruning *rule*. The rule specifies if the coordinates in each of its *fibers* can be pruned away; if so, whether there is a pattern that the per-fiber pruning should follow.

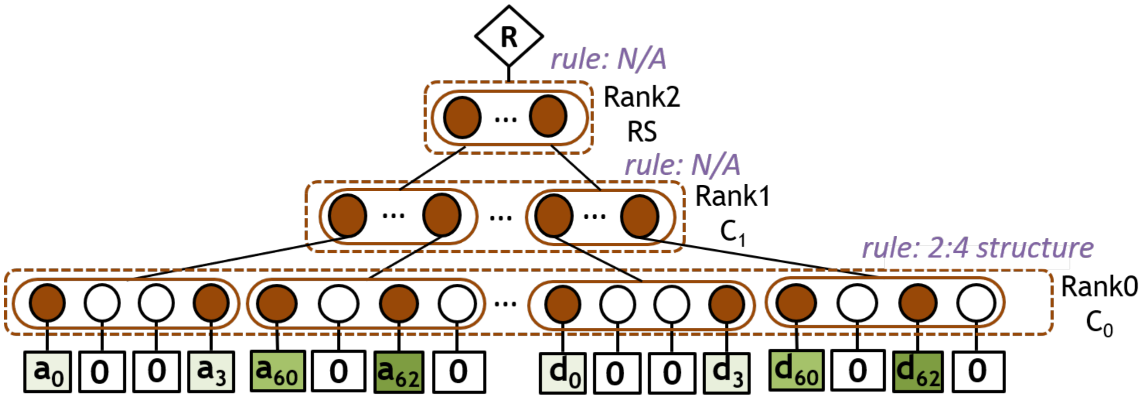
Coordinates in arbitrary ranks can be pruned away. Pruning a coordinate at the lowest rank simply removes values, whereas pruning a coordinate at intermediate ranks removes its fiber payload (*i.e.*, the entire subtree associated with each coordinate), implicitly pruning away all the associated lower-level coordinates. Because of this chained effect, the introduced sparsity is conventionally known as *structured sparsity*. Structured sparsity can have structures at different granularities, which are impacted by the rank at which the pruning rules are defined.



(a) $C(unconstrained) \rightarrow R \rightarrow S$ (channel-based structured [38])



(b) $CRS(unconstrained)$ (unstructured) [34]



(c) $RS \rightarrow C_1 \rightarrow C_0(2:4)$ (2:4 structured [68]

Figure 5-2: Fibertree-based representation of popular sparsity patterns applied to the tensor in Fig. 5-1(a)

For example, Fig. 5-2(a) shows the fibertree-based specification of the conventionally known *channel-based structured sparsity*, which demands arbitrary channels to be completely removed. The fibertree-based representation specifies the *unconstrained* pruning rule at the top rank C, as each removed coordinate corresponds to a removed channel, which is indicated by the empty circles. We specify such a structure as C(unconstrained)$\rightarrow$R$\rightarrow$S as shown in Table 5.1 and Fig. 5-2(a). The $\rightarrow$ defines the higher to lower rank order, and ranks with pruning rules carry (<rule>). The removal of the highest-rank coordinates results in all zeros in the corresponding channels. Since the removal of lower ranks is always implicit, the R and S lower ranks are not associated with any explicit pruning rules and are thus not followed by (<rule>) in the specification.

Furthermore, a sparsity pattern specification may involve altering the ranks in the original tensor tree. To illustrate the idea, Fig. 5-2b shows the fibertree-based specification of the conventional *unstructured sparsity*, which allows any coordinates associated with actual values to be pruned away. Since the unstructured sparsity pattern does not require any specific rank order, as shown in Fig. 5-2b, all the ranks are *flattened* into a single CRS rank, which is specified with pruning that allows unstructured removal of coordinates and thus turning the associated leaves into zero[1]. This representation is specified as CRS(unconstrained) as shown in Table 5.1 and Fig. 5-2(b).

A sparsity pattern specification can also involve *partitioning* the ranks for a conventionally known sub-channel-based 2:4 structured sparsity [73]. Fig. 5-2(c) shows the popular 2:4 structured sparsity that's supported by NVIDIA sparse tensor core. The original ranks are first reordered to have C as the lower rank. The R and S ranks are flattened together into one RS rank, and the C rank is then partitioned into two ranks, C_1 and C_0. The G:H-style sparsity structure is manifest by allowing at most 2 non-zero values in each fiber of the C_0 rank, which due to the partitioning have exactly four coordinates. For a specific fiber, we refer to the total number of

[1]When being processed on the hardware, the flattened rank can be re-partitioned (or tiled) to meet specific processing schedule supported by the acceelrator.

coordinates as its *shape* and the number of coordinates associated with nonzeros as its *occupancy*. Thus, the fiber shape in C_0 rank is defined by the denominator of the fraction (*i.e.*, H in C_0(G:H) structured sparsity), and the max fiber occupancy is defined by the numerator in the fraction. This sparsity pattern is thus specified as RS$\rightarrow C_1 \rightarrow C_0$(2:4).

As described in Table 5.1, our proposed fibertree-based specification allows precise descriptions of many existing work that were not easily distinguishable from conventional sparsity pattern classification techniques.

5.2 Hierarchical Structured Sparsity

With the fibertree-based sparsity specifications, it is clear that existing work in Table 5.1 all propose to apply different sparsity patterns to *one* rank. A natural approach to enhance such work to represent more sparsity degrees would be to introduce sparsity patterns to *multiple* ranks, motivating the concept of *hierarchical structured sparsity* (HSS).

5.2.1 General Concept

HSS allows *multiple ranks*, where each rank can its own sparsity pattern(s). Such a hierarchy of sparsity patterns could lead to structured sparsity with more sparsity degrees than only one rank with a very flexible sparsity pattern.

Since HSS allows more than one rank to be assigned with sparsity patterns, we introduce the parameter N which describes *the number of ranks with sparsity patterns assigned*. For each rank n where $0 \leq n \leq N\text{-}1$ and $N \geq 1$, a G:H pattern is assigned. G:H ratios can be different for different ranks. We call an instance of HSS that contains N ranks with sparsity patterns assigned an N-rank HSS[2].

[2]Note that the associated fibertree can have more than N ranks.

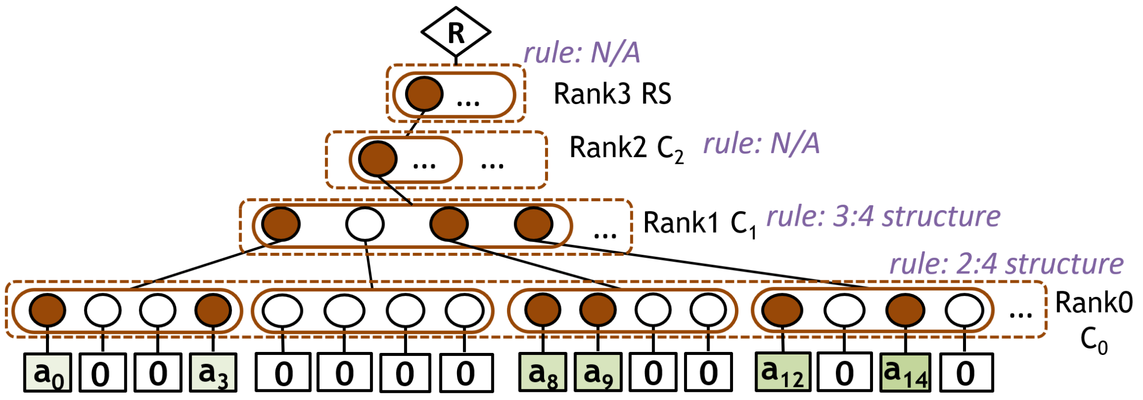


Figure 5-3: Fibertree-based specification for an example two-rank HSS, whose sparsity pattern can be described as RS→C_2→C_1(3:4)→C_0(2:4). Please note that, for general HSS patterns, the choices of rank ordering, flattening, and splitting are not limited to the ones shown in the example.

Fibertree-based Specification

To more concretely illustrate the idea of HSS, we will refer to the fibertree-based specification of an example two-rank HSS pattern for the rest of this section. However, for general HSS patterns, the choices of rank ordering, flattening, and splitting are not limited to the ones shown in the example.

As shown in Fig. 5-3, the example HSS pattern orders the ranks in R, S, C fashion, similar to the one in Fig. 5-2(b). Unlike in Fig. 5-2(b), where the original C rank is partitioned into two ranks, the example HSS pattern partitions the original C rank into three ranks C_2, C_1, and C_0, and assigns 3:4 and 2:4 to the lowest two ranks C_1, and C_0, respectively. Such an example two-rank HSS pattern can be described as RS→C_2→C_1(3:4)→C_0(2:4). As shown in Table 5-2, the specification has more than one rank with pruning rules assigned, resulting in qualitatively different sparsity patterns compared to existing works. In fact, many existing sparsity patterns [68, 126, 64] are all degenerate instances of HSS, i.e., one-rank HSS.

Sparsity Degrees

Different ranks in a multi-rank HSS have sparsity structures with different granularity. For example, in Fig. 5-3, the lowest C_0 rank's 2:4 structure is based on a single-value granularity. Whereas the higher C_1 rank's 3:4 structure is based on a larger granularity that's the shape of the lower fiber. Thus, the 3:4 ratio describes whether a fiber payload of each coordinate must contain all zeros or can contain nonzeros.

The overall sparsity degree of an HSS tensor can be derived from its sparsity structures at each rank. For example, the two-rank HSS RS$\rightarrow$$C_2$$\rightarrow$$C_1$(3:4)$\rightarrow$$C_0$(2:4) in Fig. 5-3 has a sparsity of $1 - \frac{3}{4} \times \frac{2}{4} = 0.625$. In general, the overall sparsity degree can be expressed as $sparsity = 1 - \prod_{n=0}^{N-1} \frac{G_n}{H_n}$, where G_n:H_n is the ratio assigned to rank n. Thus, by assigning a different number of ranks and different G:H ratios at each rank, HSS allows a flexible and systematic expression of various overall sparsity degrees.

Note: for ease of presentation, we will succinctly specify all sparsity patterns with only the ranks that have sparsity patterns assigned (*e.g.*, RS$\rightarrow$$C_1$$\rightarrow$$C_0$(2:4) is simplified to C_0(2:4)).

5.2.2 DNN Sparsification with HSS

Similar to all existing DNN sparsity patterns, HSS patterns can be introduced into DNNs to produce sparse DNN models. The goal for HSS-based sparsification is to ensure the most important nonzero values are preserved as much as possible.

To achieve the goal, we sparsify a dense tensor rank-by-rank in a lower-to-higher fashion. For example, for a C_1(3:4)$\rightarrow$$C_0$(2:4) HSS, we first apply the rank C_0's 2:4 pattern and then rank C_1's 3:4 pattern. For the lowest rank, we sparsify the values with the smallest magnitude. For an intermediate rank, we prune coordinates whose fiber payload has the smallest *scaled L2 norm*, defined as the average magnitude of all values in the payload. Depending on the per-rank sparsity patterns, the flexibility of HSS allows us to obtain sparse models with diverse sparsity degrees.

Since the introduced sparsity pattern is orthogonal to the pruning algorithm choice (*e.g.*, pruning on trained dense model [68], pruning from scratch [125], pruning with value revival [63], etc.), it is the algorithm designer's freedom to decide whether the ranks are sparsified at once or gradually sparsified over the process. As we will show in Sec. 6.4, even with a traditional pruning algorithm, DNN with HSS patterns can maintain reasonable accuracy.

5.3 Conclusions

In this chapter, we propose a novel DNN sparsity pattern, hierarchical structured sparsity (HSS), with the insight that we can systematically represent diverse sparsity degrees by hierarchically composing simple sparsity patterns. In addition, since simple sparsity patterns can be easily translated into hardware savings with low-overhead hardware implementations, the modularity of HSS enables efficient sparse DNN accelerator designs.

Chapter 6

HighLight Accelerator

Modern deep neural networks (DNNs) can have weights and activation that are sometimes dense and sometimes sparse with various sparsity degrees (*i.e.*, discrete values of sparsity). This phenomenon is a result of complex interactions among various DNN optimization techniques in a large DNN model design space. For example, activations can be dense or sparse based on the choice of activation functions (*e.g.*, ReLU [1] introduces sparse activations, whereas Mish [69] can result in much denser activations). Similarly, weights can be dense or sparse depending on how over-parameterized the model architecture is (*e.g.*, large models, such as ResNet50 [36], can sometimes be pruned to 80% sparsity while still maintaining accuracy, but compact models, such as EfficientNet [96], often requires dense weights to ensure accuracy).

As a result, we need a DNN accelerator that can translate any sparsity into efficiency, resulting in a good accuracy-efficiency trade-off. Specifically, the accelerator should be:

- ***Efficient***: incurs low latency, energy, and area overhead cost, referred to as having *low sparsity tax*, to implement the sparsity-related acceleration features. The sparsity tax can come from extra control logic, lack of data reuse, etc.

- ***Flexible***: supports *diverse* sparsity degrees (including dense). "Support" refers to two capabilities: **(i)** process the DNN to produce functionally correct results; **(ii)** translate weight and activation sparsity into reductions in energy and/or latency.

Specifically, the accelerator has two goals: **(i)** for medium/high-sparsity DNNs, eliminate *ineffectual operations* (*i.e.*, compute and data movement involving zeros) [39] to introduce energy and/or latency savings; **(ii)** for low-sparsity DNNs, have similar energy efficiency and latency as a purely dense accelerator (*i.e.*, have a low sparsity tax).

In this chapter, we will discuss our proposed HighLight accelerator[1], which allows simultaneously efficient and flexible processing of DNN workloads.

6.1 Motivation

6.1.1 Opportunities and Challenges

The zero values in sparse DNNs 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 [39, 113]. Thus, ineffectual computations introduce promising opportunities for the accelerators to eliminate unnecessary hardware operations (*i.e.*, buffer accesses and arithmetic calculations) to improve efficiency.

However, to translate such opportunities into hardware savings, the accelerator faces the challenge of providing hardware support to identify when both operands are non-zeros and evenly distribute them to parallel hardware components, referred to as *workload balancing.* Workload balancing is important to ensure high utilization of the available resources, thus allowing desired speedup. Often, the sparsity tax associated with such hardware support is highly related to the sparsity patterns that the accelerator aims to exploit.

Categories	**Repr. Designs**	**Sparsity Tax**	**Sparsity Degree Diversity**
Dense	TC [74]	N/A	N/A
Structured Sparse	STC [73]	**Very Low**	**Low**
	S2TA [64]	**Medium**	**Medium**
Unstructured Sparse	DSTC [107]	**High**	**Very High**
HSS	**Our Work**	**Low**	**High**

Table 6.1: Comparison of designs from different DNN accelerator design categories. *HSS* stands for hierarchical structured sparsity. An ideal design should have a low sparsity tax to achieve high efficiency and a very high number of supported sparsity degrees to achieve high flexibility.

6.1.2 Limitations of Existing Accelerators

Many sparse DNN accelerators [107, 78, 17, 15, 33, 24, 73, 64, 53, 126, 39, 76, 122, 81] have been proposed to exploit ineffectual computations to reduce data movement and compute for different sparsity patterns. At a high level, we can classify them into *unstructured sparse accelerators* and *structured sparse accelerators*. In the following sections, we discuss their limitations both qualitatively and quantitatively.

Unstructured Sparse Accelerators

Unstructured sparse accelerators have high flexibility to exploit arbitrarily distributed zeros with any sparsity degree in a wide range. However, the hardware support for unstructured sparsity introduces a high sparsity tax since it cannot make any assumptions about the locations of nonzero values when trying to identify and distribute the effectual computations. Existing unstructured sparse accelerators either pay for expensive intersections to identify the effectual computations (*e.g.*, SparTen [33] employs a prefix sum logic that occupies 55% of its processing element area), or employ dataflows that identify effectual computations without intersections but require large, and thus expensive, accumulation buffers to hold the now randomly distributed output (*e.g.*, the costly dataflow employed by DSTC [107]). Furthermore, the number of effectual computations varies significantly across sub-tensors within the workload and

[1]Named HighLight because it has two levels of gh structured sparsity.

across workloads, these accelerators can often only ensure perfect workload balance for a limited set of sparsity (*e.g.*, DSTC [107] only ensures perfect workload balancing among columns of compute units when a sub-tensor's occupancy is a multiple of 32).

Takeaway: unstructured sparse accelerators often support diverse sparsity degrees with high sparsity tax.

Structured Sparse Accelerators

Structured sparse accelerators target DNNs with structured sparsity, which refers to distributions of zeros with spatial constraints and is often introduced via *structured pruning* [37, 68, 72, 64]. Structured sparsity can have different spatial constraints for nonzero value locations. For example, one of the most popular structured sparsity patterns is the *G:H sparsity pattern*, which mandates (at most) G elements to be nonzero within a block of H elements, and thus results in a density of G/H. For example, NVIDIA Sparse Tensor Core (STC) [73] employs a 2:4 pattern, which sparsifies two elements in every block of four elements [68], resulting in 50% sparsity[2].

The predetermined constraints for nonzero value locations in structured sparsity make it much easier for hardware to identify the locations of the nonzeros and evenly distribute them to parallel hardware components (*e.g.*, for G:H sparsity, the hardware can evenly assign G nonzeros to G compute units to balance the workload). As a result, accelerating a specific pattern is often efficient with a very low sparsity tax. However, existing structured sparse tensor accelerators often only accelerate a very limited set of sparsity patterns (*i.e.*, a few sparsity degrees). For example, STC [73] is only able to exploit the 2:4 pattern, whereas S2TA [64] exploits a few $G : 8$ patterns with design-specific constraints. For example, G has to be ≤ 4 for one of the operands (*i.e.*, weights or activations).

Takeaway: structured sparse accelerators often incur low sparsity tax but only support a few sparsity degrees.

[2]STC allows more than two zeros for each block of four values, but its compression encoding only allows 50% sparsity to be exploited.

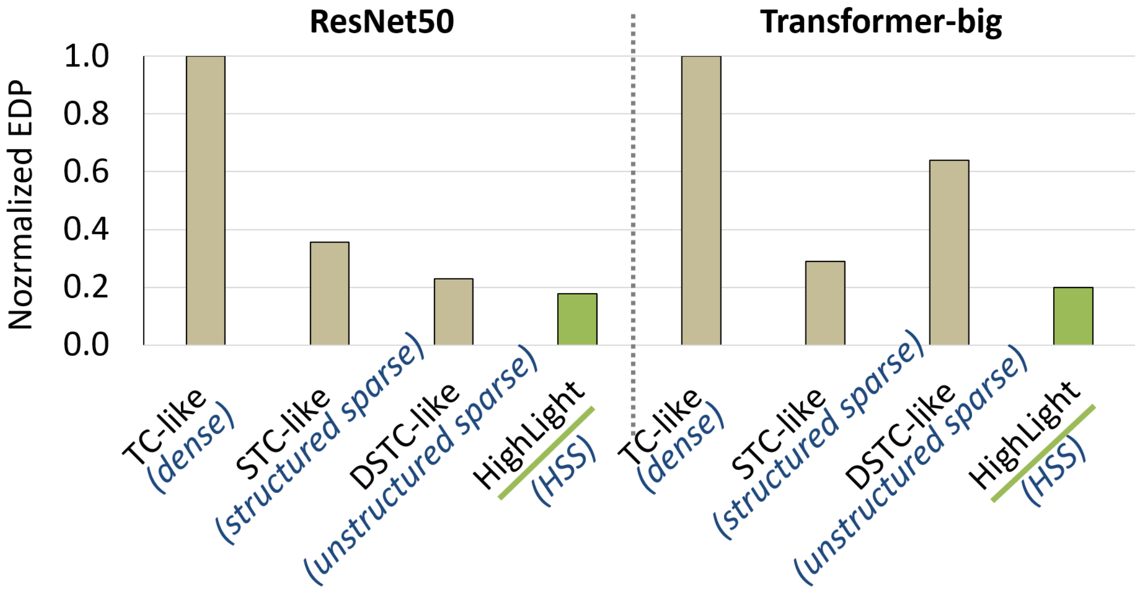


Figure 6-1: Normalized energy-delay product (EDP) of accelerators running two types of DNNs, pruned Transformer-Big [102] and pruned ResNet50 [36]. While ensuring similar accuracy (within 0.5% difference), the DNNs were structured pruned for STC [73] and HighLight (our work) and unstructured pruned for DSTC. For both models, HighLight achieves the lowest EDP while ensuring similar accuracy.

Quantitative Comparison

To concretely demonstrate the limitations of each class of accelerators, without loss of generalizability, we quantitatively compare representative designs allocated with similar hardware resources: **(i)** ***DSTC-like*** **[107]**: targets unstructured sparse DNNs with a high sparsity tax introduced by its costly dataflow; **(ii)** ***STC-like*** **[73]**: targets DNNs with weights that are dense or 2:4 sparse, introducing low sparsity tax.

To compare the designs, we normalize their energy-delay-product (EDP) to a dense accelerator, *TC-like* [74]. In particular, we evaluate their EDP running two different DNN architectures, *Transformer-Big* [102] and *ResNet50* [36]. For each DNN architecture, while ensuring similar accuracy (which we define as <0.5% difference), *TC-like* runs the dense version of the model, *STC-like* runs a structured pruned version, and *DSTC-like* runs an unstructured pruned version. Fig. 6-1 shows the EDP of the three designs running different DNNs.

Inflexibility of Structured Sparse Designs: As shown in Fig. 6-1, *STC-like* is outperformed by *DSTC-like* when running *ResNet50*. This is because *STC-like* is designed to only allow a maximum of 2× speedup with the 2:4 sparsity pattern. Furthermore, even if *ResNet50* has ∼60% sparse activations, *STC-like* cannot exploit activation sparsity for speedup. On the other hand, *DSTC-like* is able to translate the sparsity in both weights and activations into reductions in both processing speed and energy consumption. Thus, even if *STC-like* has a low sparsity tax, when running *ResNet50*, its inability to translate various sparsity degrees into hardware savings results in higher EDP than *DSTC-like*.

Inefficiency of Unstructured Sparse Designs: *DSTC-like* is outperformed by *STC-like* on *Transformer-Big*, as shown in Fig. 6-1. This is because *DSTC-like*'s outer-product-based dataflow with expensive accumulation buffer has a high sparsity tax. Since *Transformer-Big* has less than 10% average sparsity in activations, *DSTC-like*'s savings are overshadowed by its expensive hardware support. Thus, even though *DSTC-like* has high flexibility, when running *Transformer-Big*, its inefficient sparsity support with high overhead results in higher EDP than *STC-like* does.

Takeaway: there is no existing sparse accelerator that always has lower EDP for both DNNs due to their respective limitations.

6.1.3 Need for a Flexible and Efficient Design

To develop a flexible accelerator that is efficient for various DNNs with diverse sparsity degrees, *we are in great need of a general design that is simultaneously flexible and efficient*. However, as already demonstrated by the *DSTC-like* and *STC-like* comparisons above, many existing sparse DNN designs tend to trade flexibility for efficiency or vice versa, thus facing the challenge of not being able to meet both requirements.

To address this problem, we introduce a hardware-software co-design approach motivated by a novel class of sparsity patterns: **hierarchical structured sparsity (HSS)**, which leverages multiple levels of G:H structured sparsity to express diverse sparsity degrees in a multiplicative fashion. As shown in Fig. 6-1, while maintaining accuracy with our HSS-based sparsification, our low-sparsity-tax HSS-based hardware accelerator, HighLight, always provides lower EDP.

6.2 HighLight Overview

HSS unveils an organized HSS-based hardware design space, where each design can be systematically developed by considering three aspects for each HSS operand:

- G:H patterns supported at a rank.
- the number of HSS ranks supported by the hardware.
- the acceleration techniques, referred to as *sparse acceleration features (SAFs)* [113], supported at each rank.

In this section, we motivate HighLight's high-level architecture by discussing the impact of making different design decisions for the above design aspects.

6.2.1 Impact of Supported SAF at Each Rank

The accelerator can have different supported SAFs at a rank to translate sparsity into different savings. Specifically, when there are ineffectual operations, the hardware can employ

- **Gating:** lets the hardware stay idle to save energy. Gating often involves a trivial sparsity tax (*e.g.*, an AND gate).
- **Skipping**: fast forwards to the next effectual operation to save energy and time. Skipping incurs a higher sparsity tax (*e.g.*, muxing logic for leader-follower intersections).

Since gating is undesirable for many latency-sensitive applications, we will focus on discussing the impact of various design aspects assuming skipping as the supported SAF.

Skipping is highly reliant on high utilization of the components to achieve the desired speedup, so it is desirable to support G:H patterns with a fixed (set of) G that is a factor of the number of parallel hardware units (*e.g.*, with four processing elements (PEs), it is desirable to support G=4 patterns, as all PEs can be utilized with the four nonzeros in the block of H values, regardless of what H is). Thus, as shown in Fig. 6-3(a), the example designs with skipping SAFs support sparsity patterns with a fixed G value of two at each rank.

Takeaway: it's more desirable to support skipping, which favors G:H patterns with a G that's a factor of the available number of hardware instances to more easily ensure workload balancing with a high hardware utilization.

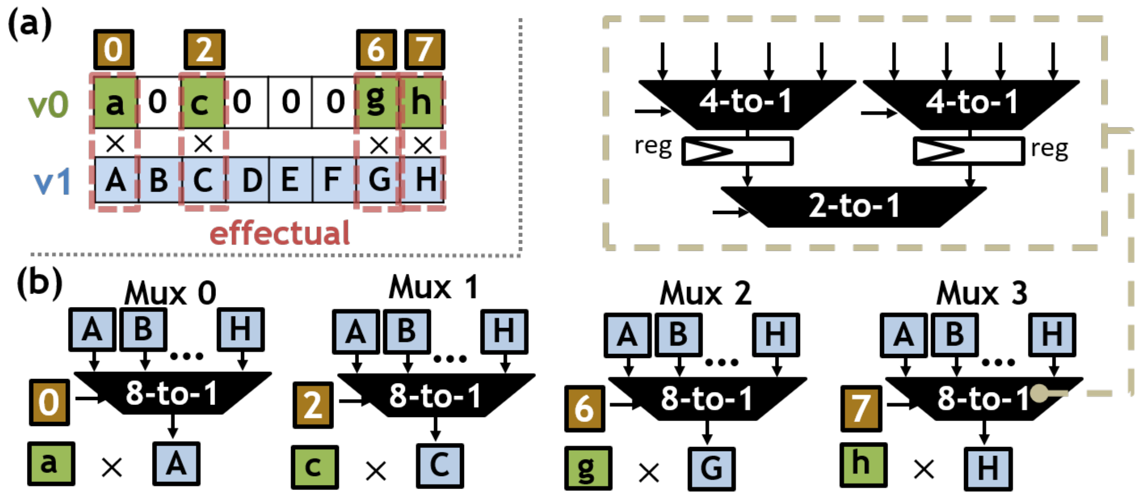


Figure 6-2: (a) Effectual computations in a dot product with 2:4 sparse vector 0 (v0) and dense vector 1 (v1). (b) Implementation of the muxing logic for selecting four values from a block of eight values.

6.2.2 Impact of Per-rank Supported Patterns

To implement skipping for a G:H pattern, additional hardware is needed. For example, Fig. 6-2(a) shows a dot product workload with two vectors: vector 0 (*v0*) is 4:8 sparse, and vector 1 (*v1*) is dense. In order to *only* perform effectual computations (*i.e.*, skip the ineffectual computations), the accelerator needs an 8-to-4 muxing logic to select the correct *v1* values. In specific, as shown in Fig. 6-2(b), the 8-to-4 muxing logic can be implemented with four 8-to-1 muxes. For example, the first 8-to-1 mux selects A based on a's coordinate 0. To ensure low latency, an 8-to-1 mux can be implemented with two 4-to-1 muxes pipelined with a 2-to-1 mux.

Since an accelerator can be designed to support multiple G:H patterns with different H values, the muxing sparsity tax increases as the largest supported H value (*i.e.*, H_{max}) increases. In specific, in order to support all possible patterns, the accelerator needs G number of H_{max}-to-1 muxes.

Takeaway: with a fixed G, the energy and area sparsity tax increases approximately linearly with $\mathbf{H_{max}}$.

6.2.3 Impact of Supported Number of Ranks

Given a target flexibility, supporting more HSS ranks reduces the H_{max} at each rank, reducing the sparsity tax. In specific, due to the nature of fraction multiplications, multi-rank HSS can easily represent a large number of sparsity degrees with a much smaller H_{max} at each rank by exploiting the composability of sparsity patterns. As shown in Fig. 6-3(a), with both designs supporting 15 different sparsity degrees across 0% to 87.5%, compared to the one-rank HSS design ***S***, which requires a H_{max} of 16, the two-rank HSS design ***SS*** only requires H_{max} of 8 at Rank1 and a H_{max} of 4 at Rank0.

The sparsity support of a multi-rank HSS design is implemented at different architecture levels, each of which target a specific rank (*e.g.*, as shown in Fig. 6-3(c), the processing element (PE) array level implements *Rank1 SAF* to exploit Rank1 patterns and the PE level implement *Rank0 SAF* to exploit Rank0 patterns). Fig. 6-3(b) shows the normalized sparsity tax for the two HSS designs in terms of their muxing overhead. Due to the reduced H_{max} values at each rank, ***SS*** introduces $> 2\times$ less muxing overhead (*i.e.*, lower sparsity tax), while achieving the same flexibility as ***S***.

Takeaway: Compared to the popular single-rank sparsity patterns supported in many existing works, hardware designed for multi-rank HSS can achieve the same flexibility with a much lower sparsity tax.

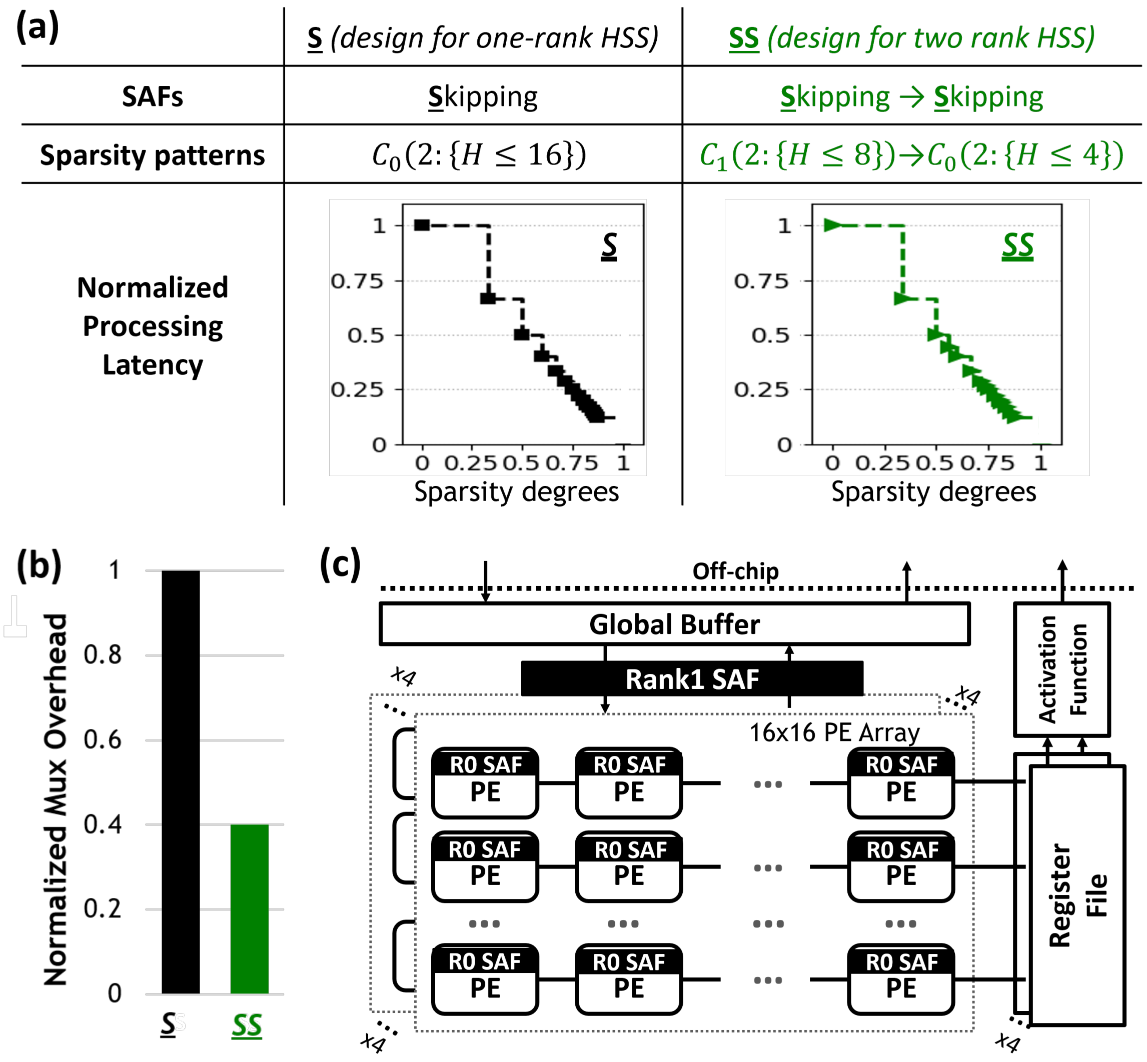


Figure 6-3: Comparison of designs with the same flexibility (15 sparsity degrees across 0%-87.5%) but different numbers of ranks. *SS* shows great potential for high flexibility and efficiency. (a) Design attributes and normalized processing latency (markers indicate the discrete sparsity degrees.) (b) Normalized muxing overhead. (c) High-level architecture of HighLight, with modularized SAFs for each rank at different architecture levels.

6.2.4 High Level Architecture

Fig. 6-3(c) shows the high-level architecture organization of our proposed HighLight accelerator, a simultaneously efficient and flexible accelerator consisting of a memory hierarchy and 1024 MACs grouped into four PE arrays. HighLight supports DNNs with two-rank HSS weights and unstructured sparse input activations. In terms of

SAF choices, HighLight implements modularized skipping SAF at different architecture levels to translate the two-rank HSS into energy and latency savings, and performs gating on input activation's sparsity to further reduce energy consumption.

We would like to point out that HighLight can be extended to support dual-HSS operands, which can both be exploited to introduce latency savings. We will present more discussions on the improvements as a case study in Sec. 6.4.5.

6.3 A Deeper Dive Into HighLight

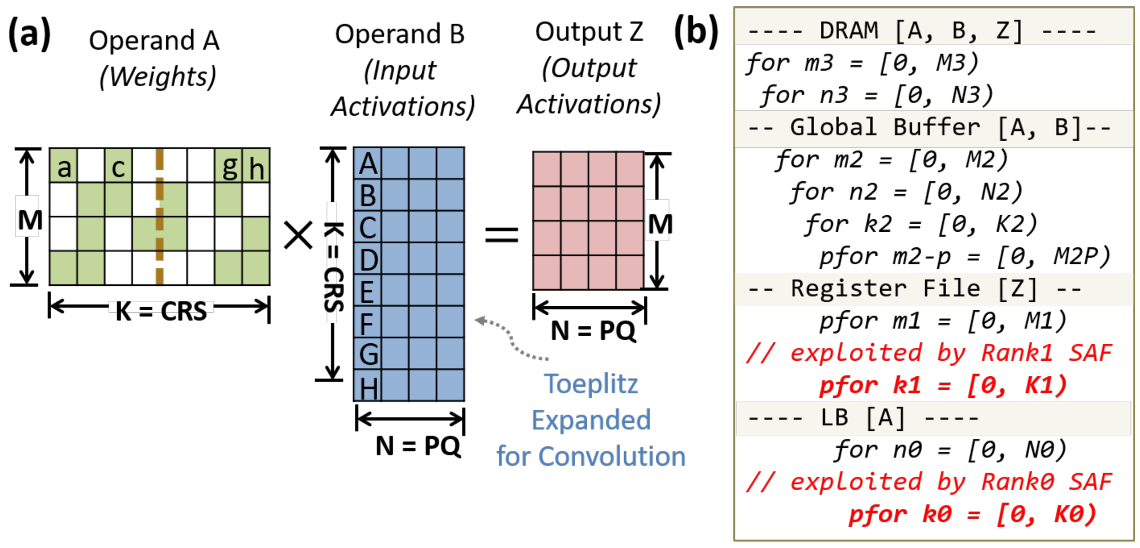
In this section, we present more details on HighLight's micro-architecture implementations. **For ease of presentation, we will use a down-sized architecture with two PEs and sparsity support for $\mathbf{C_1}$(2:{2≤H≤4})→$\mathbf{C_0}$(2:4) to discuss the core ideas of the HighLight micro-architecture.**

6.3.1 DNNs Processed as Matrix Multiplication

We design HighLight to process various layers in DNNs as matrix multiplications (MM) workloads, as many existing works do [73, 126, 64, 50, 74, 100]. Thus, DNN layers with MM kernels (*e.g.*, fully connected layers) are represented as they originally are. Whereas, as shown in Fig. 6-4(a), convolutional layers are represented as MM by flattening the weight dimensions and performing a Toeplitz expansion on the inputs [94] before sending to the accelerator for processing. Processing all layers as matrix multiplications implies interchangeable operands. Hence, instead of referring to the operands as weights and input activations, we refer to them as operands A and B, where operand A is dense or HSS, and operand B is either dense or unstructured sparse.

6.3.2 Compression Format

To correctly eliminate ineffectual hardware operations (*i.e.*, buffer accesses and computes associated with zeros), it is important to capture both ranks' sparsity structure


(a)
Operand A (Weights)
Operand B (Input Activations)
Output Z (Output Activations)
(b)
a c g h
M
K = CRS
A B C D E F G H
K = CRS
N = PQ
M
N = PQ
Toeplitz Expanded for Convolution

```
---- DRAM [A, B, Z] ----
for m3 = [0, M3)
 for n3 = [0, N3)
-- Global Buffer [A, B]--
   for m2 = [0, M2)
    for n2 = [0, N2)
     for k2 = [0, K2)
      pfor m2-p = [0, M2P)
-- Register File [Z] --
      pfor m1 = [0, M1)
// exploited by Rank1 SAF
      pfor k1 = [0, K1)
---- LB [A] ----
       for n0 = [0, N0)
// exploited by Rank0 SAF
         pfor k0 = [0, K0)
```

Figure 6-4: **(a)** Convolution represented as matrix multiplication with flattened operand A (weights) and Toeplitz expanded operand B (input activations) [94]. *M: number of filters; C: # of channels; R,S: height and width of filter kernels; P,Q: height and width of outputs.* **(b)** Loopnest representation of HighLight's dataflow.

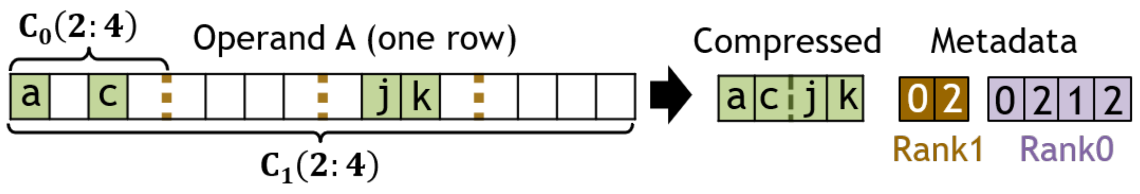


Figure 6-5: Hierarchical CP compression for operand A row.

with metadata. HighLight uses an offset-based coordinate representation (CP) [113] format to describe the position of nonzero values/non-empty blocks at each rank. Fig. 6-5 shows the metadata for an example C_1(2:4)→C_0(2:4) operand A tensor. For Rank0, each nonzero value carries a CP to indicate its position in its block of H_0 values (*e.g.*, since a is at the first position in its block, it carries a 0 metadata). For Rank1, each nonzero block carries a CP to indicate its relative position in the H_1 blocks (*e.g.*, the first and third blocks have nonzeros and thus carry upper-level metadata 0 and 2.)

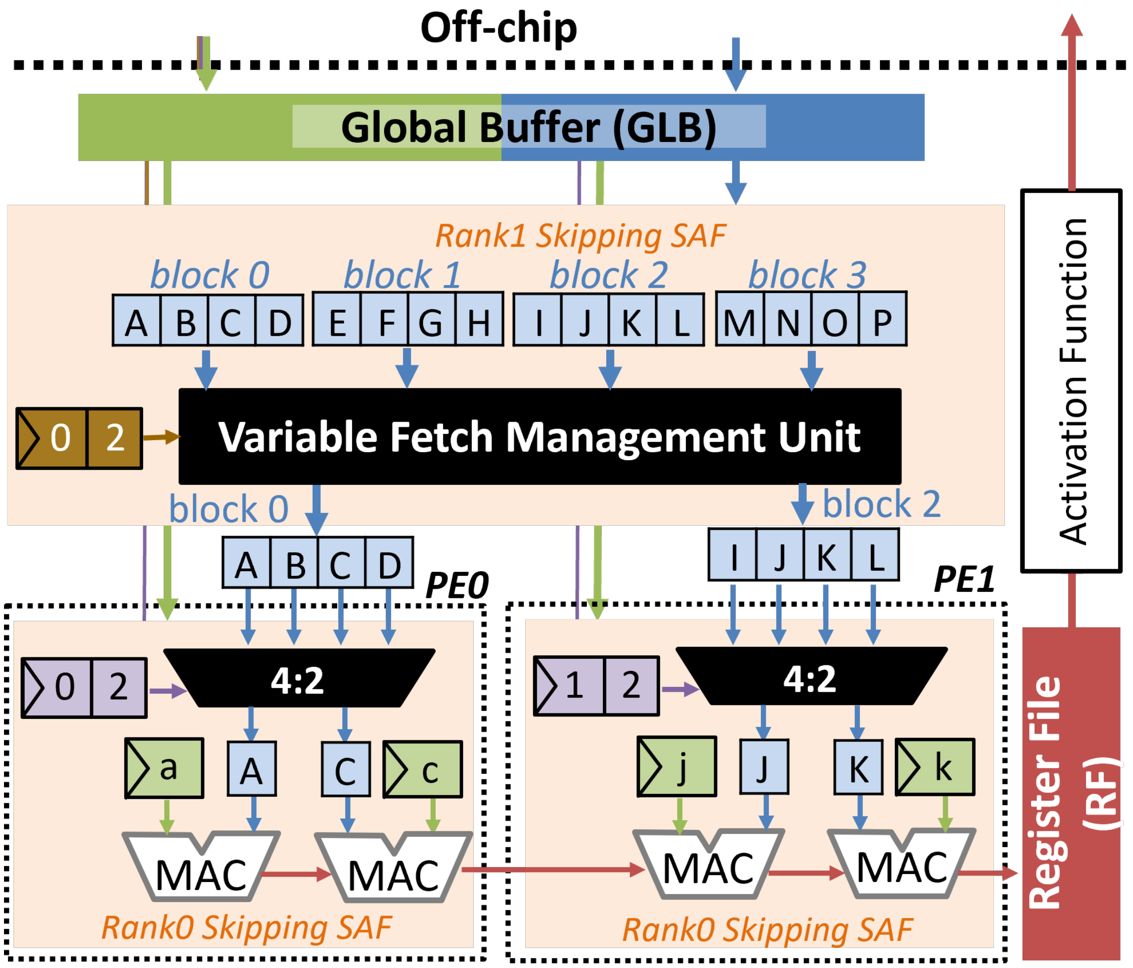


Figure 6-6: Down-sized architecture organization of HighLight with hierarchical skipping SAF. The showcased processing flow is for the $C_1(2{:}4)\rightarrow C_0(2{:}4)$ operand A in Fig. 6-5 and a dense operand B. Matched capitalized and lower case letters indicate corresponding values. Boxes with triangles are registers.

6.3.3 Hierarchical Skipping

To achieve high utilization of the hardware, and thus fast processing speed, HighLight employs a hierarchical skipping technique (*i.e.*, both *Rank1 SAF* and *Rank0 SAF* perform skipping based on their target rank's sparsity structure, as shown in Fig. 6-6). **Thus, HighLight's total speedup is the product of the speedup introduced at each rank.** To illustrate the ideas, we use the $C_1(2{:}4)\rightarrow C_0(2{:}4)$ operand A shown in Fig. 6-5 and a dense operand B as an example workload. We will discuss the support for a sparse B operand in Sec. 6.3.4.

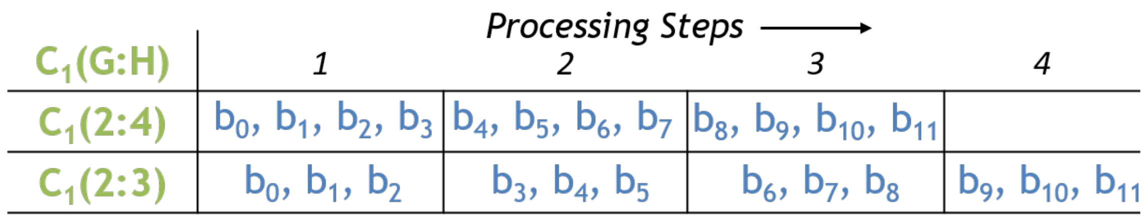
HSS-Operand Stationary Dataflow

Before diving into the SAFs, we discuss the general processing flow of HighLight by presenting its *dataflow*, which defines an accelerator's scheduling of data movement and compute in space and time [77]. To exploit the statically known sparsity structure to introduce desirable workload balancing, HighLight employs an HSS-operand stationary dataflow, where each Rank0 block of A is held stationary in each PE for reuse across different operand B values. As shown in Fig. 6-6, PE0 holds stationary in registers the two nonzero values a, c in the first block of operand A. Each MAC in the PE is responsible for working on one of the G nonzeros in its assigned block, in specific, the MAC on the left in PE0 works on a, and the right MAC works on c. The partial sums calculated at each MAC are first spatially accumulated across the PEs in the same row and updated to the *Register File*. More dataflow details are described in Fig. 6-4(b) based on the well-known loopnest representation [77, 15, 94].

Skipping SAF at Rank1

HighLight's *Rank1 Skipping SAF* exploits the sparsity structure in *Rank1* only. Specifically, it is responsible for only distributing non-empty Rank1 blocks in operand A and the corresponding blocks in operand B to the PEs for parallel processing. For example, as shown in Fig. 6-6, only the nonzeros in the first block (*i.e.*, a, c) and the third block (*i.e.*, j, k) in operand A are transferred to be processed at the PEs. Since only half of the *Rank1* blocks are non-empty in the example tensor, *Rank1 Skipping SAF* introduces a 2× speedup. Other sparsity patterns (degrees) in rank 1 can be exploited similarly to achieve different speedups.

Recall that *Rank1 Skipping SAF* is required to support sparsity patterns defined by $C_1(2{:}\{2\leq H\leq 4\})$. To maintain high utilization for different sparsity patterns (degrees), each of the two *PE*s in Fig. 6-6 should always get a non-empty block of operand A. To ensure correctness, different operand B data need to be selected for computation for different supported sparsity patterns. Fig. 6-7 shows the operand B blocks that need to be selected from at each processing step, with operand A having



Processing Steps →

C_1(G:H)	1	2	3	4
C_1(2:4)	b_0, b_1, b_2, b_3	b_4, b_5, b_6, b_7	b_8, b_9, b_{10}, b_{11}	
C_1(2:3)	b_0, b_1, b_2	b_3, b_4, b_5	b_6, b_7, b_8	b_9, b_{10}, b_{11}

Figure 6-7: Operand B blocks b_n to fetch at each processing step with operand A having C_1(2:4) or C_1(2:3).

HSS patterns with H=4 and H=3. We observe that different H values can require differently sized operand B blocks to be fetched for selection. For example, H=4 requires a fetch of four blocks of operand B at each processing step (*e.g.*, b_0, b_1, b_2, b_3 at processing step 0), whereas H=3 requires a fetch of three operand B blocks.

To avoid unaligned *GLB* fetches for different H_1 values, as shown in Fig. 6-6, HighLight's *Rank1 Skipping SAF* employs a *Variable Fetch Management Unit* (*VFMU*) to allow variable length streaming access, which is a technique commonly used for bitstream parsing (*e.g.*, in the entropy coding for video compression [16, 93]). Specifically, *VFMU* includes a small buffer that stores the $2\times H_{\max}$ blocks of operand B. The buffer is written with data that are fetched from *GLB* in an aligned fashion and can be configured with a *shift* signal to determine the offset position for the current read to start. Fig. 6-8 describes the data movement at *VMPU* for the first three processing steps when operand A has a G:H=2:3 sparsity at C_1. The *shift* signal is configured to three to allow the correct operand B blocks to be read out. Note that to keep the output width uniform, there are always four blocks read out of the VMPU. However, in the case of G:H=2:3, the last block is just a dummy padding that will never be selected by the muxing logic in *Rank1 Skipping SAF*. The *VMPU* processing is trivial to show for operand A with G:H=2:4, as all accesses are well aligned. Such variable fetch support allows correct operand B to be fetched for different 2:H structures.

With the correctly shifted blocks, to avoid implementing wide muxes that select from the entire blocks of data, VFMU employs 4-to-2 muxes to select the correct pair of start and end addresses using the metadata from operand A. The addresses are used to index into VFMU's internal registers.

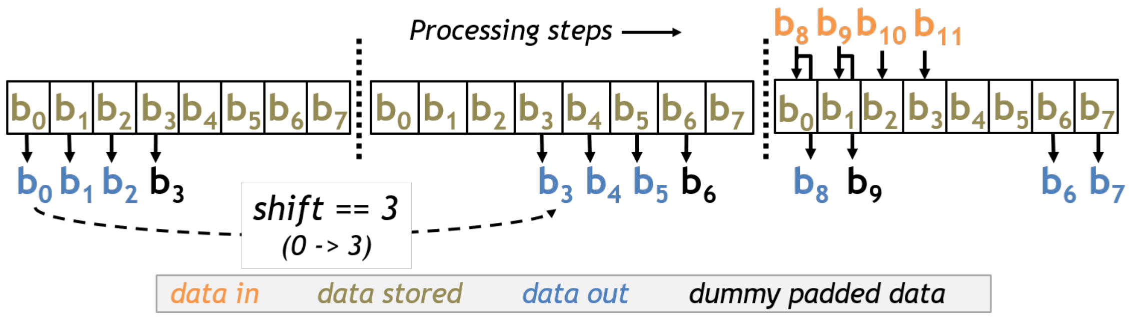


Figure 6-8: Operand B datamovement at *Variable Fetch Management Unit (VFMU)* for the first three processing steps when operand A has a *C_1(2:3)* sparsity. To output the correct operand B blocks, *VFMU* is configured to shift by three positions per read.

Skipping SAF at Rank0

As discussed above, each *PE* in HighLight works on a non-empty *Rank1* block with *C_0(2:4)*. To keep the two *MACs* busy in each *PE*, HighLight employs Rank0 skipping SAF with a 4-to-2 muxing logic. As shown in Fig. 6-6, based on *Rank0*'s CP metadata, the 4:2 mux in each *PE* selects the correct operand B for each *MAC*.

6.3.4 Exploiting Operand B Sparsity

So far, we have used a dense operand B in our example workloads to illustrate HighLight's processing flow. However, in DNN workloads, operand B for can be sparse due to non-linear activation functions (*e.g.*, ReLU) and/or activation pruning [64, 116]. HighLight exploits unstructured sparse operand B through gating and compression. Operand B is compressed within a blocked CSR fashion, carrying metadata that indicates the start and end addresses of each Rank1 block. Instead of always assuming the start and end addresses for a dense operand B, the VFMU processes the metadata of operand B to determine how much to shift for each block. The *MAC* unit in each PE gates the operation for ineffectual operations to save energy.

6.4 Experimental Results

In this section, we discuss our experimental setup and present results related to the co-designed hardware (*i.e.*, HighLight) and the software (*i.e.*, HSS-based sparsification).

6.4.1 Methodology

In this section, we introduce the baseline designs, workloads, and evaluation frameworks for hardware modeling and DNN pruning.

Baseline designs

Given the abundant prior designs, we compare HighLight to state-of-the-art representative designs in each category described in Table 6.1. Specifically, we evaluate

- TC [74] represents dense accelerators (*e.g.*, [74, 50, 13]).
- STC [73] represents single-sided G:H structured sparse accelerators (*e.g.*, [73, 126]).
- S2TA [64] represents dual-sided (*i.e.*, both operands) G:H structured sparse accelerators.
- DSTC [107] represents dual-sided unstructured sparse accelerators (*e.g.*, [107, 78, 17, 53, 33]).

Table 6.2 describes more details on each design's supported sparsity structures (if any) for each operand.

To ensure fairness, as shown in Table 6.3, we allocate similar storage and compute resources to all designs. Furthermore, all accelerators are designs that process DNNs as matrix multiplications. Since matrix multiplications accelerators treat the two operands interchangeably, we allow them to swap operands and report the best hardware performance (*e.g.*, since STC benefits from sparse operand A, we swap the operands if operand B is sparse and A is dense).

Design	**Supported Sparsity Patterns**	
	Operand A	**Operand B**
TC [74]	dense	
STC [73]	dense; $C_0(\{G\leq 2\}{:}4)$	dense
DSTC [107]	dense; unstructured sparse	
S2TA [64]	$C_0(\{G\leq 4\}{:}8)$	dense; $C_0(\{G\leq 8\}{:}8)$
HighLight (our work)	dense; $C_1(2{:}\{2\leq H\leq 8\})\rightarrow C_0(2{:}\{2\leq H\leq 4\})$	dense; unstructured sparse

Table 6.2: Supported sparsity structures for each design.

Design	**Storage**		**Compute**
	GLB	**RF**	
TC [74]	320KB	4×2 KB	4×256
STC [73]	$256 + 64$KB		
DSTC [107]	$256 + 64$KB		
S2TA [64]	$256 + 64$KB	64×64B	64×16
HighLight (our work)	$256 + 64$KB	4×2KB	4×256

Table 6.3: Hardware resource allocation. GLB is partitioned to data and metadata storage for sparse designs.

Workloads

We evaluate two classes of workloads:

- **Synthetic matrix multiplication** with operand A and B matrices that are 1024-by-1024, a common shape in DNN workloads. A and B are of various sparsity degrees: three different degrees for A: 0%, 50%, 75%, and four different degrees for B: 0%, 25%, 50%, 75%. Synthetic workloads allow us to capture the diverse sparsity characteristics in the DNN design space.
- **Representative DNN models** with distinct network architectures: the convolutional ResNet50 [36] and attention-based Deit-small [99] for image classification trained on ImageNet [25] and the attention-based Transformer-Big [102] for language translation trained on WMT16 EN-DE [7]. Actual DNN mod-

els allow us to take both accuracy and hardware performance impact into the picture.

Evaluation Frameworks

Accelerator Modeling: we use the Sparseloop-Accelergy infrastructure [113, 108] to model the accelerators. Sparseloop captures each accelerator's cycle counts and component runtime activities. We added a new density model to Sparseloop to capture the characteristics of HSS. To characterize energy and area costs, we built Accelergy estimation plug-ins for various components: (1) datapath components (*e.g.*, adders and SAF control): synthesized RTL with a 65nm PDK; (2) small SRAMs: 65nm SRAM compiler; (3) for large SRAMs not supported by our compiler: CACTI [28].

DNN Pruning: we use Condensa [49] to introduce various sparsity patterns to various DNNs, structured or unstructured. Since a good set of sparsity patterns should allow reasonable accuracy recovery even without novel or advanced pruning algorithms (*e.g.*, special ways to perform hyper-parameter searches), we reuse the pruning algorithm proposed for sparse tensor core (STC) [68]. Specifically, the algorithm for STC first statically prunes a pre-trained dense DNN by masking the appropriate weights and their gradients to zeros based on sparsity-pattern-specific sparsification rules (*e.g.*, the HSS-based rules in Sec. 5.2.2), and it then fine-tunes the masked DNN to regain accuracy. *To ensure fairness, all of our performed pruning follows the same algorithm, and the same set of hyperparameters is used for all sparsity patterns.*

6.4.2 HighLight Outperforms Prior Work

We compare HighLight to existing designs running synthetic workloads to demonstrate its flexibility and efficiency. In specific, its ability to always achieve high processing efficiency for workloads with varying sparsity degrees.

Fig. 6-9 compares the processing latency, energy consumption, and energy-delay product (EDP), a widely used metric for evaluating overall hardware performance in

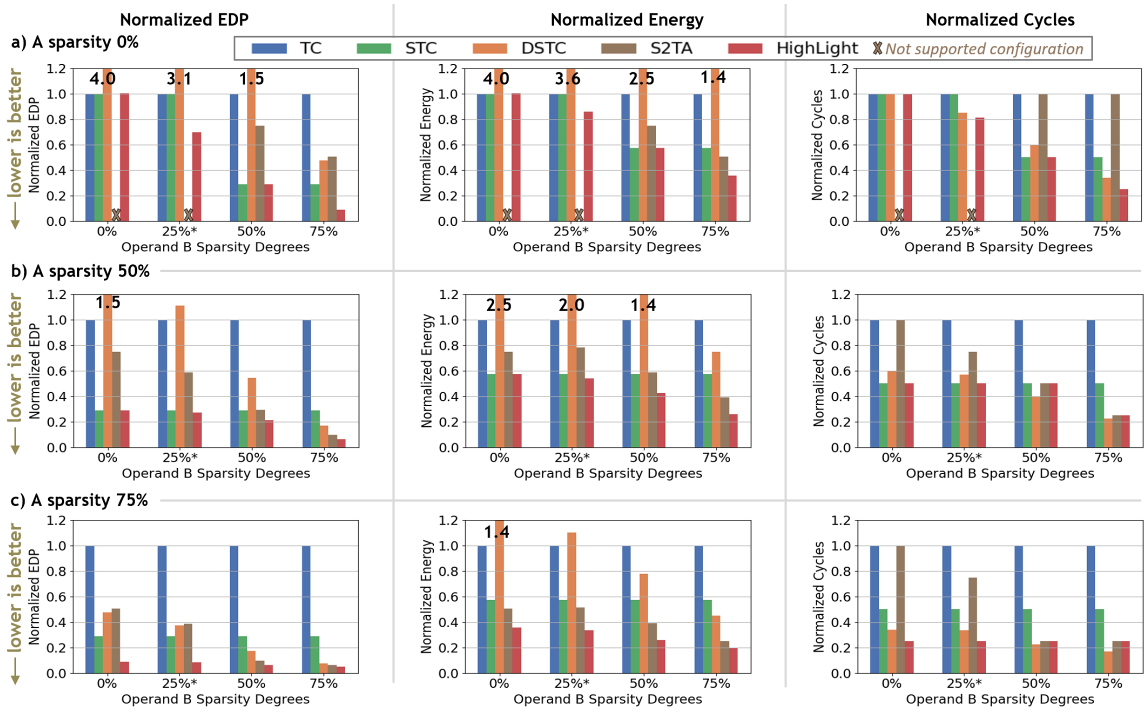


Figure 6-9: Comparison of existing designs running workloads with operands with different sparsity degrees. We compare the overall hardware efficiency energy-delay-product (EDP), energy and speed of the designs. S2TA [64] assumes both operands are structured. HighLight is always able to effectively exploit diverse sparsity degrees. **HighLight evaluated with 20% sparsity for conservative estimations.*

many existing works [45, 41, 61]. As shown in Fig. 6-9, different existing designs introduce inefficient processing at different sparsity degrees. Specifically, **(i) STC** employs simple acceleration with low sparsity tax for dense and 50% sparse workloads. However, STC's limited sparsity support fails to exploit the available opportunities for both speedup and energy for high sparsity workloads. **(ii) DSTC** introduces significant sparsity tax to identify effectual operations. In specific, it employs a dataflow that requires a costly accumulation buffer that is frequently accessed. Thus, DSTC's high sparsity tax masks the sparsity-related savings for workloads with low sparsity. Furthermore, DSTC also suffers from a not perfectly balanced workload due to the unpredictable nature of unstructured sparsity (*i.e.*, not all compute units are active). **(iii) S2TA** requires both operands to be structured sparse and has limited flexibility on the G values supported for each operand. For example, as shown in Table 6.3, S2TA requires one of the operands to have $\{G \leq 4\}$:8 (*i.e.*, cannot have more than 50%

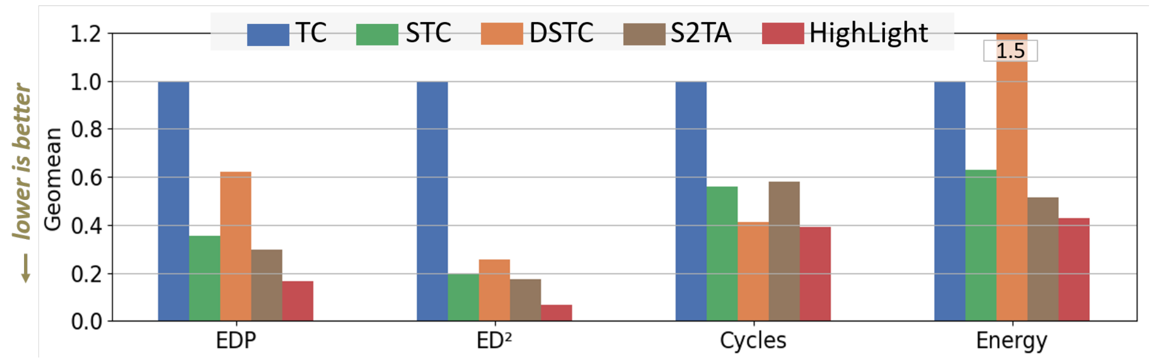


Figure 6-10: Geomean of various metrics. HighLight achieves the best geomean across all evaluated metrics.

sparsity). Thus, S2TA often fails to support or does not fully exploit the available speedup for workloads with low/medium sparse operands.

On the other hand, HighLight is always able to efficiently exploit various sparsity degrees. HighLight's per-rank skipping SAF and low-overhead hierarchical compression format introduce brings low sparsity tax, specifically low energy overhead. Furthermore, due to the structured sparsity, HighLight always achieves theoretical speedup with perfect workload balancing. Thus, as shown in Fig. 6-9, HighLight always achieves the best EDP and comparable-to-best processing speed for all evaluated sparsity degrees. HighLight achieves the best geomean for all evaluated metrics.

Fig. 6-10 shows each metrics' geomean across the evaluated workloads. **Compared to existing designs, HighLight achieves better geomean for all evaluated metrics.**

6.4.3 HighLight Provide Good Accuracy-Efficiency Trade-offs

To demonstrate HighLight provides good trade-offs between accuracy and efficiency, we compare the **EDP-accuracy loss relationship** of various design approaches, as shown in Fig. 6-11. In specific, we compare HighLight to multiple popular existing co-design approaches: **1)** dense (represented by the *TC* data points); **2)** unstructured sparse (represented by the *DSTC* data points); **3)** C_0(G:H) sparse (represented by the *STC* and *S2TA* data points).

We evaluate three representative DNNs: ResNet50 [36], Transformer-Big [102], and Deit-small [99]. For ResNet50, we prune all convolutional and fully-connected layers. For Transformer-big, we prune the feed-forward block and all projection weights. For Deit-small, we pruned the feed-forward block and the output projection weights. To ensure fairness, we use the same pruning algorithm as described in Sec. 6.4.1 for all of the sparsity patterns, structured and unstructured.

Fig. 6-11 shows the EDP-accuracy loss relationship for the three DNN models, with their weights pruned to different sparsity degrees. Ideally, we would like to always have very low EDP and accuracy loss. Unfortunately, low EDP often requires higher sparsity and thus leads to higher accuracy loss. The best design should excel at balancing the trade-off, thus always sitting on the Pareto frontier of the EDP-accuracy loss relationship.

Furthermore, the design should excel at the evaluated DNNs, which have different sparsity characteristics. Specifically, ResNet50 has much sparser activations than Transformer-big and Deit-small, and Deit-small has much fewer layers being pruned due to its already small parameter count (compared to other vision transformers).

As shown in Fig. 6-11, HighLight always sits on the Pareto frontiers. STC only delivers great accuracy-efficiency trade-off only at a single sparsity degree (*i.e.*, 50% sparse), S2TA fails to support attention-based models due to its incapability to process purely dense layers, and DSTC can introduce worse-than-dense EDP due to its high sparsity tax for the relatively dense models. **Thus, HighLight serves as a great candidate to support diverse DNNs with high hardware efficiency while maintaining a reasonable accuracy loss.**

6.4.4 Sparsity Tax Evaluation

Sparse DNN accelerators involve two types of sparsity tax: energy and area. Fig. 6-12(a) shows the energy cost breakdown across the buffers, compute, and SAF components in different architectures processing a workload with 75% operand A and a dense operand B (*i.e.*, one example set of bars from Fig. 6-9). Existing designs either do not fully exploit the sparsity for energy savings (*e.g.*, STC only recognizes upto 50% sparsity) or introduce inefficient dataflows to trade-off its insignificant SAF cost (*e.g.*, DSTC suffers from significant accumulation traffic at RF due to its outer produce style dataflow).

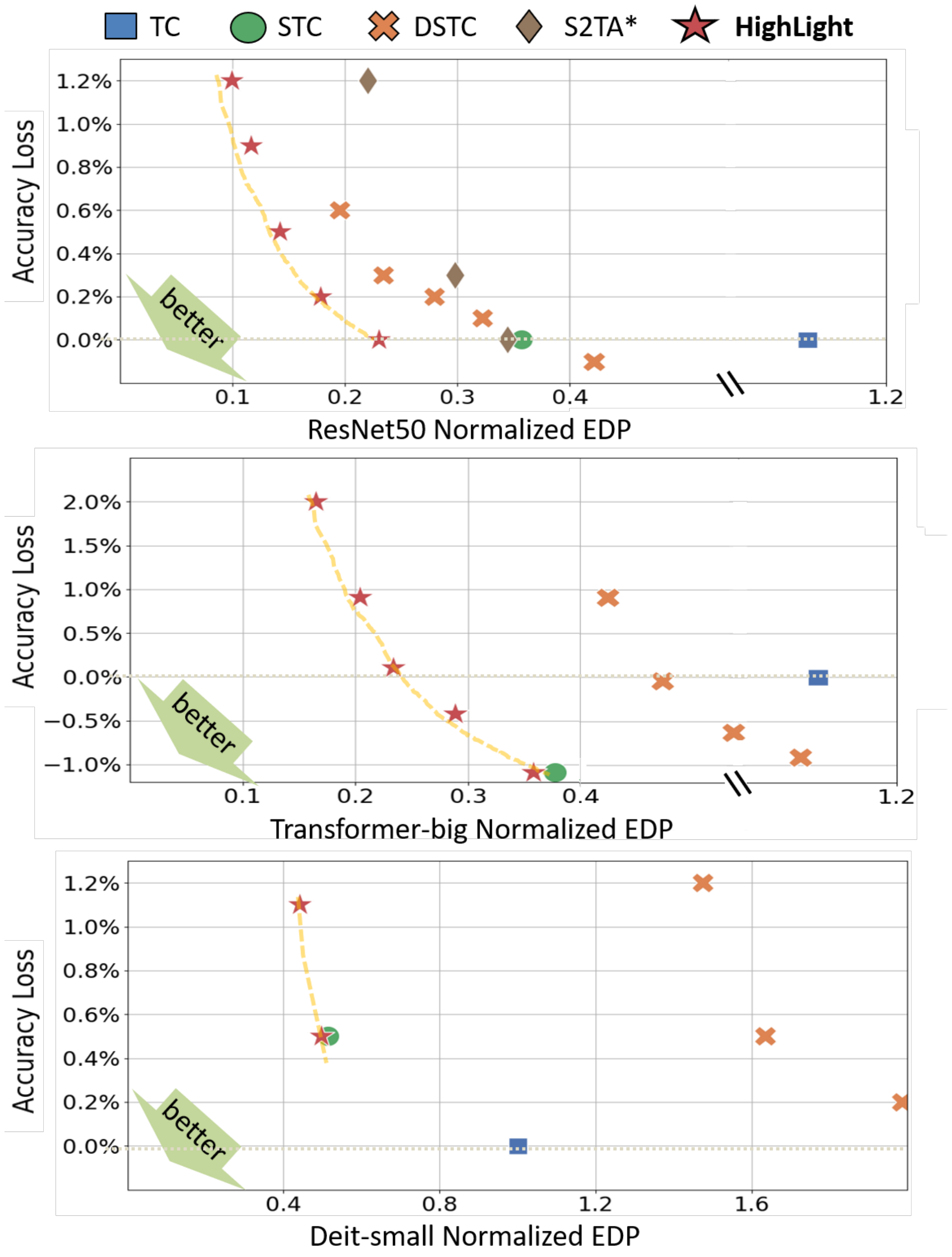


Figure 6-11: EDP-Accuracy Loss Pareto frontier for ResNet50 [36], Transformer-Big [102], and Deit-small [99]. Different markers refer to different accelerators. HighLight is always on the accuracy-EDP Pareto frontier and thus serves as a great candidate to support diverse DNNs with high hardware efficiency while maintaining accuracy.

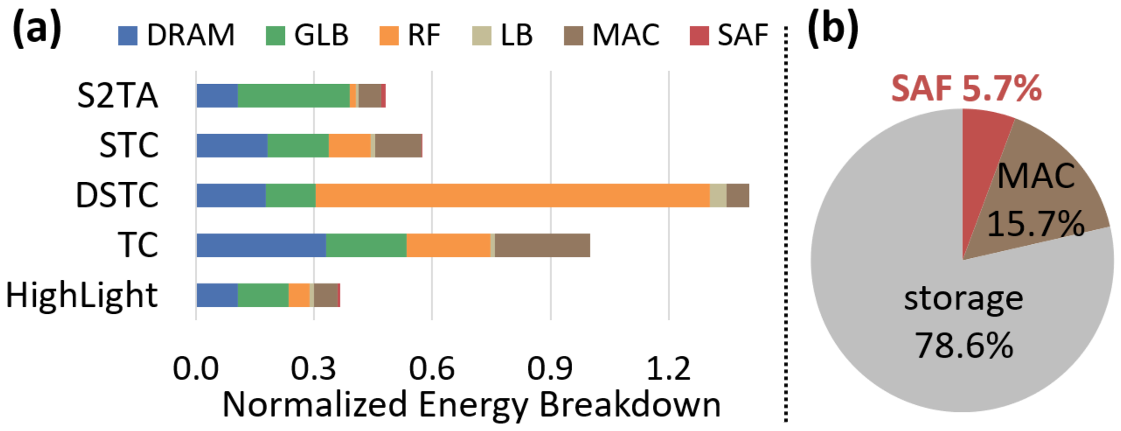


Figure 6-12: (a) Energy breakdown for 75% sparse operand A and dense operand B. (b) HighLight area breakdown. HighLight introduce low sparsity tax both in terms of energy and area.

Fig. 6-12(b) shows the area breakdown of HighLight, with the SAFs accounting for only 5.7% of the design's area. **Thus, HighLight has low energy and area sparsity tax.**

6.4.5 Case Study: Dual-Side Speedup

As briefly discussed in Sec. 6.2.4, instead of keeping input activations to unstructured sparse for energy savings only, one of the interesting improvements to HighLight is to support dual structured sparse operands and exploit both operands for speedup (*i.e.*, dual-side speedup). In this section, we demonstrate how we can realize such improvements.

We make the observation that an HSS-based accelerator can achieve dual-side speedup with easy workload balancing by supporting multi-rank HSS operands with alternating dense ranks (*e.g.*, weights with C_1(dense)$\rightarrow C_0$(2:4) and iacts with C_1(2:4)$\rightarrow$ C_0(dense)). When processing such workloads, the SAF at each rank only needs to perform dense-sparse intersections. In specific, for Rank1, skipping can be performed by intersecting C_1(2:4) in iacts with C_1(dense) in weights. for Rank0, skipping can be performed by intersecting C_0(2:4) in weights with C_0(dense) in iacts. Such dense-sparse intersections by nature lead to a perfectly balanced workload [24].

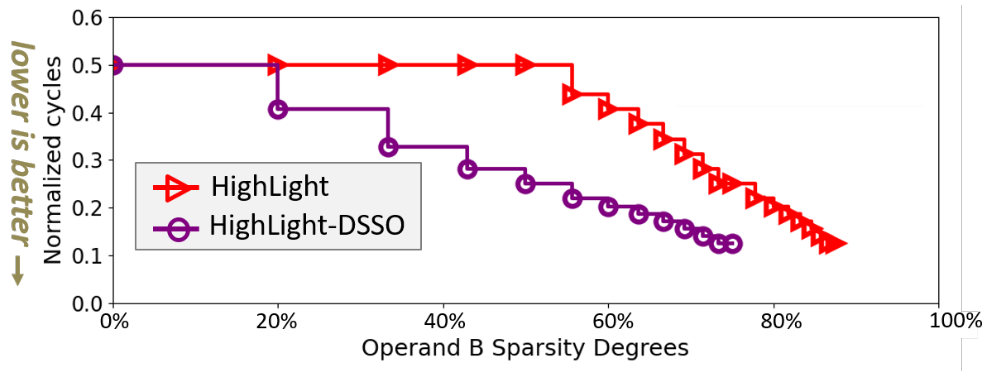


Figure 6-13: Normalized processing speed of HighLight and HighLight with dual side structured sparsity support, *i.e.*, *HighLight_DSSO*. DSSO support allows dual-side speedup for commonly shared sparsity degrees.

To implement such a design for workloads with dual side structure sparsity (DSSO), we enhance the HighLight's PE to allow skipping based on either operand by adding a simple 2:1 muxing logic, referred to as HighLight_DSSO. Fig. 6-13 compares the processing speed of the original HighLight design and *HighLight_DSSO* when a workload with operand A (weights) with C_1(dense)$\rightarrow C_0$(2:4) and operand B that follows C_1(2:2$\leq$H$\leq$8)$\rightarrow C_0$(dense).

As shown in Fig. 6-13, ***HighLight_DSSO* achieves 2× better processing speed compared to HighLight for the commonly supported sparsity degrees.** However, since *HighLight_DSSO* requires one rank to be dense to enable perfect workload balancing, there are fewer sparsity degrees supported for operand B by the design. In addition, although existing works have shown that it is possible for DNNs with DSSO to maintain accuracy [64, 116, 19], DNNs with DSSO may still require more advanced pruning techniques to recovery accuracy compared to single HSS operand DNNs.

6.5 Related Work

There is ample prior work in designing accelerators for efficiently processing sparse DNNs. These works either focus on co-designing sparsity patterns and hardware or solely focus on hardware for existing pruned models. Co-design approaches involve pruning DNNs to structured sparsity patterns that can be easily exploited by the underlying hardware. The target underlying system can be existing dense systems (maybe with relatively minor ISA updates) [72, 37, 20, 97, 125] (*e.g.*, GPUs) or custom accelerators [73, 126, 64, 59, 101] designed for the sparsity structure. The accelerators can be designed either with conventional digital technology or emerging technology (*e.g.*, processing-in-memory accelerators [101]). To better recover accuracy loss due to the enforced structure, some proposals have relatively relaxed structures and pre-process the pruned models into more compact structures before sending them to hardware (*e.g.*, pack unstructured columns into compact blocks [59]). Since structured sparsity has static nonzero values locations, the accelerators often have a low sparsity tax but low flexibility.

On the other hand, accelerators designed for existing pruned models or general sparse matrix multiplications often involve designing flexible sparsity support for unstructured sparsity [78, 17, 15, 33, 53, 39, 76, 122, 81, 118, 52]. Since supporting dynamic nonzero value locations requires extremely flexible hardware, these designs often focus on different dataflows that reduce complexity, efficient auxiliary components (*e.g.*, fast intersection unit [39]) that alleviate the significant control overhead, etc. Nonetheless, such accelerators often rely on the assumption that unstructured pruning can introduce high ($> 80\%$) sparsity to cancel out the cost of high sparsity tax.

The concept of hierarchy is also used in compressed data representations [52, 104, 39]. However, this line of work often focuses on better workload partitioning to enable efficient hardware processing, instead of using the hierarchy to provide flexibility and/or modularity, which is the goal of HSS. In fact, their proposed sparsity patterns are often unstructured or one-rank structured sparse (*e.g.*, SMASH [52] employs two levels of bitmask to represent unstructured sparse tensors), and target HPC/Graph analytics applications, which often have much higher sparsity degrees, and thus are less sensitive to high sparsity tax than DNNs.

6.6 Conclusions

Various optimization techniques introduce DNNs with weights and activations that are dense or sparse with varying sparsity degrees. The diversity challenges the assumptions made by existing DNN accelerators, which either focus on efficient but limited sparsity support or require very sparse DNNs to pay off the high sparsity tax required by their flexible sparsity support. This chapter addresses the importance of balancing accelerator flexibility and efficiency by proposing an accelerator, named HighLight, that exploits hierchical structured sparsity (HSS) introduced in Chapeter 5. Leveraging the modularity of HSS, HighLight achieves flexible sparsity support with low sparsity tax. As a result, HighLight is efficient for diverse sparsity degrees, including dense. In conjunction, we show that HSS allows DNN developers to prune DNNs to various sparsity degrees while maintaining the target accuracy. Compared to dense accelerators, HighLight achieves a geomean of $6.4\times$ (and up to $20.4\times$) better energy delay product (EDP) across layers with diverse sparsity degrees (including dense) and is at parity for dense DNN layers. Compared to sparse accelerators, HighLight achieves a geomean of $2.7\times$ (and up to $5.9\times$) better EDP and is at parity for sparse layers.

Chapter 7

Conclusion and Future Work

7.1 Summary of Contributions

Research on sparse DNN accelerator modeling and design promises significant hardware efficiency improvements. This thesis makes the following contributions:

- **Introduces a well-defined abstraction to describe high-level sparse DNN accelerator attributes.** Similar to how the widely used dataflow taxonomy [14] cleanly defines the high-level data movement choices for deep neural network accelerators, the proposed taxonomy of various sparse acceleration features provides a set of well-defined terminologies to describe how a sparse DNN accelerator exploits sparsity qualitatively, without getting into implementation details. Thus, our proposed taxonomy of sparse acceleration features facilitates succinct and effective communication between researchers from both the hardware and software communities.

- **Provides a systematic modeling tool to compare and gain insights on the impact of various sparse DNN accelerator designs**. Like other widely-used flexible modeling tools (*e.g.*, Gem5 [5] and GPGPU-sim [3]), the proposed Sparseloop tool provides a common platform for computer architects to easily compare various sparse DNN accelerators with user-defined parameterizations (*e.g.*, iso-area comparison of existing designs). We have seen how a popular, extendable tool can greatly facilitate the quantitative understanding of various designs, and we expect Sparseloop has the potential to serve this purpose for sparse DNN accelerators.

- **Enables easy and proper characterizations of each sparse DNN accelerator by facilitating fast mapspace explorations.** A proper accelerator characterization requires finding the best scheduling strategy (*i.e.*, the mapping), for each workload among thousands of available options. Thus, ensuring the speed of each potential mapping evaluation is important. Like existing widely used core modeling engines for mapspace exploration of other types of accelerators (*e.g.*, MAESTRO [60] for *dense* DNN accelerators), Sparseloop helps designers characterize each *sparse* DNN accelerator by quickly iterating over the large mapspace.

- **Provides an artifact with the potential of having broad use across academia and industry.** Sparseloop is built on top of Timeloop [77], which is already widely used for dense tensor accelerator evaluations. *Sparseloop received the MICRO-55 distinguished artifact award* by contributing to the community with experimental setups for evaluating various sparse DNN accelerator designs.

- **Demonstrates two important insightful solutions to challenges faced by many hardware modeling framework developers.**

  (1) Allow use of statistical characterizations for data-dependent analysis, instead of performing analysis based on the exact data, to balance simulation time and accuracy. This approach shortens simulation time significantly for data-intensive workloads (*e.g.*, DNNs and graph analytics).

  (2) Perform progressive modeling to maintain tractable complexity and high flexibility. Such a modeling style is particularly beneficial when the modeling task requires analyses of interactions among a considerable number of design aspects, which is often necessary for flexible hardware evaluation infrastructures.

- **Presents a systematic way to precisely define the sparsity patterns for effective comparison and exploration.**

  As more DNN sparsity patterns are proposed, it becomes increasingly important to find a systematic way to precisely define sparsity patterns to facilitate effective communication and help researchers more easily discover innovative patterns that are qualitatively different from existing patterns. The proposed fibertree-based specification allows a systematic and precise way to specify various existing sparsity patterns and motivated a novel sparsity pattern as proposed by the thesis.

- **Presents a novel class of sparsity patterns that enable efficient and flexible DNN accelerators and ensures DNN accuracy.**

  A desired DNN sparsity pattern should be hardware-friendly to allow efficient DNN acceleration with low overhead and flexible to represent diverse sparsity degrees. The proposed concept of hierarchical structured sparsity (HSS) meets both goals, with the key insight that we can systematically represent diverse sparsity degrees by having them hierarchically composed of simple sparsity patterns. HSS opens up great potential for both the hardware community to explore efficient designs and for the software community to develop pruning algorithms that target the HSS patterns.

- **Present a simultaneously flexible and efficient DNN accelerator for modern DNNs.**

  Motivated by HSS, this work presents an HSS-based accelerator design, called HighLight, from an organized HSS-based hardware design space. HighLight's modularized design methodology of using simple hardware to accelerate different sparsity patterns at different architecture levels enables simultaneously efficient and flexible processing of DNNs with diverse sparsity degrees. Such a design approach opens up a novel way to design flexible yet efficient DNN accelerators.

7.2 Future Work

There still exist many more opportunities in the topic of accelerator modeling and design.

- **Modeling of sparse fused-layer dataflows.**

  While Sparseloop is flexible to model many sparse DNN accelerators, it does not provide modeling support for designs with fused-layer dataflow [66, 92, 105, 10, 30], which is a class of accelerators that are gaining popularity in recent years. Since Sparseloop already provides important building blocks for sparse acceleration feature analysis, an interesting and impactful extension of the tool would be to support fused-layer dataflow modeling [32], taking sparse tensors and acceleration features into account.

- **Sparse DNN accelerator design with emerging technology.**

  Accelerator designs with emerging technology, such as in-memory/near-memory computing and optical computing, is a relatively new research area. Even though different technologies introduce different limitations, the core idea of eliminating hardware operations associated with ineffectual computations introduced by sparsity is still applicable such as accelerators. In addition, our proposed Sparseloop framework does have a flexible energy estimation backend that allows designers to take the special properties of such emerging technologies into account [109]. Thus, there are promising opportunities for designers to explore potential sparse DNN accelerators with emerging technology using Sparseloop.

- **Generalization of proposed contributions to sparse tensor accelerators beyond sparse DNN accelerators.**

  Sparse tensor algebra is a popular kernel in many applications beyond the ones employing DNNs (*e.g.*, graph algorithms [23, 57] and scientific simulations [123, 4]). Thus, there are many unsolved research questions on whether our proposed design space classification and modeling methodology can be applied to such accelerators. To facilitate such explorations, we have already provided an example Sparseloop use case (Sec. 4.9.2) of modeling and studying the interesting trade-offs for general sparse matrix-sparse matrix multiplication kernels for workloads with sparsity degrees that are much higher than DNNs, and thus can be accelerated with different approaches.

Appendix A

Sparseloop Artifact

In this appendix, we provide the source code of Sparseloop, its energy estimation backend based on Accelergy [108], 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.

A.1 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

A.2 Description

A.2.1 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`).

A.3 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.

A.4 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. 3-1).
- Validations on various sparse tensor accelerators (Fig. 4-8, Table 4.4, Fig. 4-9, and the STC design in Fig. 4-11.)
- Example design flow using Sparseloop to perform apples-to-apples comparison, identify design limitations, and explore various solutions to the limitation (Fig. 4-11).

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.

A.5 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).

A.6 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`

Bibliography

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016.

[3] Bakhoda et. al. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, pages 163–174, 2009.

[4] Peter Benner. Numerical linear algebra for model reduction in control and simulation. *GAMM-Mitteilungen*, 29(2):275–296, 2006.

[5] Binkert et. al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.

[6] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the state of neural network pruning? In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[7] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. Findings of the 2016 conference on machine translation. In *Proceedings of the First Conference on Machine Translation*, pages 131–198, Berlin, Germany, August 2016. Association for Computational Linguistics.

[8] Ralph-Uwe Börner, Oliver G Ernst, and Stefan Güttel. Three-dimensional transient electromagnetic modelling using rational Krylov methods. *Geophysical Journal International (Geophys. J. Int)*, 202(3):2025–2043, 2015.

[9] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector

multiplication using compressed sparse blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 233–244, 2009.

[10] Xuyi Cai, Ying Wang, and Lei Zhang. Optimus: An operator fusion framework for deep neural networks. *ACM Trans. Embed. Comput. Syst.*, 22(1), oct 2022.

[11] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. Origami: A convolutional network accelerator. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, page 199–204, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric approach for mapping deep learning operators on spatial accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–26, 2021.

[13] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '14, page 269–284, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Yu-Hsin Chen, Joel Emer, and Vivienne 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)*, pages 367–379, 2016.

[15] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(1):127–138, 2017.

[16] Yu-Hsin Chen and Vivienne Sze. A deeply pipelined cabac decoder for hevc supporting level 6.2 high-tier applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(5):856–868, 2015.

[17] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne 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)*, 9(2):292–308, 2019.

[18] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[19] Brian Chmiel, Itay Hubara, Ron Banner, and Daniel Soudry. Optimal fine-grained n:m sparsity for activations and neural gradients, 2022.

[20] Kyusik Choi and Hoeseok Yang. A gpu architecture aware fine-grain pruning technique for deep neural networks. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings*, page 217–231, Berlin, Heidelberg, 2021. Springer-Verlag.

[21] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. The a100 datacenter gpu and ampere architecture. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 48–50, 2021.

[22] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2:123:1–123:30, 2018.

[23] Timothy A Davis. Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[24] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo 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)*, pages 1110–1123, 2021.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[26] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, Yifan Gong, and Alex Acero. Recent advances in deep learning for speech research at microsoft. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8604–8608, 2013.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

[28] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement. In *2008 45th ACM/IEEE Design Automation Conference*, pages 554–559, 2008.

[29] A. Einstein. The foundation of the general theory of relativity. *Annalen der Physik*, 354(7):769–822, 1916.

[30] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 807–820, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Yizhao Gao, Baoheng Zhang, Xiaojuan Qi, and Hayden Kwok-Hay So. Dpacs: Hardware accelerated dynamic neural network pruning through algorithm-architecture co-design. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 237–251, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Michael Gilbert, Yannan Nellie Wu, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Looptree: Enabling exploration of fused-layer dataflow accelerators. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[33] Ashish Gondimalla, Noah Chesnut, Mithuna 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)*, pages 151–165, 2019.

[34] Song Han, Huizi Mao, and William 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)*, pages 1–14, 2016.

[35] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, Bryan Catanzaro, and William J. Dally. Dsd: Dense-sparse-dense training for deep neural networks, 2017.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv: 1512.03385*, 2015.

[37] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[38] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1389–1397, 2017.

[39] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 319–333, 2019.

[40] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, Columbus OH USA, October 2019. ACM.

[41] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. Mind mappings: Enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 943–958, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[43] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin. Nodetrix: a hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics (IEEE Trans Vis Comput Graph)*, 13(6):1302–1309, 2007.

[44] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.

[45] Mark Horeni, Pooria Taheri, Po-An Tsai, Angshu Parashar, Joel Emer, and Siddharth Joshi. Ruby: Improving Hardware Efficiency for Tensor Algebra Accelerators Through Imperfect Factorization. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022.

[46] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv: 1704.04861*, 2017.

[47] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *Proceedings of the 48th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 554–566, 2021.

[48] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, Jun-Seok Park, Heonsoo Lee, Dongwoo Lee, Myeong Woo Kim,

Hanwoong Jung, Heewoo Nam, Dongguen Lim, Seungwon Lee, Joon-Ho Song, Suknam Kwon, Joseph Hassoun, SukHwan Lim, and Changkyu 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)*, pages 15–28, 2021.

[49] V. Joseph, G. L. Gopalakrishnan, S. Muralidharan, M. Garland, and A. Garg. A programmable approach to neural network compression. *IEEE Micro*, 40(5):17–25, 2020.

[50] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[51] Jouppi et. al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, page 1–12, 2017.

[52] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 600–614, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Sheng-Chun Kao and Tushar 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)*, pages 1–9, 2020.

[54] Liu Ke, Xin He, and Xuan Zhang. Nnest: Early-stage design space exploration tool for neural network inference accelerators. In *Proceedings of the Interna-*

tional Symposium on Low Power Electronics and Design (ISLPED), pages 1–6, 2018.

[55] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[56] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 943–948, 2017.

[57] Scott Kolodziej, Mohsen Mahmoudi Aznaveh, Matthew Bullock, Jarrett David, Timothy Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software (J. Open Source Softw.)*, 4:1–4, 2019.

[58] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, volume 25, 2012.

[59] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 821–834, New York, NY, USA, 2019. Association for Computing Machinery.

[60] 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*, 40(3):20–29, 2020.

[61] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83. IEEE.

[62] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[63] Zi Lin, Jeremiah Zhe Liu, Zi Yang, Nan Hua, and Dan Roth. Pruning redundant mappings in transformer models via spectral-normalized identity prior, 2020.

[64] Z. Liu, P. N. Whatmough, Y. Zhu, and M. Mattina. S2ta: Exploiting structured sparsity for energy-efficient mobile cnn acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 573–586, apr 2022.

[65] Francisco Muñoz Matrínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators. *IEEE Computer Architecture Letters (CAL)*, (01), 2021.

[66] Linyan Mei, Koen Goetschalckx, Arne Symons, and Marian Verhelst. Defines: Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling, 2023.

[67] Linyan Mei, Pouya Houshmand, Vikram Jain, Juan Sebastian P. Giraldo, and Marian Verhelst. Zigzag: A memory-centric rapid DNN accelerator design space exploration framework. *arxiv:2007.11360*, 2020.

[68] Asit K. Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *CoRR*, abs/2104.08378, 2021.

[69] Diganta Misra. Mish: A self regularized non-monotonic neural activation function. *CoRR*, abs/1908.08681, 2019.

[70] 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)*, pages 575–580, 2016.

[71] Apple Newsroom. The future is here: Iphone x.

[72] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. *PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning*, page 907–922. 2020.

[73] NVIDIA. Nvidia a100 tensor core gpu architecture. Technical report, NVIDIA, 2020.

[74] NVIDIA. Nvidia v100 tensor core gpu architecture. Technical report, NVIDIA, 2020.

[75] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel Emer, and Christopher Fletcher. Accelerating sparse data orchestration via dynamic reflexive tiling. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 3, March 2023.

[76] 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)*, pages 724–736, 2018.

[77] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[78] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, 2017.

[79] Maurice Peemen, Arnaud A. A. Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 13–19, 2013.

[80] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–151, Providence RI USA, April 2019. ACM.

[81] 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)*, pages 58–70, 2020.

[82] 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)*, pages 1147–1152, 2017.

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

[84] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017.

[85] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar 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)*, pages 58–68, 2020.

[86] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A massively parallel coprocessor for convolutional neural networks. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 53–60, 2009.

[87] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[88] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2015.

[89] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 1–7, 2015.

[90] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru 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)*, pages 766–780, 2020.

[91] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting tensor cores via microbenchmarks: Latency, throughput and numerical behaviors. *arXiv: 12206.02874*, 2022.

[92] Arne Symons, Linyan Mei, Steven Colleman, Pouya Houshmand, Sebastian Karl, and Marian Verhelst. Towards heterogeneous multi-core accelerators exploiting fine-grained scheduling of layer-fused deep neural networks, 2022.

[93] Vivienne Sze and Anantha P. Chandrakasan. A highly parallel and scalable cabac decoder for next generation video coding. *IEEE Journal of Solid-State Circuits*, 47(1):8–22, 2012.

[94] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

[95] Emil Talpes, Douglas Williams, and Debjit Das Sarma. Dojo: The microarchitecture of tesla's exa-scale computer. In *IEEE Hot Chips 34 Symposium (HCS)*, pages 1–28, 2022.

[96] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[97] Yijun Tan, Kai Han, Kang Zhao, Xianzhi Yu, Zidong Du, Yunji Chen, Yunhe Wang, and Jun Yao. Accelerating sparse convolution with column vector-wise sparsity. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[98] The SciPy community. Scipy documentation.

[99] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, volume 139, pages 10347–10357, July 2021.

[100] Fengbin Tu, Yiqi Wang, Ling Liang, Yufei Ding, Leibo Liu, Shaojun Wei, Shouyi Yin, and Yuan Xie. Sdp: Co-designing algorithm, dataflow, and architecture for in-sram sparse nn acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2022.

[101] Fengbin Tu, Yiqi Wang, Ling Liang, Yufei Ding, Leibo Liu, Shaojun Wei, Shouyi Yin, and Yuan Xie. Sdp: Co-designing algorithm, dataflow, and architecture for in-sram sparse nn acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2022.

[102] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[103] 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)*, pages 1–8, 2019.

[104] Richard W Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications: First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005. Proceedings 1*, pages 807–816. Springer, 2005.

[105] Luc Waeijen, Savvas Sioutas, Maurice Peemen, Menno Lindwer, and Henk Corporaal. Convfusion: A model for layer fusion in convolutional neural networks. *IEEE Access*, 9:168245–168267, 2021.

[106] Francis Wang, Yannan Nellie Wu, Matthew Woicik, Joel S. Emer, and Vivienne Sze. Architecture-level energy estimation for heterogeneous computing systems. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 229–231, 2021.

[107] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. Dual-side sparse tensor core. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1083–1095, 2021.

[108] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[109] Yannan Nellie Wu, Vivienne Sze, and Joel 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)*, pages 116–118, 2020.

[110] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. Highlight: Efficient and flexible dnn acceleration with hierarchical structured sparsity, 2023.

[111] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Timeloop code base.

[112] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Sparseloop: An Analytical, Energy-Focused Design Space Exploration Methodology for Sparse Tensor Accelerators. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 232–234, March 2021.

[113] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Sparseloop: An analytical approach to sparse tensor accelerator modeling. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2022.

[114] Zi Yu Xue, Yannan Nellie Wu, Joel Emer, and Vivienne Sze. Accelerating sparse tensor algebra by overbooking buffer capacity, 2023.

[115] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko 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)*, pages 711–724, 2020.

[116] Qing Yang, Jiachen Mao, Zuoguan Wang, and Hai Li. Dasnet: Dynamic activation sparsity for neural network efficiency improvement. *CoRR*, abs/1909.06964, 2019.

[117] Tien-Ju Yang, Yu-Hsin Chen, Joel Emer, and Vivienne 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)*, pages 1916–1920, 2017.

[118] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 236–249, New York, NY, USA, 2019. Association for Computing Machinery.

[119] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel 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)*, page 687–701, 2021.

[120] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[121] 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)*, pages 1–8, 2018.

[122] 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)*, pages 261–274, 2020.

[123] M. Zhao, R.V. Panda, S.S. Sapatnekar, and D. Blaauw. Hierarchical analysis of power distribution networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 21(2):159–168, 2002.

[124] Yang Zhao, Chaojian Li, Yue Wang, Pengfei Xu, Yongan Zhang, and Yingyan 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)*, pages 1593–1597, 2020.

[125] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n: M fine-grained structured sparse neural networks from scratch. *arXiv preprint arXiv:2102.04010*, 2021.

[126] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan 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)*, pages 359–371, 2019.